INFORMATION TO USERS

This reproduction was made from a copy of a document sent to us for microfilming. While the most advanced technology has been used to photograph and reproduce this document, the quality of the reproduction is heavily dependent upon the quality of the material submitted.

The following explanation of techniques is provided to help clarify markings or notations which may appear on this reproduction.

1. The sign or “target” for pages apparently lacking from the document photographed is “Missing Page(s)”. If it was possible to obtain the missing page(s) or section, they are spliced into the film along with adjacent pages. This may have necessitated cutting through an image and duplicating adjacent pages to assure complete continuity.

2. When an image on the film is obliterated with a round black mark, it is an indication of either blurred copy because of movement during exposure, duplicate copy, or copyrighted materials that should not have been filmed. For blurred pages, a good image of the page can be found in the adjacent frame. If copyrighted materials were deleted, a target note will appear listing the pages in the adjacent frame.

3. When a map, drawing or chart, etc., is part of the material being photographed, a definite method of “sectioning” the material has been followed. It is customary to begin filming at the upper left hand corner of a large sheet and to continue from left to right in equal sections with small overlaps. If necessary, sectioning is continued again—beginning below the first row and continuing on until complete.

4. For illustrations that cannot be satisfactorily reproduced by xerographic means, photographic prints can be purchased at additional cost and inserted into your xerographic copy. These prints are available upon request from the Dissertations Customer Services Department.

5. Some pages in any document may have indistinct print. In all cases the best available copy has been filmed.
DESIGN AND IMPLEMENTATION

OF

A FORM-BASED SOFTWARE ENVIRONMENT

DISSERTATION

Presented in Partial Fulfillment of the Requirements for
the Degree Doctor of Philosophy in the Graduate
School of The Ohio State University

BY

Hong-Chih Jeremy Kuo, B.S., M.S.

* * * * *

The Ohio State University
Department of Computer and Information Science
1983

Reading Committee:                        Approved By

Dr. Jayashree Ramanathan

Dr. M.T. Liu

Dr. Sandra Mamrak

Advisor

Jayashree Ramanathan
Dedicated to my mother,
for trivial but important reasons.
I would like to first express my warmest gratitude toward my advisor, Prof. Ramanathan, for her encouragement and valuable guidance. Without her, the completion of this dissertation would have been impossible. I would also like to express my thanks to the reading committee, Prof. Mike Liu and Prof. Sandra Mamrak, for their valuable comments on the draft. Thanks are also due to Jim Kiper and Mike Kaelbling, who have corrected and corrected all chapters of the draft. Special thanks are due to JoAnn Blum, who, with her outstanding writing skill, has done an excellent job in polishing the English in this dissertation.

The members of the TRIAD project, Dilip Soni, Chin Li, Jim Kiper, Charlie Schubra, Markku Suni and Walter McKnight, deserve credit for their efforts in making the TRIAD project successful. They also deserve the author's gratitude for providing a warm, supported environment. In particular, I appreciate the efforts of Dilip Soni and Chin Li who have been implementing the TRIAD prototype with me. We shall all remember those nights we slept on the desks and the tons of cookies we ate while we were fighting with bugs. I would also like to express my personal pleasure in working with my fellow student, now Dr. Dilip Soni, for sharing all the sweetness and bitterness of the graduate student's life during the last five years.

Finally, I would like to express my gratitude toward my sister, Ru-I, for her understanding and all the delicious meals she cooked for me during the preparation of this dissertation. Certainly, my last thanks go to my two lovely dogs, Hsiao-Hei and Hsiao-Hsiung, for their support and understanding of my staying in office late. Though they have already got back in shape now, they both gained 10 pounds due to lack of exercise.
VITA

EDUCATION

1970-1974 Department of Mathematics, National Tsing-Hua University, Taiwan, B.S. awarded in June 1974.

1976-1977 Department of Mathematics, the Ohio State University, M.S. awarded in Sept. 1977.

1978-1979 Dept. of Computer and Information Science, the Ohio State University, M.S. awarded in March 1979.

1980-present Dept. of Computer and Information Science, the Ohio State University.

RESEARCH INTERESTS

Design and implementation of integrated software environments for program and project development. Issues of interest are: customizability and extensibility of the environments, integration of various tools, project information management, program and document editors, human engineering methodologies, environments for quick prototyping, etc.

Specification of abstract data types and its applications: abstraction techniques, conversions of high level data object, automatic implementation of data abstractions, etc.

Other research interests: Computability and complexity theory, computer-aided problem-solving techniques, executable functional languages, algorithms for transformation between functional and procedural specifications, etc.


# TABLE OF CONTENTS

Dedication ii  
Acknowledgment iii  
Vita iv  
Table of Contents vi  
List of Tables ix  
List of Figures x

1. Introduction 1  
1.1. General Motivations 1  
1.2. Overview of the Limitations of Software Engineering in the Last Two Decades 4  
1.3. Existing Problems in Software Development 6  
1.4. Introduction to TRIAD 13  
1.5. Outline of the Thesis 22  
1.5.1. Primary Contributions 22  
1.5.2. Outline of the Remaining Chapters 29

2. Comparisons between TRIAD and Related Systems 33  
2.1. Organization of Information 36  
2.2. Supporting Programming/Software Methodologies 40  
2.3. Re-Usable Modules 42  
2.4. Human-Engineering 44  
2.4.1. On-Line Assistance 44  
2.4.2. Immediate Visual Feedback 45  
2.4.3. Mistake Recovery 46  
2.4.4. Command Languages 46  
2.4.5. High Level Views of Program or Project Information 48

2.5. Tool Support 52  
2.5.1. Information Editing 52  
2.5.2. Development Tools 55  
2.5.3. Maintenance tools 56  
2.5.4. Integration of Tools 57

2.6. Customizability and Extensibility 59

3. What Are Software Methodologies? 63  
3.1. Characteristics of Software Methodologies 65  
3.2. Methodologies and Languages 70  
3.3. The Syntax and Semantics of Concepts Embedded in Methodologies 72  
3.4. Conceptual Trees and Grammar Forms 78  
3.5. Semantics of Conceptual trees and Attribute Grammars 81  
3.6. Attribute Grammar Forms and Software Methodologies 88
## LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 1</td>
<td>Data for Documentation of Students' Designs for the Wordbag Problem</td>
<td>151</td>
</tr>
<tr>
<td>Table 2</td>
<td>Number of Modules Used in the Program Designs</td>
<td>156</td>
</tr>
</tbody>
</table>
**LIST OF FIGURES**

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>User's View of the TRIAD Environment</td>
<td>14</td>
</tr>
<tr>
<td>2</td>
<td>Blank Forms of a Methodology</td>
<td>15</td>
</tr>
<tr>
<td>3</td>
<td>Outline of the Thesis</td>
<td>32</td>
</tr>
<tr>
<td>4</td>
<td>Overview of the LISP-Prototype</td>
<td>99</td>
</tr>
<tr>
<td>5</td>
<td>Implementation of RTDB</td>
<td>103</td>
</tr>
<tr>
<td>6</td>
<td>Implementation of MDDB</td>
<td>104</td>
</tr>
<tr>
<td>7</td>
<td>Interpretation of the Transaction Processing Pattern for Product Master File</td>
<td>107</td>
</tr>
<tr>
<td>8</td>
<td>Forms</td>
<td>127</td>
</tr>
<tr>
<td>9</td>
<td>Form for a Transaction Processing Problem Requiring Update of a MasterFile Using Transaction Records from a Transaction File</td>
<td>128</td>
</tr>
<tr>
<td>10</td>
<td>Filled in Form</td>
<td>128</td>
</tr>
<tr>
<td>11</td>
<td>Form Mechanisms</td>
<td>132</td>
</tr>
<tr>
<td>12</td>
<td>Blank Forms of a Data Abstraction Based Methodology</td>
<td>134</td>
</tr>
<tr>
<td>13</td>
<td>Wordbag Problem</td>
<td>148</td>
</tr>
<tr>
<td>14</td>
<td>Some Typical Comments of the Problems</td>
<td>164</td>
</tr>
<tr>
<td>15</td>
<td>Three Reasons for Poor Engineer/Program Productivity</td>
<td>175</td>
</tr>
<tr>
<td>16</td>
<td>Cause and Effect between Factors that Affect Productivity</td>
<td>179</td>
</tr>
<tr>
<td>17</td>
<td>Scope for Improvement</td>
<td>180</td>
</tr>
<tr>
<td>18</td>
<td>Overview of a Software Factory</td>
<td>182</td>
</tr>
<tr>
<td>19</td>
<td>Relationship between Engineer's and Programmer's View</td>
<td>185</td>
</tr>
<tr>
<td>20</td>
<td>Hierarchy of Help Information</td>
<td>205</td>
</tr>
<tr>
<td>21</td>
<td>Display of the Result of the Query HELP(DA,6,EXAMPLE)</td>
<td>206</td>
</tr>
<tr>
<td>22</td>
<td>Information Retrieval of Structure-Based Query Functions</td>
<td>212</td>
</tr>
<tr>
<td>23</td>
<td>Overview of the C-Prototype's System Architecture</td>
<td>221</td>
</tr>
<tr>
<td>24</td>
<td>Organization of Command Tables in the Keyboard Monitor</td>
<td>233</td>
</tr>
<tr>
<td>25</td>
<td>Organization of Buffer Structures in BMM</td>
<td>236</td>
</tr>
<tr>
<td>26</td>
<td>Relationships between Data Abstractions Defined in FTDB and Methodology Library</td>
<td>239</td>
</tr>
<tr>
<td>27</td>
<td>Organization of Data Structures in Display Module</td>
<td>246</td>
</tr>
<tr>
<td>28</td>
<td>Blank Forms of the Meta-Methodology: ALPHA</td>
<td>259</td>
</tr>
</tbody>
</table>
1. INTRODUCTION

We shall not cease from exploration
And the end of all our exploring
Will be to arrive where we started
And know the place for the first time.

-- T. S. Eliot, *Four Quartets*

1.1. General Motivations

In the 60's and early 70's, a traditional computer system consisted of tools like language compilers, low-level debuggers, and text editors to support program development. All these tools were under the control of a fairly powerful operating system and were poorly engineered to support the development and maintenance of software. As a result, software costs were excessive. Demands for better programming support came from both project managers, driven by the tremendous rise in software costs, and programmers, caught between the demands of their managers and the limitations of their tools. Two fundamental reasons for this software crisis are discussed below:

1. "Throw-away programs": Programs were developed in accordance with strong personal tastes. The task of programming was more like that of artistic creation than that of a
controlled process producing a well-defined product. Poorly structured programs were difficult for others to modify, and poorly documented programs were not comprehensible to other programmers. Not only were existing programs difficult to maintain, but it was generally infeasible or impossible to create new programs by extending or reusing existing programs. Therefore, most programmers believed that it was easier to write a new program than to modify the existing code developed by others. Thus the problem of "throw-away programs" was actually one of terrible waste as more and more new programs were eventually destined to be thrown away. This caused an inevitable increase in the cost of software.

2. Poorly-engineered tool support: To write a non-trivial program a programmer had to design an algorithm in his mind or on pieces of paper, use a text-editor, or the like, to enter his programming language code, and send it off to a compiler. Then usually ensued the repetitious process of re-entering the editor to correct syntactical errors, and then re-compiling the program. This back and forth cycle between editor and compiler continued and expanded to include a low-level debugger used to locate semantic errors, ending finally when the programmer believed that the
repeated trips through editor, compiler, and debugger had produced a correct program. The programmer's turn-around among different tools and lack of tool support (for program design, modification and maintenance) limited his productivity.

These two fundamental problems have seriously restricted the programmers' productivity. Very briefly, we believe that a rich set of software/programming methodologies will be the solution to the problem of "throw-away programs," and a set of integrated tools which support methodologies will be the solution to the problem of poorly-engineered tool support. One should note that the two fundamental problems are dual to each other. Therefore, the solutions we propose should complement each other. In the last two decades, there have been many efforts aimed at improving software productivity. However, none of them was able to attack these two fundamental problems in a balanced manner. Therefore, these approaches all had limitations and could not provide a satisfactory solution.
1.2. Overview of the Limitations of Software Engineering in the Last Two Decades

In the late 60's there were quite a few debates and discussions about whether programming habits should be changed. A few years later the concepts of "structured programming" [Dahl72] and the "software life-cycle" were introduced and became quite popular in the 70's. In the meantime, some non-computer-aided programming methodologies emerged. Examples are HIPO [Stay76], Structured Design [Your79], and Jackson Methodology [Jack75]. However, since these methodologies were not integrated with any programming tools or languages, they could best serve only as off-line, intuitive guidelines for program development. Given the same application problem, two programmers using the same methodology could still create programs with very different structures. The lack of programming tools that supported methodologies actually placed more burdens on those programmers willing to use programming methodologies. Therefore, the support provided by these intuitive methodologies in software/program development was very limited.

In addition to the above mentioned efforts at changing programming habits, contributions were made by a revolutionary advance in the design of data structures -- the concept of "abstract data types" [Gutt77]. This concept of "abstract data types" has been successfully incorporated into quite a few programming...
languages, like CLU [Lisk77a], Euclid [Lamp77], Gypsy [Good78], Alphard [Wulf76], and Ada [DoD80]. Programs written in these languages, especially those programs using complicated data structures, are much easier to test, modify, and maintain. There are also some other language oriented approaches, like CIP [Baue78], VDM [Bjor78] and PDS [Chea79], which regard the process of program development as a formal mathematical activity. These approaches use a formal program transformation model in their language design, such that programs with fairly good structure and quality can be developed/generated in a top-down manner with the aid of the system. However, these languages are still too "formal" and unrefined to attract programmers in general. We believe that, though such language approaches can reduce the complexity of programming to some extent, most of the problems stemming from the lack of methodologies and tool support will remain.

By the late 70's there were a few programming environments, like Inter-LISP [Sand78], CPS [Deme81], and Mentor [Donz80], which attempted to provide better programming support for programmers. Systems like CPS and Inter-LISP, which can incrementally compile and execute programs to detect errors at earlier stages, were designed to eliminate the programmer's turn-around among different tools and, thus, improve productivity. CPS and Mentor also provide programmers with programming language oriented editors to minimize the number of
errors that are made during program editing. These programming environments basically provided better tool support for programmers and improved the programmers' productivity to a certain degree. However, in these programming environments, methodologies were not supported and there were no tools for program maintenance. Therefore, programs developed in such environments were still likely to be thrown away.

1.3. Existing Problems in Software Development

The above discussion seems to focus only on program development. As a whole, the software crisis is far more than just the problems of poor programming habits and programming tool support. The domain of a software development is no longer limited to programs, programming languages, programming tools, and programmers. For instance, the specification language to be used in the requirement specification and design phases during the software life-cycle is as important as a programming language; the information to be collected during software development should not only be programmer and machine oriented, but also user and management oriented. The content of the software-related information must also be extended from a single program to families of programs, resulting in a tremendous increase in the amount of software-related information.
Though some of the design philosophies of a programming environment can be applied to the design of a software environment, the complexity of the design is drastically increased and the design issues are less comprehensible due to the much wider domain of the software environment. In other words, as far as the development of large scale software projects is concerned, not only are the problems of program development magnified, there are also new problems to be faced. As a matter of fact, these problems echo the two fundamental reasons discussed earlier and they can be categorized into the following seven issues. The problem of "throw-away programs" is echoed in the first three issues, and the problem of "poorly-engineered tool support" is echoed in the remaining four issues.

1. Poor organization of software-related information: Since the development of a software project usually involves more than one person and lasts much longer than that of a program, the software-related information is usually scattered in various media and in different formats. Some of the problems resulting from scattered information are perceptual. For example, it is difficult to support the differing views of the project for its members (managers, 

---

1 Past experience, memos, minutes, comments within programs, user manuals, conversations, project management information, etc.
programmers, users, etc.). Since the relationships among these pieces of information are not structured, no useful views can be generated automatically. Some of the software information records the development of a software project through its entire life-cycle, independent of personal perception. A series of transformations from design specifications to actual implementations of more and more refined programs occurs. If the information in these different phases is not structured properly, as development proceeds, the process of manually maintaining all of the information and keeping track of different versions of designs and programs becomes an annoying and time-consuming task. If the information collected is to be useful for developing software, the structuring of all software-related information according to certain standards (methodologies) is a necessity.

2. Absence of software methodologies: Currently, since no computer-aided methodologies have been used in developing software, much of the software thus created is very complex, unstructured, and difficult to maintain. Good patterns and techniques of software development should be encoded into methodologies in order to systematically guide the user's thought process, as well as to provide standards for
structuring information. Useful experience can also be reflected in methodologies. As the complexity of software grows, experience of developing similar software becomes more precious in creating new software. In order to reduce the software cost, it is necessary to incorporate these experiences into existing methodologies.

3. Useful modules not re-usable: Since most of the software-related information has been either missing or poorly structured, it is extremely difficult to comprehend and modify existing software modules. In many cases, useful modules can not even be identified. Even if they could be identified, it is very difficult to modify them. That is, the problem of "throw-away programs" results in tremendous amounts of effort, time and money wasted in either duplicating similar software or maintaining existing software.

4. Poor human-engineering: In a human-engineered system, high level views and multiple views for different viewers should be provided in order to permit users in a project to access the software information comfortably in different ways. This requires a human-engineered interface to the information and smooth integration of various tools in the system. The
software systems that we currently use in developing software are usually very complex, inflexible, and difficult to use. Rudimentary user-interfaces and poor tool integration limit the software productivity and thus discourage the user. Studies in human factors [Shne80] indicate that human-engineering should be one of the top priorities in systems design.

5. Lack of tool support: Though people have been using whatever tools are available (like text editors, program editors, compilers, version control, etc.) to develop their software, these tools are inadequate to support software development. Given all software-related information, the need for more powerful tools which can create, manipulate, manage, analyze and display all of this information is critical. Currently, such tools are either not developed or are available only at very primitive levels.

6. Non-customizable systems: Users of new systems software often find that they can make use of the functions available in other systems for developing their new applications. Modifying an existing system usually involves the modification of almost all of the old system. The reason for this is that software systems are not properly modularized,
and thus various components are highly interdependent and tightly interwoven through global variables, data structures, and function invocations. For example, a more specific or new application may require only altering the parameter type of a function, but this is usually not possible unless the designer has envisioned such customization. Due to the difficulty of modifying existing programs, users who need more customized features often end up writing their own programs. Especially when the problem domain becomes more specific, customizability also becomes more desirable.

7. Non-extensible systems: Customizability of a system concerns the ease of modifying a system to suit a more specific or similar application, but the major functionality of the system remains. However, we often find that we need to extend the functionality of an existing software. Only a very few existing software systems facilitate such extension by allowing their users to define new functions or commands. In most software systems one might be able to accomplish some extensions, although less efficiently, by building a level on top of the current commands and from there invoking commands and functions in the existing software. If there are changes in the data structures, we again face the
modification problem. Therefore, a more general-purpose software system usually duplicates the entire lower level software simply because the implementation of the original software is not extensible. For example, the software for a numerical application may be originally implemented for only real numbers. If one wishes to extend the same application for complex numbers, extra support for handling complex numbers should be designed, but one should not have to re-implement the entire software. It is, in general, impossible for a software designer or user to anticipate all of the possible future extensions for some software. Nevertheless, many software systems simply trade their extensibility for some unimpressive savings in time or space. This not only restricts their application domains, but also forces unnecessary duplication in the development of similar software systems.

One should note that these problems are all inter-related. No satisfactory solution is possible if we attack each problem individually without considering the others. For example, better tool support can be facilitated only if the information in the databases is well-structured; and structuring the information is possible only if good methodologies are used. Furthermore, methodologies can be effectively enforced only if human-engineered
tool support is provided through an appropriate interface. Therefore, we feel that a software environment should adopt a suitable model to structure all software-related information, support the entire life-cycle of software development, provide tools for the easy manipulation of information, and interface with users in a "friendly" manner. Methodologies should also be supported in order to guide the programmers' thought processes on the one hand, and to standardize the structure and documentation on the other. The environment itself should be easily customizable and extensible. The TRIAD software environment, described in the following section, has been designed to meet all these requirements.

1.4. Introduction to TRIAD

The work presented in this thesis is based on work done on the design and implementation of TRIAD, a TRee-based Information Analyzer and Developer. TRIAD is an adaptable, integrated, tool-box environment which supports the development and manipulation of all software-related information during the entire life-cycle of a project. In other words, in contrast to the existing programming environments which create and manipulate only programs, TRIAD can create, structure, and manipulate all software-related information. Components visible to the users in the environment are organized as shown in Figure 1.
When a user first logs in the TRIAD environment, he/she is asked to select a methodology from the Methodology Library. This library consists of a set of pre-defined methodology descriptions; each methodology description is in turn a set of blank forms. An example of a set of blank forms for the description of Project Development Methodology is given in Figure 2.

A methodology describes what information should be collected
<table>
<thead>
<tr>
<th>PDM-Form-3</th>
<th>Project Design</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Architectural specification:</td>
</tr>
<tr>
<td></td>
<td>(7) Functional Specification Form-use-#:</td>
</tr>
<tr>
<td></td>
<td>Reusable Modules</td>
</tr>
<tr>
<td></td>
<td>Module name: (More?)</td>
</tr>
<tr>
<td></td>
<td>Design Decisions: (More?)</td>
</tr>
<tr>
<td></td>
<td>Version Control</td>
</tr>
<tr>
<td></td>
<td>Version Number:</td>
</tr>
<tr>
<td></td>
<td>Review</td>
</tr>
<tr>
<td></td>
<td>Reviewer: Date: Comments:</td>
</tr>
<tr>
<td></td>
<td>Previous Version: Form-use-#:</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PDM-Form-1</th>
<th>Project Form-use-#:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Project Name:</td>
</tr>
<tr>
<td></td>
<td>Project Description</td>
</tr>
<tr>
<td></td>
<td>Project Goals:</td>
</tr>
<tr>
<td></td>
<td>Project Manager:</td>
</tr>
<tr>
<td></td>
<td>Project Participants: (More?)</td>
</tr>
<tr>
<td></td>
<td>Project Proposal</td>
</tr>
<tr>
<td></td>
<td>Proposer:</td>
</tr>
<tr>
<td></td>
<td>Abstract:</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PDM-Form-10</th>
<th>Data Abstraction Module Form-use-#:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Data abstraction name: (this will be copied)</td>
</tr>
<tr>
<td></td>
<td>Purpose of this module:</td>
</tr>
<tr>
<td></td>
<td>Lower level data abstractions</td>
</tr>
<tr>
<td></td>
<td>(10,13) Data abstraction module name: (More?)</td>
</tr>
<tr>
<td></td>
<td>Canonical form:</td>
</tr>
<tr>
<td></td>
<td>(16) Operation name: Form-use-#: (More?)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PDM-Form-7</th>
<th>Functional Specification Form-use-#:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Input specification:</td>
</tr>
<tr>
<td></td>
<td>Output specification:</td>
</tr>
<tr>
<td></td>
<td>(8) Control algorithm specification: Form-use-#: (More?)</td>
</tr>
<tr>
<td></td>
<td>(this control algorithm can use only the visible lower level functions or data abstractions listed below.)</td>
</tr>
<tr>
<td></td>
<td>Visible lower level modules</td>
</tr>
<tr>
<td></td>
<td>(9,12) Function module name: Form-use-#: (More?)</td>
</tr>
<tr>
<td></td>
<td>Visible lower level data abstractions</td>
</tr>
<tr>
<td></td>
<td>(10,13) Data abstraction name: Form-use-#: (More?)</td>
</tr>
</tbody>
</table>

Figure 2: Blank Forms of a Methodology
and how it should be organized using the underlying attribute grammar form model. A methodology is presented to the users as a set of blank forms. Each blank form consists of a number of organizers (Figure 2) which prompt the users about what and how to fill in the blank form. Help information is provided with each organizer, and more detailed help information is available in the corresponding help form upon request. The relationships among different blank forms are also specified in the methodology description. After a blank form has been filled in by a user, he/she can select another blank form, suggested by the methodology, to refine an organizer. Through a successive process of refinement, the user can build up form trees which structure the information according to the methodology. The developed form trees can then be stored in the user's Form Tree Data Base (FTDB). During the development of a project, several form trees are usually created to document different versions, different components, or different phases in the life-cycle of a project.

The User Interface interprets the incoming user commands, and then invokes appropriate routines defined in the Tool-Box. It also insures that all the requested information is properly displayed. Tools such as the Software Information Editor, Tuner, Analyzer, Display Module, and compilers are contained in the tool-box. However, the users do not have to pass their form trees from tool to
tool. All the tools in the tool-box are integrated through the uniform form tree representation. A rich set of commands will automatically bring in the appropriate tools to manipulate the information. Therefore, users no longer have to worry about the interface between tools.

The Software Information Editor can edit, structure, and manipulate information in a form or in form trees. It also supports powerful commands, which are similar to the query language in a relational data base, to retrieve methodology-oriented chunks of information. For example, a project member can look at only the information in which he is interested. A project manager can look at the management-oriented information, while the programmer can look at the program-related information. The Analyzer, which is being designed, can further aid the users in analyzing software-related information. The Tuner is being implemented to create and modify methodology descriptions in order to customize the environment.

The following seven important characteristics of TRIAD correspond to the seven issues of software development discussed earlier. These characteristics illustrate how the general issues are addressed by the TRIAD approach:

1. TRIAD structures all software-related information: All software-related information, collected using a methodology
description, is structured as form trees in the Form Tree Data Base. Carefully designed methodology descriptions in the Methodology Library provide structure for collecting all software-related information of a project during its entire life-cycle. This information structure also facilitates better tool support (to be discussed later).

2. TRIAD supports software methodologies: "Good" patterns of software development have underlying concepts which, when encoded into a set of blank forms, systematize the development process. TRIAD supports general as well as problem domain-specific methodologies in order to guide the user's thought process during the software development. Useful experience of developing similar software can also be reflected in methodologies by tuning an existing methodology description. Therefore, software developed using such methodologies has good patterns and more standardized structure.

3. TRIAD facilitates re-usable modules: A blank form provides a standard template by which to document software-related information. That is, a blank form can simply be completed by a project member, thus providing standardized documentation for the use of the related module. Many
modules with useful functionality can be developed using a methodology and can be stored as form trees in FTDB. These form trees can be slightly enhanced by an expert and kept in the library as re-usable modules. Since all form trees developed using the same methodology have fairly standardized structure, they are ready to be used to create new modules. It is also possible that when the same pattern is developed repeatedly, the expert can pre-fill most of the entries in that form tree and make it a re-usable module in the library. It is expected that, when the problem domain becomes more specific, the design and programming effort can be greatly reduced with the help of such re-usable modules.

4. TRIAD uses a human-engineered form-based approach: Issues related to the organization and manipulation of software-related information according to methodologies were first studied using a prototype editor implemented in LISP. Experience in using the prototype suggested that alternate ways to human-engineer had to be determined. Forms were then proposed as an approach to this problem and were used in a classroom situation to enforce methodologies. The suitability of this approach to organizing information was also studied in an industrial setting. The success of the form approach led to a complete re-design and implementation
of the system. These forms are carefully designed, considering various human factors, to serve as the front end of the environment in order to effectively enforce methodologies.

5. TRIAD provides improved tool support: Methodology-oriented tools no longer force the user to develop his/her software in terms of low-level programming language constructs. Tools in the environment can manipulate very high level concepts, and can selectively retrieve meaningful, methodology-oriented chunks of information, thus making all software-related information readily accessible to the user. Tools in TRIAD are also smoothly integrated in such a way that the user no longer has to worry about the interface between them. Different modes for different tools are eliminated and eventually the user's turn-around time between tools is largely reduced.

6. TRIAD is a customizable environment: In the Methodology Library, there are some manually constructed methodologies which are, in turn, used by the Tuner to construct other methodologies. Using the Tuner, one can create a new methodology or systematically alter a methodology description to reflect the project oriented experience.
Tools like TRIAD/SIE, Analyzer, and Display Module are generic tools, that is, they are methodology independent. Since the methodology description is also available to these tools through the use of appropriate parameters at the command level, these generic tools can be customized by any methodology. Thus, when a "tuned" methodology is used, these tools can also be adapted to the user's needs. Therefore, a project manager can customize the environment by using a "tuned" methodology.

7. TRIAD's user interface is extensible: The User Interface is designed and implemented in a similar manner to that of Emacs [Fins80] [Stal79]. An expert user can re-bind an existing procedure or command to a new key, or define new commands and add them to the command set.
1.5. Outline of the Thesis

1.5.1. Primary Contributions

The work described in this thesis was done in the context of the TRIAD project described above. The primary contributions of this thesis are identified and summarized below:

1. The development of a method for structuring all software-related information:

   - Problem: One of the most immediate problems in existing programming and software environments is the need of a model to structure all software-related information through the entire life-cycle of a project development. A substantial amount of software-related information of a project used to be generated and scattered in various media and maintained by different project members manually. As a result, this information was usually incomplete, inconsistent, and unstructured. Therefore, it is practically inaccessible. Since a large portion of the software-related information is human-oriented, the context-free grammar model, which is normally used to structure programs, is no
longer adequate to structure this information.

- **Contribution:** This thesis provides an approach to structure all software-related information according to the underlying attribute grammar form model. This model was first used to structure only the program-related information during a program development [Kuo81]. Later, it was successfully extended to structure all software-related information during a project development. The feasibility of using the attribute grammar form model to structure all software-related information was established by first designing a methodology based on this model and then having students use this methodology to develop a medium-size project [Kuo82]. This study not only contributes to the extension of TRIAD from a programming environment to a software environment, but also facilitates the improved tool support for software development and maintenance.

2. The design and implementation of a construction and maintenance tool for software development:

- **Problem:** As discussed in the earlier sections, one
of the fundamental problems in software engineering is the lack of tool support for both software design and maintenance. Almost all of the existing tools (such as editors and compilers) can only edit or manipulate the collected information as text for a text file or as a syntax tree for a program. In other words, they are unable to construct, edit, or manipulate the information as software-related information. Therefore, the power of these traditional tools is very limited.

Contribution: This thesis describes the design and implementation of a powerful tool, called the Software Information Editor (TRIAD/SIE), which can construct, edit, and manipulate the information in the software information database, and thus, makes all software-related information readily accessible to the user. The significant contributions of this software tool can be demonstrated by its versatile functionalities and its flexible design: it can behave like a smart text editor, a tree editor, a form manipulation system, and an information manager; it is also a generic tool in the environment and can be easily extended. It is the
integration of the current state of technology in structure editors, database management systems, and software engineering principles that makes this tool a powerful construction and maintenance tool in the TRIAD software environment.

3. The design and implementation of software methodologies:

- Problem: Though there have been many methodologies specifically designed for program or project development, their uses are either very restricted or biased due to the following three drawbacks: Firstly, methodologies are usually imprecisely specified and thus can not be enforced systematically. Secondly, most methodologies are designed to be used to design or test a project only in a particular phase of the entire life-cycle. Finally, methodologies are usually manually enforced and all suffer from the lack of system support (tools, databases, etc.).

- Contribution: This thesis first describes the required characteristics of a software methodology, and then provides an approach to specifying a software methodology using the underlying attribute
grammar form model. The advantages of this approach are that it can structure the development history of a project development, capture the semantics of high level concepts, and facilitate reusable modules using patterns. Therefore, this approach allows a software methodology to be specified in a much more precise and forceful manner. Furthermore, this thesis describes the rationales of the design of two methodologies using this approach. This study opens a new direction of computer-supported methodologies to support software development through its entire life-cycle, and will have definite impact on the design of contemporary and future programming and software environments.

4. The design and implementation of a software information database:

- **Problem:** The databases in traditional programming systems usually contain only the text files (documentation, source code, etc.) and object code for programs. Such a primitive database structure seriously restricts the manipulation and management of software-related information. Newer systems adopt syntax trees and files as basic data units in their
databases, but they are still inadequate to support the manipulation and management of all software-related information.

- Contribution: This thesis describes the design and implementation of a sophisticated software information database which serves as the kernel of the entire system organization. From the user's point of view, high level data structures such as form trees, forms, organizers, methodologies, blank forms, patterns, and help information are all defined in the database. From the system's point of view, the database also contains the refinement trees, concept trees, concept grammars, and system library. This software information database is designed based on the current practices of data base management systems, and implemented as a hierarchical database using the data abstraction techniques. Two of the most noticeable contributions of this database are: it facilitates better tool support (TRIAD/SIE and Tuner) for software design and maintenance, and it facilitates the integration of tools in the TRIAD environment via the uniform form-based representation.
5. The human-engineering of the software environment using a form-based approach:

- Contribution: Issues relating to the use of the structured information, methodologies, and tools were first studied using a prototype editor implemented in LISP [Kuo81]. Experience with this prototype suggested that it was necessary to devise an alternate way in which to human-engineer the software environment. Forms were then used as an approach for the enforcement of software methodologies. The suitability of these forms for the organization of information was then studied in both an industrial and a classroom setting [Kuo82]. The forms organized information related to project management, requirements specifications, engineers' design specifications, and programs developed using a data abstraction based methodology. The contribution of this form-based approach was established by its usefulness for enforcing methodologies, standardizing project structures and documentations, as well as, human-engineering the user-interface. The success of this form-based approach led to a complete re-design of the TRIAD
environment.

6. The design and implementation of a working prototype of TRIAD:

- Contribution: The experience obtained from the LISP-prototype has been a very valuable lesson in the re-design of the current prototype which is implemented in C under UNIX. The implementation of this C-prototype has been divided into three sub-projects. In addition to the intensive participation in the overall design and the final integration of these sub-projects, it is the author's responsibility alone for designing and implementing the software information database and the Software Information Editor [Kuo83a] [Kuo83b].

1.5.2. Outline of the Remaining Chapters

- Chapter Two: this chapter discusses the related work so as to emphasize the contributions of the work described in this thesis. More than a dozen of related systems are discussed based on the criteria mentioned in the previous section.

- Chapter Three: this chapter will first discuss the desired characteristics of software methodologies and then discuss
the underlying model which is used to encode methodologies and to guide the programmers' thought processes. It concludes the chapter with a working definition of the computer-supported software methodologies in terms of the attribute grammar form model.

- Chapter Four: This chapter describes a prototyping approach to system design using a prototype implemented in LISP with the FRL package. The implementation and issues identified by this prototype are discussed.

- Chapter Five: This chapter first describes the form-based approach to enforcing software methodologies and human-engineering the user-interface. Two methodologies are also presented in this chapter as two case studies. The Data Abstraction Based Methodology is the first computer-supported methodology to enforce the algebraic data abstraction techniques and the layer-approach for program modularization. Another larger scale methodology, called the Project Development Methodology, was designed based on the valuable experience obtained during the consulting for a computer company. These case studies emphasize the usefulness of the form-based approach to human-engineering the environment, the rationales of the design of
methodologies, and the experience obtained by manually enforcing methodologies.

- Chapter Six: this chapter is devoted to describing the powerful Software Information Editor tool which can construct, edit, and manipulate all software-related information in the software information database. The power of this tool demonstrates the achievements of the work described in previous chapters.

- Chapter Seven: this chapter discusses the human-engineered design and implementation of the C-prototype of TRIAD.

- Conclusions: Conclusions and future research directions of software environments in the context of software engineering are discussed in this chapter. Issues related to other research areas are also discussed.

A "human-engineered" diagram which sketches the outline of this thesis is given below:
Figure 3: Outline of the Thesis
2. COMPARISONS BETWEEN TRIAD AND RELATED SYSTEMS

Surely the good man is the bad man's teacher; and the bad man is the good man's business. If the one does not respect his teacher, or the other doesn't love his business, his error is very great.

— Lao Tsu

What, is the jay more precious than the lark, Because his feathers are more beautiful? Or is the adder better than the eel, Because his painted skin contents the eye?

— Shakespeare, Taming of the Shrew

Since the design phase of the TRIAD project started in 1978, research in programming environments and programming tools has increased dramatically. Within the three to four years prior to 1980, more than two dozen such systems or tools were designed or implemented. These systems or tools range from program/structure editors and office automation systems, to programming environments and software environments. Though TRIAD supports many features found in these systems, it has been uniquely designed. TRIAD is a software environment as opposed to a programming environment. It is adaptable, supports methodologies, and structures and manipulates all software-related information during the entire life-cycle of a project.
The comparisons in this chapter serve only to emphasize the unique features and contributions of TRIAD; they by no means attempt to "evaluate" the other systems. A list of related systems to be compared with the TRIAD environment is given below:

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Z</td>
<td>-- Yale University</td>
<td>[Wood81]</td>
<td></td>
</tr>
<tr>
<td>2. CPS</td>
<td>-- Cornell University</td>
<td>[Teit81]</td>
<td></td>
</tr>
<tr>
<td>3. sds</td>
<td>-- University of Arizona</td>
<td>[Fras81]</td>
<td></td>
</tr>
<tr>
<td>4. KD3</td>
<td>-- Linkoping University, Sweden</td>
<td>[Stro81]</td>
<td></td>
</tr>
<tr>
<td>5. MENTOR</td>
<td>-- INRIA, France</td>
<td>[Donz80]</td>
<td></td>
</tr>
<tr>
<td>6. Etude</td>
<td>-- MIT</td>
<td>[Good81]</td>
<td></td>
</tr>
<tr>
<td>7. GANDALF</td>
<td>-- CMU</td>
<td>[Habe82]</td>
<td></td>
</tr>
<tr>
<td>8. Smalltalk</td>
<td>-- Xerox</td>
<td>[Gold82]</td>
<td></td>
</tr>
<tr>
<td>9. ALOE</td>
<td>-- CMU</td>
<td>[Medi82]</td>
<td></td>
</tr>
<tr>
<td>10. PDS</td>
<td>-- Harvard</td>
<td>[Chea79]</td>
<td></td>
</tr>
<tr>
<td>11. CIP</td>
<td>-- Muenchen</td>
<td>[Baue78]</td>
<td></td>
</tr>
<tr>
<td>12. Programmer's Apprentice</td>
<td>-- MIT</td>
<td>[Wate82]</td>
<td></td>
</tr>
</tbody>
</table>

These systems cover a range of environments. The list includes program editors (Z, MENTOR), structure editors (ED3, sds, Etude), program development systems (CPS), program transformation systems (PDS, CIP), program-editor generators (ALOE), knowledge-based programming systems (Programmer's Apprentice), personal computers (Smalltalk), and project development environments (Gandalf). Though
their emphases are slightly different, many features provided by these systems or tools are very useful for software development. There is also a great deal of commonality among these systems. For instance, they are all interactive systems with screen display; they all collect and organize certain kinds of information and provide tools for the user to manipulate this information; and they all aim at improving the programmer's productivity. However, the approaches taken by these systems to the achievement of this goal are quite different. It will be difficult to evaluate these different approaches because each has different priorities in various design issues. The intention of this section is to "learn" from these existing systems and to "integrate" any useful techniques with our ideas, so that these can be applied in the extended domain — the domain of software development.

The existing issues in software engineering environments which were discussed in the previous chapter, will be used for the comparisons. It is, therefore, useful to refer to those issues while reading the following sections. The terms "program development system" and "programming environment" are used interchangeably to mean a system with integrated tools and friendly user interface for program development.
2.1. Organization of Information

Information organization to an information system is like the root to a tree. Only a healthy and strong root can guarantee that the rest of the tree above the ground surface will flourish. So must information organization provide a sound and solid foundation for an information system. First of all, the information domain of these related systems, with the exception of Gandalf, is restricted to program-related information. However, since the collection of programs is the core of all software-related information, the organization of program-related information is also important. There are several different approaches to representing programs among these systems:

1. **Abstract syntax trees** — A program is stored as an abstract syntax tree in MENTOR, ALOE and Gandalf. An abstract syntax tree is different from a concrete syntax tree in the sense that the abstract syntax tree stores only the syntactic information necessary for semantic analysis in the compiler. Therefore, the same abstract syntax tree may actually correspond to several concrete syntax trees which use different lexical units to denote the same language constructs. Since the abstract syntax is not the same as the concrete syntax which a programmer sees, a parser and an unparsener are required to perform the conversions between the
abstract and concrete syntax. Since an abstract syntax tree hides inessential syntactic details of the target language, it is a more efficient representation for programs in MENTOR. A language Mentol is designed to manipulate these abstract syntax trees. Though a user still sees the pretty-printed program text on the screen, she/he is actually manipulating the abstract syntax trees. For example, the parentheses are not part of the abstract syntax. The unparsers takes care of this and the user can still see the parentheses displayed properly.

2. Concrete syntax trees: CPS uses templates to encode most of the higher level language constructs in PL/C and provides the user with commands to use these templates and to fill in the "holes" in them. Since the syntax trees are incrementally compiled, programs are maintained as concrete syntax trees in CPS.

3. Text: Z is the only program editor which insists that a program should be kept as text. This is an easier implementation if a tool is only to be used for program editing and nothing else. In a programming or software environment, all tools should be integrated to optimize the overall capability. Such representation has a very limited
structure and, thus, no incremental compilation or high-level view of the information is possible. Therefore, text representation for programs or project information is not suitable in programming or software environments.

Representing the programs as abstract syntax trees results in more efficient memory utilization with the overhead of an unparser to display programs. Representing the programs in terms of concrete syntax trees does not require the unparser and is more efficient in terms of time, but storing the entire concrete syntax trees is expensive in terms of space. The abstract syntax is, probably, a more desirable representation because it can support multiple languages. That is, it is more flexible than concrete syntax.

In Gandalf, a more sophisticated approach is used to handle the project-related information. High level information like functional specification of modules, system components, and system descriptions is functionally organized into a structure called "project". A "project" in Gandalf is a specific structure with a specific set of modification rules. This notion of "project" in Gandalf is comparable to that of "data abstractions" in programming languages. The information one can manipulate in this system is no longer limited to programming language code or comments.

In TRIAD, all software-related information is structured as
form trees according to a methodology description. The information domain ranges from the requirements specification phase to the coding phase of the entire life-cycle of a project development. In contrast with the generic project structure in Gandalf, the information structure in TRIAD is methodology-oriented and thus contains more semantics than in Gandalf. This point and other advantages of such a methodology-oriented information structure will become clear as our discussion proceeds.

A program in TRIAD is represented as a form tree. Such a form tree structures not only the syntax tree needed for compilation, but also the entire development history and documentation which are useful for program maintenance. In TRIAD, the users are dealing with the form-trees (concrete syntax), but the system maintains the refinement trees (abstract syntax). The unparsing schemes (or display specifications) are stored with the blank forms in a methodology description. One can easily change the display specification of a blank form and cause all the instances of this blank form to be displayed accordingly.
2.2. Supporting Programming/Software Methodologies

Gandalf provides structure and tools for developing projects. CPS, as well as other systems, provides structures and tools for developing programs. None of these can suggest patterns to guide the user's thought processes and assist the user in "how" to develop a program or a project.

In program transformation systems, such as CIP and PDS, a program is first completely specified at the abstract level and then a sequence of transformation rules is selected to transform this specification into a semantically equivalent implementation. These semi-automatic transformation rules in such transformational systems are similar to TRIAD's methodologies. However, there is no support for creating or modifying the abstract specifications (which is usually the most difficult part for large programs). Therefore, these systems are only suitable for developing small programs in which the specifications are easy to obtain.

In Programmer's Apprentice, the "plan library" consists of a collection of plans for common algorithmic fragments. A plan can be selected by its name, such as "sorting" or "summation", and can be slightly modified by the user to suit his particular purpose. These plans are basically abstract representations of some existing algorithms and are independent of any programming languages. As soon
as a plan is chosen, the user can use the tool "coder" to generate program code for this plan automatically. To do this, this system has to understand the data structures, the specifications, and the relationships between different plans. Unfortunately, none of these has yet been achieved. So far, the Apprentice can only deal with plans for developing very small programs.

The goal of the TRIAD project is more modest and flexible than that of Apprentice. Good patterns are encoded as abstract concepts in a methodology. The relationships between these concepts are also kept in the methodology description in order to guide the user's thought process. Using such methodologies, a program or project can be developed in a systematic manner. In addition to guiding the user's thought process, the use of methodologies has the following advantages:

- Methodologies can reflect useful experience by presenting concepts, suggestions, check-lists, and specific procedures. Methodologies can also encode project-oriented concepts and thus can guide the user's thought process more effectively in a specific project domain.

- Methodologies can be used to standardize the documentation, including its notation, content, and placement.

- Methodologies provide good patterns and specific refinement
rules such that a program or project developed using a methodology has fewer variations and is thus easier to maintain.

TRIAD differs from typical AI systems in the degree of user control. TRIAD is not intended to generate code or transform specifications into its implementation code in an automatic manner. A user still has to interpret and refine the abstract concepts. The whole development process is always monitored by using the methodology description but only in a semi-automatic manner. On the other hand, this approach is more ambitious than that in Apprentice in the sense that this tries to provide methodology support in a much wider problem domain. TRIAD can also encode useful concepts for software development and not simply concepts for program fragments. The details of how a software methodology is encoded using the underlying attribute grammar form model, and how it is enforced using the form-based approach, will be discussed in later chapters.

2.3. Re-Usable Modules

Among these systems, the Programmer's Apprentice is the only system which facilitates re-usable modules. As soon as a useful algorithm is recognized, an expert user can create a plan specification of this algorithm and add it to the "plan library". When this plan is used to develop a problem, the system can generate
a program for the user based on the corresponding implementation of that algorithm. The programmer still has to examine the program to see if it is correct, and this can only be done at the level of small routines such as sorting or searching.

Though facilitating re-usable modules is not an explicit intention of the expert systems such as Apprentice, the approach mentioned above is similar to that used in TRIAD. Since "experience" and "re-usable modules" are two of the most valuable things in the treasure house of software or program development, it is TRIAD's explicit goal to support experience and re-usable modules. Similar to a plan and its implementation, an experience is encoded into abstract concepts in a methodology description and its implementation becomes a re-usable module whenever the same abstract concept is used to develop a problem. More specifically, a re-usable module in TRIAD is a developed form tree or subtree based on a methodology description. This module can be a design specification and not necessarily a program fragment. Such re-usable form trees are also stored in the Methodology Library and are ready for use whenever the same abstract concept is used.
2.4. Human-Engineering

There are many aspects to human-engineering a computer system. Some of the important aspects are: ease of use, reliability, efficiency, immediate feedback, mistake recovery, data security, comfortable command language, help information, flexibility, and high level views. The fundamental guideline for human-engineering a system is to make the user feel that she/he has complete control of the system and not the other way around.

Not all the aspects of human-engineering are discussed here. Only those aspects which are addressed by the TRIAD design, in a unique way, will be discussed.

2.4.1. On-Line Assistance

An effective interactive system should be able to give explanations of both a **static** (what the system can or cannot do in a general sense) and a **dynamic** (what the system is doing and why it is doing that and the outcome of the past event) nature to its user [Good81]. Most systems provide help information to a certain degree (at least in regard to the static nature of the system). Both Smalltalk and Etude have a hierarchical "menu system" such that a user can query in depth. Even though the help information provided by these systems is still in an ad hoc manner, such on-line
assistance does greatly reduce the user's frustration and timidity.

TRIAD also provides hierarchical help information, in varying degree, about the data structures, forms, methodologies, and the system status. In addition to help information, a system should provide useful error messages. This is a weak area in the current design of the TRIAD prototype, as it is in most prototype systems.

2.4.2. Immediate Visual Feedback

The feedback for any command issued by the user should be immediate. In other words, the response time should be under the user's tolerance. Z shows the line numbers during program execution and Scribe displays the page numbers while it is scribing a file, so that the user does not sit idle as he/she is temporarily "disconnected" from the user-environment system. CPS and Gandalf incrementally compile and execute the program to whatever degree possible, so that the user is always informed by the system about the outcome of the task she/he is performing. These are all excellent design decisions. TRIAD will be designed to inform the user about the errors during different phases of the life cycle. This will be facilitated by an incremental analysis of the structured software-related information.
2.4.3. Mistake Recovery

A user should be able to recover from critical commands which would result in a drastic modification of the existing data. "Abort" a command before the completion of its execution and "undo" a command after the completion of its execution have been recognized as necessary support for mistake recovery. Most systems support "abort" because it is easy to implement. However, the "undo" operation is much more difficult. Of the systems studied, only Z and Etude have support for "undo". Most systems are very weak in this feature.

The current TRIAD prototype supports "abort" for all commands and "undo" for some crucial commands like deletions or overwriting of data structures. The "undo" support in TRIAD should also be enhanced in the future.

2.4.4. Command Languages

The command language of a system should be carefully designed to ensure that the system is easy to use and flexible. Most of the related systems here use a simple command language, that is, a language that consists of only a set of simple commands. A single editing or analysis task usually requires a series of commands. Since these commands are defined at a fairly low level, the users
are forced to be aware of the exact structure of the information with which they are dealing (like syntax trees). That is, these commands are constrained by the data structures (which are all hierarchical), and the user's vocabulary is restricted to using only these structure-oriented commands.

**Emacs** allows users to define their own commands using macro definitions. This facilitates high level commands and also reduces the number of keystrokes required. In **MENTOR**, a language called Mentol is provided to support procedures. This makes Mentol more like a programming language instead of a command language in the usual sense. But, a complicated programming language-like command language may not be the most desirable solution. The design and test of these procedures would place burden on users and cannot be accepted by general users who are not programmers.

**Etude**, which is a component of an office automation system, uses the verb-modifier-object format in its basic command structure. The popularity of this approach is due to the fact that the user can perform complex functions by combining relatively small sets of primitive objects into commands rather than by having to remember and choose from a much larger set of separate commands.

The **TRIAD** project envisions a three-stage approach. The design of command language in the C-prototype of **TRIAD** is implemented
similar to that of Emacs. The second stage will be something like the verb-modifier-object structure in Etude. The Emacs approach is not sufficient since the data structures in TRIAD are far more complex than the text buffers in Emacs and, thus, the number of commands needed is much larger. The verb-modifier-object approach reduces the number of commands and relieves users from the inconvenience of having to memorize all of them. The final stage will be a powerful query language for entering, accessing and manipulating the information. A query language based on a relational data base is superior to a simple command language because it does not force the user to think of the information in terms of the structure and can access and modify the information more effectively. It is also more desirable than a procedural language because it does not require programming and, thus, is easier to use.

2.4.5. High Level Views of Program or Project Information

It is commonly agreed that the syntactic structure provided in a programming language is not the best way to understand or design a program, especially a large scale program. Formal specifications and documentation standards for programs have been suggested to improve the understandability and maintainability of programs. In the meantime, the concept of structured programming has also been introduced to improve the maintainability of programs. How can one
enjoy reading "Alice's Adventures in Wonderland" if she/he only understands English grammar? Or, how can one find her/his way in Tokyo or New York City without a map (assuming that nobody is willing to help you)? Syntax-directed editors provide a language-oriented view of programs which is better than the text view of programs. But, higher and higher views of programs and software-related information are needed in order to grasp the most important, intrinsic concepts behind the "machine-motivated" language syntax. When the amount of information collected gets larger, such higher level views (or abstract views) become even more important.

Among the related systems, there are four approaches taken to facilitate high level views:

1. Tree structures are used to represent programs in systems like MENTOR, sds, ED3, or even Z (which uses indentation to simulate tree structure). In these systems, programmers can "zoom" in and out of the tree structure to suppress the lower level details of a program. Thus, the programmer can have a better view of the overall structure instead of being trapped inside the tedious, inessential details. However,

\[A \text{ comment from Dr. Ramanathan: Streets don't have names in Tokyo; in New York, the street names don't correspond to the names on the map!}\]
the high level view provided by these systems is still bound at the language level. That is, though the programmer now can look at the higher level of the syntax trees, she/he can never go beyond the syntax trees.

2. CPS moves one step forward by including the comments of program fragments in the tree structure. In CPS, programmers can suppress a whole chunk of program code by simply displaying the comment (which documents that chunk of code) and replacing that piece of code by ellipses. Comments, which can be pseudo code, specifications, or design decisions, are more meaningful than language keywords like the "begin...end" pair. Therefore, the views of programs supported by CPS are higher than those supported by the above systems. However, storing the comments with the syntax tree can only provide views based on the program structure, and a single level of abstract views (comments) is also inadequate.

3. Since CPS is designed for users to develop small programs, the approach it adopts may be well-suited to its environment. In Gandalf, a specific "project" structure and rules of manipulating the information in a "project" are defined. Therefore, higher level views (project
specifications) and different views (manager, programmer, and general user) of a project are possible.

4. In *Programmer's Apprentice*, a knowledge-based plan library is included in the system to provide abstract views of programming concepts. A "plan" is an abstract concept of a certain program fragment which is independent of any programming languages. A "drawer" is also provided to draw a plan as a flow diagram. Such high level views are much more useful and attractive than those mentioned above. However, the current focus of such an AI approach is still on very small programs.

The major difficulty in software development is in organizing the large amount of information and supporting *programming-in-the-large*. Therefore, the issue of supporting users to manipulate information at a higher level than the programming language level should be treated seriously. Manually maintaining a large amount of project information is either impossible or time-consuming and error-prone. In TRIAD, explicitly organizing all software-related information and providing abstract views of the information are main tasks instead of just co-products.

TRIAD organizes the project information into form trees and provides tools like Software Information Editor and Analyzer to
manipulate and retrieve the information. Since the form trees are developed based on the pre-defined methodologies (like Gandalf's project structure), TRIAD/SIE can retrieve meaningful chunks of information to provide higher level views and different views.

2.5. Tool Support

2.5.1. Information Editing

Computer-aided text editors have had revolutionary impact on all kinds of editing tasks, and thus push the secretaries and programmers to advance one step forward from the age of traditional typewriters and keypunchers. Once Emacs-like systems found acceptance, researchers recognized the need for more powerful editing tools in areas including office automation, programming, graphics, etc. EMILY and MENTOR are two of the pioneers that provide syntax-directed editing functions on structured data instead of editing strings of characters or lines of characters. CPS, MENTOR, and Gandalf are successful examples of such syntax-directed editors which are designed specifically for editing programs in a programming environment.

The originators of the Z editor have a strong objection to representing the programs in any structure other than ordinary text. The debate is rooted in the fact that structure editors, though
having definite advantages over traditional text editors, sacrifice a great deal of convenience in editing simple text which can be done in text-based editors very easily.

Here is a closer look at the pros and cons of the syntax-directed editors. The syntax-directed editors can

1. provide automatic pretty printing based on the syntactic structure of the language, and automatically parse the program text into syntax trees,

2. allow the programmer to select or manipulate programs using complete syntactic units (that is, language constructs) with only one or two keystrokes,

3. zoom out and suppress the lower level details of a program,

4. quickly detect errors with the cursor positioned at the locations of errors in a program, and thus, guarantee the syntactic correctness and some semantic correctness of a program at all times, and

5. allow the programmers to edit his programs in terms of language constructs rather than strings mixed with characters and punctuation.

But, the syntax-directed editors also

1. constrain the programmer by the syntax and semantics of the target language; thus, editing tasks that are simple in text editors can be quite complicated in these editors,

2. usually support only one language, and if this is not the case, the system has to provide support for different languages. That means the system has to integrate the lexical analyzer, parser and pretty printer of each different language to the editor, which results in tremendously increased costs and complexity of implementation.

It appears that the advantages of syntax-directed editing are
worth the increase in complexity of the editing tool to support this. At the very least, such editors eliminate the situation in which the first few runs of compilation serve only to detect typos and missing semi-colons. However, in situations such as editing algebraic expressions or documentations, the text view is much more convenient. In CPS and ALOE, the lower level language constructs, like expressions and assignments, are entered as text strings. Parser and lexical analyzer are embedded in these systems to take care of data entered as text. In TRIAD, both kinds of editing operations are also provided. The text view and the structure view of a program, or project-related information, are like two sides of a coin; neither one of them is a complete view.

In addition to editing text and programs, TRIAD also provides editing commands to edit the forms which are similar to templates in CPS and ALOE but are not necessarily programming language-oriented, as well as to form trees which structure all software-related information. These two sets of editing commands permit a user to develop his/her software in a more flexible manner than is possible with a system restricted to only top-down design. These high level editing commands are also useful when a user is putting re-usable modules together to create a new module.
2.5.2. Development Tools

Besides editors, most of the programming systems also provide tools such as (incremental) compilers and program debuggers for program development. Other structure editors have software routines to perform text reformatting or word processing. Apprentice contains a "drawer" which can display a plan in a graphical format. In Gandalf, since projects developed in this system are long-term, large-scale projects, different tools are provided to deal with information at different levels. The Incremental Program Constructor (IPC) provides syntax-directed editing and dynamic debugging for each programmer to develop his/her program in relative isolation from other programmers. Functionality-wise, these development tools are sufficient for program or project development.

In TRIAD, the Software Information Editor can create, edit, manipulate, and retrieve all software-related information; the Analyzer is designed to perform consistency checking, data and control flow analysis, etc. One of the unique features in TRIAD is that these development tools are driven by methodologies. That is, these tools are generic and adaptable. Thus, the tools support development/maintenance according to different methodologies.
2.5.3. Maintenance tools

Since programmers spend at least 40% of their time in documenting and maintaining programs, the need for maintenance tools is obvious. However, no specific maintenance tools are provided in the programming systems mentioned above. The editor is still the only tool that a programmer may use to maintain a program. One of the reasons for this is that the program structure and documentation are not standardized, so it is extremely difficult to design any maintenance tools. Programs are usually stored as low level syntax trees. The development history and important design decisions are all buried in the non-standard program comments. Therefore, it is not possible to provide any systematic support for maintaining programs.

In Gandalf, different versions of a project information is maintained by the System Version Description (SVD) and Project Management Tools. Such support is particularly useful for long-term project development. In TRIAD, the TRIAD/SIE can behave like a database query system to retrieve project-oriented information to help information maintenance; the Analyzer is also designed to handle versions control; and the Tuner is designed to enhance the Methodology Library (that is, the systems maintenance). Due to the fact that its information structure is more standardized by using
methodologies, TRIAD is able to provide better maintenance support than other related systems. It is also possible that TRIAD can provide methodologies for certain basic information maintenance. However, such maintenance support still cannot meet the maintainer's demand. It is therefore an important and immediate research issue in all areas of software engineering.

2.5.4. Integration of Tools

Using the traditional operating system to manage and control different tools in a programming or project development environment is no longer a satisfactory solution. In order to improve the productivity of program or software development, users have been demanding more control of and access to the tools. In traditional computer systems, each tool usually creates a mode. Every task can only be done using a tool in the right mode, and thus many useful capabilities of one tool can not be used when a user is in a different mode. Flipping back and forth between compiler and editor to correct errors is a well-known example. Another is that many useful editing functions can not be used outside the editor. Other well-described complaints about this "mode problem" can be found in [Tesl81].

The "mode problem" is mainly due to the poor integration of tools. In these systems, tools are isolated from each other and the
operating systems dominate the control of inter-communication between these tools. More specifically, only the operating system can invoke different programs defined in each tool. In the meanwhile, the control is transmitted to the process running that program, thus making unavailable the capabilities of other programs. In an "integrated" environment, the distinction between operating system and various applications fades. Every useful capability of each tool is always available to the user to apply to any information. For example, Smalltalk is a personal computer system that includes file system, process management system, graphic capability and compiler all in one system while eliminating the modes from its environment.

Integration of tools does not mean combining all different tools into a huge monster. It simply means that the services provided by all tools should be "uniformly" designed, pushed forward to the user interface, and smoothly interfaced with each other. In TRIAD, this is achieved by having all tools share the common databases using a uniform data representation, and including all useful capabilities of different tools in the user interface. Therefore, there is no physical distinction between tools, and the user interface is the only mode. Thus, one mode is no mode.
2.6. Customizability and Extensibility

In the literature, software customizability and extensibility together are often called software adaptability. In general, this includes the adaptability of a software to different computer hardware. However, only the adaptability of software to software will be discussed here. Customizing a software is to "tailor" a software in such a way that more specific user-oriented applications can be supported, but the quality or main functionality of the software remains the same. The extension of a software may involve adding new features to the software in order to support new applications. So far, some of the most promising techniques to support customizability and extensibility are all based on the concept of software abstraction [Boy179]. This notion of software abstraction can be found in various programs and software implementations in the shape of abstract syntax, data abstractions, control abstractions, macro expansions, parameterized functions, substitutable library routines, step-wise refinement, etc.

Many program editors maintain the abstract syntax trees for programs. Thus, one can customize the editors for different languages by supplying a grammatical description and the mapping between the concrete and abstract syntax. Structure editors like ED3 can also be customized for different applications in the same
manner. Customization of programming environments is much more complicated. In CPS, all the data structures and tools are tied to a language; therefore, customization of such an environment for different languages or different problem domains is not possible.

ALOE (A Language Oriented Editor) is a program editor generator which separates language-independent features from language-dependent features and can generate an ALOE instance from a grammatical description of a language. A wide range of ALOEs can be generated. On one end of the scale is a set of purely syntactic program editors which use only the environment specific routines. On the other end of the scale one can generate a fairly sophisticated environment like Gandalf which contains complex routines, extended commands, and its own version of the environment specific routines.

Though its application domains are different, the customization of the TRIAD environment is very similar to that of ALOE. TRIAD separates methodology-independent features from the methodology-dependent features. Thus, database structure and tools are generic to all methodologies. However, TRIAD provides a tool Tuner which is explicitly designed to support the customization. Tuner itself uses tuning methodologies to tune the other methodologies. The "tuned" methodology can then be used to drive the generic tools accordingly. Methodology-specific commands can be
easily added to the User Interface as extended commands, and thus the entire environment is customized. This unique feature of Tuner makes TRIAD a self-supported, customizable software environment. However, TRIAD does not generate instances of TRIAD as is the case in ALOE. We think that generator systems are only useful in generating well-defined products such as editors or compilers. In the case of programming or software environments in which local and global evolutions are taking place constantly, new applications are likely to emerge along with experience. Therefore, customizing the environment is a more flexible and cost-effective approach than generating instances of the environment for different applications.

Other than Emacs, none of the related systems addresses the issue of extensibility. A fundamental difficulty in achieving an extensible system is that the implementation languages themselves are not extensible. Therefore, only very limited extensions are possible. Emacs supports extended commands which can be defined by a user using macro expansions. A version of Emacs run in UNIX also provides a simple LISP-like language called MLISP (Mini-LISP) for the user to define more complex commands, as well as new library routines which can be loaded to extend (and customize as well) the functionality of text editing in Emacs.

TRIAD supports both the extended commands and the user-defined
library routines. This is because TRIAD's commands are first carefully designed based on their abstract concepts, and then carefully implemented by using the techniques of macro expansion, parameterization, subroutine or function substitution, and generic tools design. The implementation of the prototype has been controlled in such a way that it will not restrict the extensibility of this prototype.
3. WHAT ARE SOFTWARE METHODOLOGIES?

Archimedes ... he used anything and everything that suggested itself as a weapon to attack his problems.

-- J. L. Heiberg

Methods are masters of masters.

-- Talleyrand-Perigord

In this chapter we discuss, in detail, the nature of software methodologies. This discussion provides a basis for the approach to supporting methodologies presented in this thesis.

It is useful to go beyond the dictionary meaning of a methodology and to distinguish a software methodology from the tools or languages which support this methodology. It is also important to distinguish the mechanisms for enforcing a software methodology from the methodology itself. This chapter first answers the question "what are software methodologies?". It then presents the model used in TRIAD to formulate these methodologies. The mechanisms for enforcing methodologies, using a form-based approach, are discussed in Chapter Five, and the tools which support

1methodology: 1. a system of methods, principles, and rules, as those of an art or science. ... - The Random House Dictionary
The desired characteristics of software methodologies are first discussed in Section 3.1. Building upon this discussion, the reasons for choosing the attribute grammar form model to formulate these properties of methodologies are given. A working definition of software methodologies is also derived from the model. In this chapter, the models will be discussed only in informal terms in order to illustrate the intuition behind their use. However, the formal definitions of grammar forms, attribute grammars and attribute grammar forms are given in Appendix A.

Though researchers have often proposed using the computer systems as models for the human brain, they cannot prove whether such a model is complete or not. Similarly, we are not able to prove the completeness (or incompleteness) of our model for software methodologies. Nevertheless, we believe that this working definition explains many important characteristics of software methodologies and that thus serves its purpose well. The usefulness of this model will be demonstrated in different ways through examples in later chapters.
3.1. Characteristics of Software Methodologies

A software methodology is an assemblage of methods, principles, and rules, which guides the designer or implementor of a software project to carry out his/her tasks in a systematic and consistent manner. Hereafter, the term "methodology" will refer to either a software methodology or a programming methodology.

A methodology should not be merely a collection of instructions by which the user's success depends on his/her having chosen the right instructions. An unreliable methodology could be worse than no methodology at all. Therefore, a good methodology should provide the user with precise guidelines as to when and how to apply each instruction, so that the user can use the methodology to carry out the solution in a systematic way. Most of the methodologies being used for software or program development are imprecisely described and, thus, cannot be enforced systematically. Many methodologies have been developed: for example, data abstraction techniques for data structure design; Jackson's methodology, top-down design, structured design, etc. for program design; system decomposition, layer approach, etc. for system design; and many others for program verification, compiler design, database design, and so forth. These methodologies all provide good design techniques based on some design principles or experience. They are all described using
natural language and are not necessarily procedural. It is important that a methodology be precisely described, so that different users of the methodology can apply the methodology in a standard way.

As we all know, there is no single methodology that can guide the users to solve all software problems. Software projects in different problem domains usually require different methods of solution. It will be extremely difficult if one tried to use the strategies of "Chess" to play "Go" or vice versa. That is, methodologies are problem domain dependent. A software methodology is usually designed based on the understanding of a problem domain, the experience accumulated in the past, or theorems available. When a problem domain becomes more specific, problems in that domain are better understood and, therefore, a more detailed, precise methodology evolves.

A useful methodology does not have to be a fully automated procedure or as detailed as a step-by-step problem solver. There are many useful techniques, at various design levels, which cannot be fully automated, or are not yet automated, but which should be encoded as methodologies to support the software development. That is, any useful concepts, patterns, techniques, rules, disciplines, theorems, facts, or plans should be reflected in methodologies. More simply, useful experience should be reflected in methodologies.
One of the most important characteristics of methodologies is that methodologies evolve along with users' experience. Usually, such evolution continues until it can be fully automated and optimized to a satisfactory degree. A more general methodology can evolve to many more specific methodologies as experience is gained in more specific projects in a problem domain or its sub-domains. On the other hand, a well-developed methodology, provided in a more specific domain, can also be used in a more general domain by removing some domain-dependent restrictions.

As new technology and new experience emerge, methodologies evolve. This implies that the model used to encode methodologies should allow such evolution, and this further implies that the model should also be evolvable. In other words, the model itself should be able to model itself.

One characteristic of existing methodologies, which is usually considered to be a drawback, is that sometimes the methodologies dominate the user's thought processes. A methodology usually provides a specific way of producing the solution for a problem. But, there are often many other ways to solve the same problem, some of which could even be better than the one suggested by the given methodology. At ten years old, Gauss, "the Prince of Mathematics", refused to use the adding method taught by his teacher to add a
number series from 1 to 100. Instead, he implicitly used the formula \( n(n+1)/2 \) and obtained his answer in seconds. However, since it is the responsibility of the methodology (or the teacher) to guide the user's thought process, such limitation is probably inevitable. In order to alleviate this problem, adequate help information and alternatives should be provided so that a methodology will not completely limit the user's creativity.

Most importantly, a software methodology should guide the user's thought process during the development of a software. Depending on the methodology, such guidance can be provided in one or more of the following ways:

- provide a general framework of the plan to attack the problem,
- provide a pattern to solve a local problem and explain how this pattern should be applied,
- give instructions about how to perform the next step toward the solution,
- show examples if the user can not understand the instruction,
- give suggestions if no specific instruction is possible,
- provide all alternatives at some critical point and let the user decide what to do,
- prompt or remind the user to supply some information needed,
- inform the user whenever the rules of using the methodology are violated,
- inform the user of any inconsistency of the information he supplied and tell him the reasons.
Certainly, there is no magic about what a methodology can do. The degree to which a methodology can guide the user's thought process apparently depends on how fully a methodology "understands" the problem domain.

To summarize, software methodologies should

1. be precisely described,
2. be problem domain-oriented,
3. reflect useful experience (patterns, reusable modules, etc.),
4. evolve along with experience,
5. not limit the user's creativity,
6. be customizable to suit more specific problem domains, and
7. guide the user's thought processes in a consistent and systematic manner.

A difficult subject, which needs to be studied further, is the relationship among different but related methodologies. In a way, a methodology usually takes one view of a problem and suggests a particular method to solve the problem. If more than one view is possible, more than one methodology is also possible. For example, a methodology may specifically suggest that the user study the input/output data structures first and then design the functionality of a program to correctly transform the input structures into output structures. Another methodology may suggest specifying the functionality first and then designing the data structures. It
would be useful if both methodologies are available and a user could select the one with which he/she is more experienced. To cook a meal, a person can either decide the dishes first and then prepare the ingredients, or he/she can determine the ingredients that are available and then decide on the dishes. The relationship between such methodologies is not clear. Whether or not it is better to combine both methodologies into one is problem domain dependent and cannot be answered in general. There are also methodologies which can be easily combined as a hierarchy. For example, a methodology for specifying a program design and a methodology for program implementation can both be used in developing a program. However, when a program is developed using two different methodologies, the global consistency of the development is no longer guaranteed by either methodology.

3.2. Methodologies and Languages

Grammars, or more specifically, the context-free grammars, have been used to specify the syntax of programming languages for a long period of time. The semantics of the languages is usually implicitly defined in terms of translation processes inside the compilers. To fully understand the semantics of a language, one has to understand the compiler details. A newer approach, proposed by Knuth [Knut68], uses the attributed grammars to specify the syntax and semantics of
a programming language such that the semantics can be understood in terms of the semantic rules of attributes associated with each symbol of a production. These attributed grammars are also perfectly capable of generating machine code for derivation trees constructed using the grammars. That is, such grammars can specify the structure (syntax) of the languages and provide a way for the machine to understand the specified meaning (semantics) of the structure.

One should note that a language grammar alone does not teach people how to speak; it only helps people understand the language. In other words, methodologies are not languages. Moreover, the characteristics of methodologies discussed earlier should be useful to clarify the following:

- A methodology is not the "product" of using the methodology. That is, a methodology is not a program, a software system, documentation or anything produced by using the methodology.

- A methodology is not the "notation" used in the development process. That is, a methodology is not the language to express the product, nor the representation of the methodology itself.

- A methodology is not the "tools" used in the activity of problem-solving while using the methodology. For example, methodologies are not the editors, compilers, debuggers, etc. which could be required for using the methodology.

- A methodology is not even the "activity" of problem-solving while using the methodology. The activity is simply a realization or an execution of the methodology.

A methodology is, in its most general sense, the collection of abstract concepts of "how" to solve a problem. Its representation
can be as abstract as a mathematical theorem, a map, or a metaphor, or as concrete as having someone solve the same problem in front of your eyes, step by step. The language used to express the product (programs, documents, etc.) is also immaterial to the methodology. This indicates that methodologies are not languages, and, therefore, grammars alone are not sufficient to model methodologies.

3.3. The Syntax and Semantics of Concepts Embedded in Methodologies

Programming language constructs — like subroutines, block structure, and data abstractions — have underlying concepts. The syntactic representations of these concepts are quite different in various programming languages and the semantics associated with these concepts are also implemented differently in various language compilers. For example, the concept of function invocation is slightly different in different languages due to its implementation (like call by-value, by-reference, or by-name). Similarly, the concept of data abstractions also has different syntax and slightly different semantics in different languages. However, they all achieve similar effects and thus are treated as the same methodological concept. In other words, the syntactic representation and implementation of the semantics are immaterial as long as the same effect can be achieved.

If the semantics of a programming concept can be realized in
terms of data structures and procedures in an existing programming language, then it can be incorporated into that language by using a preprocessor for the compiler. But, the semantics of a programming concept is restricted by the data structures and control procedures supported by that language. For example, consider the following "if-then-else" construct:

```plaintext
if (condition C is true)  
  then (do task A)  
  else (do task B)
```

The semantics of this construct is clear as long as C is a decidable predicate with only two possible outcomes and A, B are any two actions. However, in a programming language, the "if-then-else" concept is meaningful if A, B and C are also meaningful to the compiler. In other words, one can use the pattern "if-then-else" to say almost anything in a natural language but only very limited things in a programming language. If we do not require a computer to understand the meaning of A, B and C, then this syntax can be used to express many other concepts. It is like the pseudo code of a program, which has the same underlying concept as the actual programming code, but is not bound to express that concept using the language-oriented data types and control procedures. The semantics of the pseudo code can not be understood by the language compiler, but it can be understood by a programmer. In this sense, the pseudo code can be regarded as a methodology for writing that program.
There are other concepts the semantics of which can be implemented using a programming language, such as sorting of an integer array, searching for a string in a text file, or linear regressional analysis using the general linear model. The methodologies for carrying out these tasks can be automated by programs, and thus the implementations of these methodologies are no longer important to the user. That is, if a concept (or methodology) can be automated, the semantics of this concept can be pushed up to the syntax level. So, a user can issue a command such as "Sort the array X in ascending order" without having to know or care about the methodology of sorting.

As long as the syntactic structure of a language is sufficient to support the underlying algorithm to be performed, one can view the syntactic structure as supporting the methodology. Generally speaking, the syntax provided by a language to support a concept is dictated by the implementation of the semantics of that concept. For example, type declaration of the arguments in a function invocation is required in most languages but not in LISP.

There are still other concepts which are user-oriented and are not yet supported by any programming languages. For example, the Structured Design Methodology [Your79] suggests that a module should be decomposed into smaller modules until different modules have low
data cohesion and high function coupling. Since a programmable
algorithm to enforce a methodology for achieving such decomposition
is not yet available, no programming language can really support the
entire concept of structured design. However, such non-procedural
methodologies are still useful in program or software design.
Similar to pseudo code mentioned above, we can still provide the
syntax to denote the concepts behind a methodology as long as the
user can understand the concepts. For example, the following pseudo
code is also non-procedural:

```plaintext
function: check-unique-element(S)
{ if (there are the same elements in the set S)
    then (keep only one of them in S)
    else (do nothing) }
```

This does convey the concept of "check-unique-element" to the user
and indicates what should be done. The concept also implicitly
indicates how the task can be done, although not necessarily how it
can be done in the most efficient way. The underlying syntactic
structure in the example is the "if-then-else". This illustrates how
we can provide syntactic structure by which the user can use, as
well as understand, the methodology.

At one end of the spectrum of all methodological concepts,
there is a set of very specific methodologies which can be fully
automated. At the other end, there is a set of very general
methodologies which can only serve as a check-list or a reminder and
in which the user has to do most of the work. As in Mathematics, the method to solve quadratic equations of a single variable or a system of linear equations can be fully automated, while to solve integrals only some general techniques like "u-substitution" and "integration by parts" are available. Good examples of automated methodologies in computer science are the system generators like compiler generators, editor generators, etc. Here, not only is the methodology for writing a compiler for a particular language automated, but also the methodology for writing a compiler for a language in general is automated.

In order to further illustrate what we mean by pushing the semantics to the syntax, let us again consider the example of methodologies for integration. Given a definite integral and an integration method such as "u-substitution", it is the user's responsibility to recognize the exact portion of the original function to be substituted in order to simplify the integral until the substituted integral is in the form of some standard formula which can be solved by programs. When people adapt the Simpson's Rule (which requires a lot of calculation and becomes feasible only after computers are available) to approximate the value of any definite integral, the semantics of integration methods fades. We no longer care whether it is done using "u-substitution" or "Simpson's rule" or whatever; we can just talk about a definite integral as it
is defined. To evaluate a definite integral is as simple as writing down its standard representation (the syntax). No understanding of how it is evaluated is necessary. By the same token, as we gain more experience in software or program development, more semantics of methodologies will be pushed to the syntax level; thus the only important semantics behind the syntax will be less and less procedural.

Thus far, we have pointed out the following issues:

1. A single methodological concept can be supported by different languages in a variety of ways based on the implementation of its semantics in a particular language.

2. If the implementation of a methodology can be automated, the semantics can be pushed to the syntax level. Otherwise, the semantics should be conveyed to the user.

3. Existing methodological concepts form a spectrum depending on the degree to which they can be automated or supported by syntax.

Our goal in this thesis is to present an approach to supporting methodologies. Thus we would like to point out that if a methodology cannot be automated, we can still provide syntactic structure to encode the concepts behind it. For example, consider the following production:

\[
\text{<program> := <input/output specification> <initial statement>}
\]
\[
\text{<program body> <final statement>}
\]

This suggests that a program must be composed of distinct initial, body, final parts, and must be preceded by an input/output
specification. This concept is usually implicitly interpreted by the programmer and kept in the comments or documentation. Eventually, it is refined into language syntax like type declaration, begin-end block, etc. Typically, such concepts have not been important to the language compiler, but have been useful for programmers during program design. The symbols appearing in the production are language-independent, and they are designed to capture higher level concepts. The notion of interpretation of a symbol permits a user to generate a more specific instance of that concept under the framework provided by the entire production. That is, the user can interpret the symbol using some notation (not necessarily programming language) to obtain an instance which also bears the same underlying concept. The grammar forms model [Gins77] [Blat79a] [Blat79b], which permits us to encode such high level concepts, is discussed next.

3.4. Conceptual Trees and Grammar Forms

A production in which the symbols denote high level concepts (as opposed to programming language concepts) is called a conceptual production. If the concept behind a symbol can be further refined, another conceptual production can be associated with that symbol. The process of expanding a concept into more detailed concepts by using another conceptual production is called a refinement. A
grammar consisting of a set of conceptual productions is thus called a *conceptual grammar*, and a tree obtained through a sequence of refinement processes by using these conceptual productions is called a *conceptual tree*. A *grammar form* is composed of a conceptual grammar and an infinite set of interpretation rules. This conceptual grammar is a "template" for other grammars, called the *interpretation grammars*, that have similar structure.

A pre-defined conceptual tree reflects a plan, a pattern, or a template which can be used in the development of software projects or programs. The main difference between a conceptual tree and a syntax tree is that the semantics of a conceptual tree is to be understood by the *user* rather than the *compiler*. By using a conceptual tree as a whole, we can suggest, to the user, *good patterns* which are known to be useful in similar problems. In programming languages, we can also find some very low-level patterns. For instance, iteration can be written in several different ways in a language and each method can be thought of as trying to encode certain patterns of iteration. Conceptual trees can be used to encode useful patterns for more human-oriented concepts and to suggest them to the user in a much more direct and explicit manner than can programming languages. For a detailed justification for the use of the grammar form model, the reader is referred to [Soni83].
There are an infinite number of interpretation rules. Therefore, corresponding to each conceptual tree, there are an infinite set of interpretation trees, each of which is simply a relabeled version of the conceptual tree. It is obvious that the conceptual tree provides the abstract structure for all the corresponding interpretation trees, and that the infinite set of interpretation rules allows one to record all software-related information (not necessarily program-related information) into such a structure.

**Significance of the Conceptual Tree**

Note that the conceptual tree serves to trigger the user's thought processes when solving a problem. However, perhaps the more significant conclusion is that the conceptual tree is a concrete representation of the development processes, which can be manipulated automatically. Thus, the precise representation of the methodological concepts permits the design of an editor to manipulate these concepts. In other words, the purpose of software methodologies is to suggest "how" to solve a software problem, as well as to record the process by which the final product of the whole problem-solving activity is created. The conceptual trees provide the structure of the process for the final product. The final product — the solution as well as the development history
(that is, all the software-related information) — is thus structured based on the underlying conceptual trees.

The syntactic structure of concepts and patterns alone are by no means the methodology. The semantics of a conceptual tree has to be understood by the user in order to interpret and refine a concept appropriately. It is the responsibility of a methodology to inform the user about the semantics behind a conceptual tree and the relationship between conceptual trees. An overview of the approach for providing the semantic support is given below.

3.5. Semantics of Conceptual trees and Attribute Grammars

The purpose of the section is to present, for the sake of completeness, the extended grammar form model envisioned for TRIAD. Details presented here are not directly related to the contributions discussed in this thesis.

Briefly, an attribute grammar is a context-free grammar in which each symbol is associated with a set of attributes. These attributes are further divided into two disjoint sets of synthesized and inherited attributes. For each attribute occurrence of a symbol in a production, there is a semantic rule which updates the value of that attribute. Each production thus has a collection of semantic rules for all of its attribute occurrences.
The most important idea behind the attribute grammars is the use of synthesized attributes (which are based on the attributes of the descendants of the non-terminal symbol) and inherited attributes (which are based on the attributes of the ancestors). Synthesized attributes are evaluated from the bottom up, while the inherited attributes are evaluated from the top down. Since the development process of most programs is not strictly top-down or bottom-up, these "dual" attributes are naturally useful to describe the semantics. The semantic rules are used to assign "meanings" to strings of the context-free language. This is done by evaluating each inherited or synthesized attribute of a node in the derivation tree using appropriate semantic rules. More specifically, an initial semantic tree is first constructed based on the context-free grammar with all the attributes undefined. Then the evaluation of attribute values begins according to some evaluation algorithm and ends, either when all the values of attributes are defined, or when no further evaluation can be done. Various evaluation algorithms can be found in [Knut68] [Lewi74] [Boch74] [Schu76] and will not be discussed here.

As claimed by [Knut68], attribute grammars define each language construct only in terms of its "immediate environment" and thus minimize the interconnection between the definitions of different parts of the language. This means that the attributes and semantic
rules permit
- the **programmer** to understand the language constructs based on its specification (definition time), and
- the **computer** to understand the semantics of a derivation tree based on the updated attribute values (evaluation time).

If we do not restrict ourselves to the domain of programming languages, attributes and semantic rules can be used to describe the semantics of higher level concepts such as data abstractions or modularization which are to be encoded in our methodologies. As an example, consider the following two productions in a concept grammar:

P1: `<data abstraction> ::= <purpose of this data abstraction>`
    `<lower level data abstractions>`
    `<canonical form>`
    `<operation>*`, and

P2: `<operation> ::= <purpose of this operation>`
    `<input/output types>`
    `<semantic equations>`
    `<implementation>`.

Suppose a methodology requires that the `<lower level data abstractions>` are only those data types used in the `<input/output types>` of `<operation>`s. Also, suppose the methodology requires that the `<input/output types>` of each `<operation>` are the types specified in `<lower level data abstractions>` or the data abstraction itself.
This can be realized using attributes and semantic rules in the following way:

<table>
<thead>
<tr>
<th>symbol</th>
<th>synthesized attributes</th>
<th>inherited attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>lower level data abstractions</td>
<td>not_defined, used_in_op</td>
<td>not_used, lower_level_types</td>
</tr>
<tr>
<td>operation</td>
<td>not_defined, used_in_op</td>
<td>not_used, lower_level_types</td>
</tr>
<tr>
<td>input/output types</td>
<td>not_defined, used_in_op</td>
<td>not_used, lower_level_types</td>
</tr>
</tbody>
</table>

All the attributes defined above take a set of data abstraction names as values. Used_in_op is the set of data types used in the <input/output types> of an <operation>. It is compared with the inherited attribute, lower_level_types, to see if there is any type which is used but not specified in the <lower level data abstractions>. If so, the value of the synthesized attribute not_defined will contain this undefined type name and pass it up to the <lower level data abstractions> to reflect this fact. Similarly, a type name specified in the <lower level data abstractions> but not used in any operation can also detected by means of the inherited attribute -- not_used. The semantic rules which perform some standard operations on the sets of type names are very straightforward and, therefore, are not given here.

This example, though simple, illustrates two important points:

1. The inherited attributes can be used to provide suggestions.
for the next step during the development of a project. For instance, when a user is specifying the input/output types of an operation, the value of the attribute, lower_level_types, can suggest that the user use only the types defined in it.

2. The synthesized attributes can be used to provide feedback to the user during the development of a project. Using the same example, the synthesized attribute, not_defined, of <lower level data abstractions> would inform the user whether a type used in some operation is not yet specified.

In other words, attributes and semantic rules can be employed in the domain of software development in very much the same way they are used in programming languages. If the language constructs are replaced by abstract concepts (or the conceptual trees discussed earlier), then the attributes are not necessarily programming language-oriented. If we consider the evaluation algorithm to be the "methodology" for the semantic evaluator to generate an executable code for a program automatically, then we can also design an algorithm for both the user and an analyzer to generate a complete software product in a semi-automatic manner. It has to be semi-automatic because the attributes may take informal notation such as English text as values and thus are not analyzable by
existing tools. This means that some semantic rules may have to be "performed" by the user.

The methodology to generate the machine code for a program written in a high level programming language can, as we all know, already be automated. The methodologies to generate programs from a concept or generate a complete software system from its requirement are not even formulated yet. In order to achieve this, we have been investigating different models as discussed above. Before we introduce the attribute grammar forms, it is useful to pause here and summarize the results of our observations.

1. Grammar forms can be used to provide the **structure** for the final products of software development, as well as to encode higher level **concepts**, **good patterns**, or experience in developing similar projects.

2. Attributes and semantic rules can be used to reinforce the encoding of concepts, patterns, etc. and to provide **suggestions** and **feedback** at the evaluation time (or the development time).

3. A semi-automatic algorithm for evaluating attributes (similar to those described above for attribute grammars) can be used to support a general strategy or to provide guidelines for software development based on the information
provided by 1 and 2 above.

Because the symbols, attributes, and semantic rules are no longer programming language-oriented, no effective algorithms for evaluating attributes are guaranteed. However, in the case of programming languages, an evaluation algorithm may be specified as follows [Raih77]:

```plaintext
repeat
  choose an applicable attribute instance a in the derivation tree;
  evaluate the value of a;
until
  no choice of a is possible;
```

It is obvious that such an algorithm does not help much in evaluating software attributes in general. Even though this may be the case for many software problems, we believe that the first two points mentioned above can still help the user a great deal. If the problem domain becomes more specific (at least not as general as generating machine code for "any" program written in a language), it is possible to design a more specific procedural algorithm.

As a final remark on the attribute grammars, before we proceed to the attribute grammar forms, the evaluation time of attributes is also important. When using attribute grammars, the derivation trees are usually constructed in advance before any attribute is evaluated. However, as implicitly mentioned earlier, this should
not be a reason to restrict the use of attributes. We would like to evaluate all attributes, to whatever degree possible, during the tree construction time, so that the user can be promptly informed about what to do next (suggestions), as well as about what was done wrong (feedback).

3.6. Attribute Grammar Forms and Software Methodologies

It should be clear at this point that the support of methodologies requires at least all the features facilitated by both grammar forms and attribute grammars. This necessity motivated us to merge the grammar forms and attribute grammars into an extended model called the attribute grammar forms. The formal definition of this model is given in Appendix A.

Informally, an attribute grammar form consists of a concept attribute grammar, an infinite set of interpretation rules, a set of attributes, and a collection of semantic rules for the concept attribute grammar. These are all direct extensions of grammar forms and attribute grammars. In addition to these, there is a set of attribute interpretation rules which maps the attributes and semantic rules of the conceptual attribute grammar to that of the interpretation attribute grammar so that the attributes and semantic rules can still work properly in the interpretation attribute grammar.
By making use of this extended model, we obtain a working definition of software methodologies (in Appendix A) which can encode a methodology as precisely as possible. The intuition behind this working definition is given below:

A **Software Methodology** consists of the following three parts:

1. **A conceptual attribute grammar**: This conceptual grammar provides the structure for software-related information. The attributes and semantic rules are used to keep track of the semantics of the concepts and the relationships between concepts. Moreover, the attributes and semantic rules are more flexibly used here in order to guide the user's thought process, as well as the computer analysis during the development.

2. **A set of conceptual trees**: This set of conceptual trees provides good patterns which are useful for developing similar software problems. The symbols in these patterns can be pre-interpreted or the attributes can be enhanced in order to reflect the experience of using these patterns. New conceptual trees can also be added to this set and, thus, become new patterns.

3. **A set of semantics evaluators**: A semantics evaluator is like the driver of the entire methodology. It specifies the order
of applying a concept production (or a conceptual tree) and
the procedure of applying the semantic rules to evaluate
attributes. The actual execution of this evaluator could be
automatic, semi-automatic or even manual. As mentioned
earlier, as soon as an executable algorithm is available,
the evaluation can be automated and thus we can push the
concept up to the syntax level. The effectiveness of a
methodology is therefore determined by the degree of
automation and effectiveness of the semantics evaluator.
4. PROTOTYPING OF SYSTEM DESIGN

It is wisdom to know others;
It is enlightenment to know one's self.

-- Lao Tzu

After the practice of the software life-cycle was widely accepted (at least in academia), there were strong objections (mostly from industry) to this practice as the only paradigm for software development [McCr82] [Glad82]. The conflict arose because, in the real world, the nature of software problems and various external pressures prohibit the designers and implementors from adhering to the cycle. For instance, when the time/cost factor and the requirements from users become critical, the designer has to find out, as soon as possible, if the design is feasible and will meet the specifications. Since any modification of the requirements or design would require a new iteration of a life-cycle, it is risky to stay in the present life-cycle and evaluate the system after having invested in a costly design and implementation. Therefore, most industrial environments do not believe that a rigid adherence to the life-cycle practice is feasible.

The "conflict" between these two practices (the software life-cycle vs. the protection of investment) is due to the differing
nature of the software problems in the academic world and the industrial world. That is, if a software problem is fully understood and has stable requirements from which a suitable design can be made, then the life-cycle is a good practice to adopt for the development of such software. However, if the problem has too many unknown factors and the likelihood of change is large, the life-cycle practice is not suitable. Therefore, industry has adopted the concept of "rapid prototyping" [Balz82].

Rapid prototyping emphasizes the quick implementation of a software design, and can provide "accelerated feedback" for the early stages of analysis in the life-cycle. Such an approach is particularly useful when there are areas of risk that only experience with a running system can clarify. The technique of rapid prototyping can be very effective in identifying the unknown factors that make changes or improvements necessary at early stages of software development. Although a prototype cannot show all aspects of the final product, it is still beneficial to identify some, if not all, of the critical problems at early stages. The concept of "rapid prototyping" is an old technique in industrial environments. Many existing systems or products were developed using this approach, as, for example, in the automobile and aircraft industries.
The "conflict" between the iterative life-cycle practice and the iterations through prototypes is, in fact, not a conflict. They are simply the two extremes — one which emphasizes developing the final system systematically and the other which emphasizes early visibility using prototypes with limited functionality.

In the case of TRIAD we followed the life-cycle approach faithfully until we realized that we needed the feedback from users in order to see how the methodologies could be enforced. Since there were no existing systems which explicitly supported methodologies and user feedback had become critical to the rest of the system design, we decided to build a simple prototype. The implementation of this prototype started in May 1980 and took only about four weeks to complete.

The TRIAD prototype was implemented in LISP using the FRL-package [Robe77] on a DEC-20. The study of TRIAD's system design, using the prototyping approach, is described in this chapter. The purpose of building this prototype is highlighted in Section 4.1. Section 4.2 discusses the implementation details. Experience gained from this prototype is then discussed in Sections 4.3 and 4.4.
4.1. Goals of the prototyping approach

Since the prototype had to be implemented in a short period of time, it was clear that the complexity of the prototype had to be substantially reduced from that of the final product. However, if the prototype were too simple, it would be very difficult to perform any significant analysis nor obtain any meaningful experience from it. In order to reach the equilibrium between these two conflicting factors, a number of goals were first listed for the prototype, and those which could not be implemented within the time-limit of about one month were eliminated. Such a list, which states the original goals in building this prototype, is given below:

1. **Data base design**: Since the major characteristic of TRIAD is the structuring of information according to methodologies, the design of the data bases was critical to the rest of the system design. The data bases — the refinement tree data base and the methodology description data base — were designed as hierarchies of data abstractions. These data abstractions were implemented in order to study their complexity and completeness.

2. **Methodology encoding**: A simple methodology, called Transaction Processing Methodology, would be implemented in order to see if the grammar form model is sufficient to
encode a methodology.

3. **Structuring information according to methodologies:** A set of editing functions would be implemented to structure information according to a methodology. However, this set of editing functions should be generic to all methodologies and would be used to integrate the two major databases.

4. **Information manipulation:** Since the information collected is not necessarily programming-language oriented, a set of functions to manipulate software information was to be implemented, based on the underlying concept grammar, in order to provide useful views of this information. As the value of the information depends on its effective use, rather than on the amount collected, this issue is critical to the system design.

5. **Standardization of information structure:** The information for different programs, collected according to a single methodology, should have less variation in its structure. Programs were to be developed using a sample methodology and analyzed to test this belief.

6. **User interface:** Since TRIAD is intended to be a software environment, a highly interactive and friendly user
interface is necessary. It will not be clear to what extent the interactivity and friendliness should be developed unless a working prototype is available. The prototype should include basic help information and system prompts as the minimum support, and the experience gained in using such a prototype should improve the user interface design.

7. Run-time support: The integration of TRIAD and the external environment (operating system, file management system, compilers, etc.) should also be studied using the prototype. The goal here was to study what sort of run-time support is required and what sort of support can be obtained from the existing external environment.

8. Implementation language: It is difficult to decide the implementation language for the final TRIAD system before the design of the final system is complete. From several candidates (such as LISP, C, CLU, and Pascal), LISP and FRL were chosen to be the implementation language for the prototype. This was because LISP has dynamic storage allocation, and had the well-defined Frame package called FRL. The complexity of the resulting implementation and the performance of the prototype were to indicate if LISP was a suitable choice for subsequent implementations.
Not all of the goals listed above were met. For instance, since the prototype did not include major components like the display module, keyboard monitor, tuner, and others, we were not able to fully understand what sort of run-time support was needed from the external environment. Other goals were achieved to varying degrees and are discussed in Section 4.3.
4.2. Implementation of the LISP-Prototype

In this section, only those prototype implementation details which led to significant conclusions will be highlighted. Some statistical data is first given below:

Implementation Time: about 2.5 weeks (two programmers)
Implementation Code: about 3000 lines (55 pages)
Implementation Documentation: about 5000 lines (100 pages)
Number of Data Abstractions: 20-25
Number of LISP Functions: 160-170
Methodology Implemented: Transaction Processing Methodology

4.2.1. Overview of the Prototype Architecture

The LISP-prototype consists of five logical components: Refinement Tree Data Base (RTDB), Methodology Description Data Base (MDDB), Structure Editing Functions, High Level Editing Functions, and Display Functions. The organization of these components is shown in Figure 4.
1. Methodology Description Data Base (MDDB): In the LISP-prototype, a methodology description is represented as a grammar form, which is, in turn, a set of productions. Regular expressions are allowed to appear on the right-hand side of a production. Each production is pre-defined by the methodology designer and cannot be altered by users. The symbols, productions, and help information can prompt the user during the development. These productions are also
used by the structure editing functions to construct the refinement trees.

2. **Refinement Tree Data Base (RTDB):** A refinement tree is basically a derivation tree constructed using the underlying concept grammar. Each node of the tree contains a symbol in a production and its interpretation. The data base is simply a set of collections of refinement trees under different users. A refinement tree stores all the related information and the development history of a program or project.

3. **Structure Editing Functions:** This set of editing functions provides operations to create, manipulate, and modify the structure of a derivation tree. More specifically, these functions can refine a non-terminal symbol using the chosen production, interpret the regular expressions, access information stored in the tree nodes, insert, delete, or copy a subtree, etc. It is similar to a syntax-directed editor or a template editor. Through this set of editing functions, the RTDB and MDDB are integrated.

4. **High Level Editing Functions:** This set of functions is designed to manipulate the collected information based on the underlying concepts and some limited semantics. The
context-sensitive information based on the relations among
different symbols is handled by this set of functions. For
example, the collection of programming code from a
refinement tree and the sending of it to a compiler are
performed by a single high level editing function.

5. Display Functions: These functions are defined in terms of
the output functions available in LISP and FRL. Their
purpose is to display various types of information on the
screen. Different types of information to be displayed are
refinement trees, subtrees, tree nodes, productions, help
information, and system prompts.

4.2.2. Implementation of Data Bases

As mentioned earlier, the RTDB and MDDB are designed as two
hierarchies of data abstractions. Most of these data abstractions
are simple data structures such as records, lists, or tables.
Therefore, they were all implemented as LISP lists or lists of
pointers to other lists. The only exception was the refinement tree
which was implemented as recursive Frame structures in FRL. The
implementation of the generic tree structure is given below:

(FASSERT tree_id
  (NODEID ($VALUE (node_id)))
  (SYMBOL ($VALUE (symbol)))
  (INTERP ($VALUE (interpretation))))
The reader should not be bothered by the syntax. The first five "slots" stored information about the root node of a subtree, and the last two slots stored information about the tree structure (including both upward and downward links). Since productions (or templates) are used to construct trees, productions were implemented with the same Frame structure but contained empty values in the NODEID, INTERP, and PARENT slots. The LISP and FRL data structures used to implement these two data bases are shown in Figure 5 and 6.

Each data abstraction in the hierarchies has a set of operations which can create, delete, access, and modify the information stored in the data structures. These operations were all implemented as re-entrant LISP functions. Since LISP and FRL do not directly support data abstractions as such, it was our responsibility to ensure that only those operations defined for a data abstraction were used to manipulate an instance of the data abstraction. Another inconvenience in using LISP and FRL was that they support no user-defined types other than the generic Frame structures of FRL.
Figure 5: Implementation of RTDB
Figure 6: Implementation of MDDB
Without the help information, the productions of the Transaction Processing Methodology [Rama81] are given below:

1. \(<\text{transaction program}> := \langle\text{requirements}\rangle
\quad \langle\text{transaction processing program}\rangle\)

2. \(<\text{transaction processing program}> := \langle\text{define files}\rangle
\quad \langle\text{program body}\rangle\langle\text{output}\rangle\)

3. \(<\text{define files}> := \langle\text{transaction file}\rangle \langle\text{master file}\rangle\)

4. \(<\text{transaction file}> := \langle\text{tkey}\rangle \langle\text{key ordering}\rangle\)

5. \(<\text{master file}> := \langle\text{mkey}\rangle \langle\text{key ordering}\rangle\)

6. \(<\text{program body}> := \langle\text{initial part}\rangle \text{WHILE} \langle\text{EOF in transaction file}\rangle
\quad \langle\text{form unit and process it}\rangle \text{OD} \langle\text{final part}\rangle\)

7. \(<\text{form unit and process it}> :=
\quad \text{IF} \langle\text{tkey}\rangle > \langle\text{mkey}\rangle \text{THEN} \langle\text{flush master file}\rangle
\quad \langle\text{process transaction}\rangle\)

8. \(<\text{flush master file}> := \langle\text{process master records that are needed}\rangle
\quad \langle\text{read new master record}\rangle\)

9. \(<\text{process transaction}> := \text{IF} \langle\text{tkey}\rangle < \langle\text{mkey}\rangle \text{THEN} \langle\text{no match}\rangle
\quad \langle\text{match}\rangle\)

10. \(<\text{no match}> := \text{WHILE} \langle\text{same transaction key}\rangle \langle\text{read next transaction record}\rangle \text{OD} \langle\text{no match process}\rangle\)

11. \(<\text{match}> := \text{WHILE} \langle\text{same transaction key}\rangle \langle\text{read next transaction record}\rangle \text{OD} \langle\text{validation process}\rangle\)

12. \(<\text{validation process}> := \langle\text{validation step}\rangle\)*

There are many examples of the transaction processing problems: for example, the update of a Master Product File using Product Update Transactions, or the update of Customer records using Customer Payment Transactions. The productions of the concept grammar listed above can be used as the underlying concepts for a
large variety of transaction processing programs.

As the following example demonstrates (Figure 7), the concepts for solving certain transaction processing programs can be encoded in grammar productions which actually guide the thought processes of programmers. More specifically, the symbols of the productions (in bold-face) act as prompts to which a programmer can respond. The programmer's responses to (or interpretations of) the production symbols are optional (regular type-face). Note that the interpretations are given in a pseudo-language (though some programming language could have been used), and that some of the interpretations are deferred.

4.2.3. Editing Functions

All of the structure and high level editing functions were implemented in terms of the operations defined in two data bases. Fifty-three editing functions were implemented. Users could use these functions to create, delete, access and update information stored within tree nodes, as well as to create, delete, insert, copy and traverse refinement trees. However, when dealing with methodology descriptions, users only had access to read-only functions. Only authorized personnel could change the methodology descriptions. A subset of the editing functions was designed to ensure that refinement trees were constructed correctly using
Transaction Program: Product Master File Update

Requirements

Transaction Processing Program

Define Files

Name of Transaction File: Product Update File (PUF)
  Tkey: Product Number
  Key Ordering: Ascending

Name of Master File: Product File (PF)
  Mkey: Product Number
  Key Ordering: Ascending

Program Body

Initial Part: Open PUF, PF; initialize all variables;

WHILE "EOF in transaction file:
  DO form unit and process it

  IF tkey > mkey
    THEN flush master file
        process master records that are not needed: Write PF
        read new master record: Read PF
    ELSE process transaction

  IF tkey < mkey
    THEN no match
        WHILE same transaction key
          DO no match process: Print Invalid Transaction
          read next transaction record: Read PUF
        OD
    ELSE match
        WHILE same transaction key
        DO validation process
          validation step: Check if transaction file is valid
          validation step: Check if product description in
          PUF record matches PF record
        OD

  OD

final part:

output:

Figure 7: Interpretation of the Transaction Processing Pattern
for Product Master File Update
appropriate productions from a methodology description. Most of the structure editing functions are straightforward and will not be presented here. The other set of editing functions, called high level editing functions, are much more complex.

Recall that a grammar form provides higher level concepts, and that all refinement trees developed using the same concept grammar have the same underlying concepts. This fact was exploited by the prototype in the design of high level editing functions to ensure that all instances of the same pattern of programming were manipulated in a "uniform" way. The following list illustrates the high level editing functions which can manipulate information developed using the Transaction Processing Methodology. In order to improve readability, the functions’ names and parameters are given in an English format. The action taken by the system for each function is also briefly described.

1. Have I interpreted all <initial statement>?
   Action: search for nodes with the <initial statement> nonterminal and display only the un-interpreted nodes.

2. Have I interpreted all concepts?
   Action: traverse refinement tree and display deferred or un-interpreted concepts.

3. Refine <read next transaction record> of <no match>.
   Action: search for <read next transaction record> in the subtree of <no match> and apply subsequent concepts and their interpretations to it.
4. What is the name of <master file>?
Action: display the interpretation of <master file> node.

5. What are <validation step>s of <match>?
Action: display the interpretations of all the <validation step> nodes under the subtree of <match>.

6. Add concepts <validation step1>, <validation step2> and <validation step3> to <validation process>.
Action: check if the concepts to be added can be derived from a production associated with <validation process>.

7. Display the cross-reference for the variable MPF in <match>.
Action: use attributes of the variable MPF and its semantic functions to synthesize the request information for the <match> subtree.

8. What should I do about <no match>?
Action: display the help information associated with each production of <no match>. (In this case, there is only one production.)

9. What should I do about <read new master record>?
Action: display help information associated with the symbol <read new master record>.

10. Synthesize program.
Action: traverse through the tree, collect the nonterminal interpretations as documentation and synthesize the interpretations of terminal nodes as the program.

Note that the parameters used in the above examples can be replaced by any compatible instances. In other words, these functions can be applied to any refinement tree developed using the TPM methodology.
4.3. Evaluation of the Prototype System Design

As soon as the implementation of the LISP-prototype started, the process of evaluating the prototype began. The "exact" value of the experience gained from this prototype will never be known. However, a simple evaluation of this prototype, based on the goals we set in the beginning of this chapter, is presented in Section 4.3.1. An overall evaluation is given in Section 4.3.2.

4.3.1. Evaluation against Original Goals

1. Data base design: The design of the two major data bases using data abstraction techniques appears to be very satisfactory. It was easy to implement operations, easy to change the implementations of existing operations without effecting the rest of the implementation, easy to define functions on top of operations, and easy to maintain the code. The only problem was that implementations following the data abstraction specification were usually not efficient in terms of time or space. In spite of these inefficiencies, the security of the information and the maintainability of the implementation code are more important for an evolving system like TRIAD than gains in speed and space that could be obtained only at the expense of harmful "tricks" in the implementation of data bases.
2. **Methodology encoding**: As demonstrated by the example in Figure 7, the concepts encoded in a grammar form along with help information can guide programmers' thought processes in the development of transaction processing programs. If the problem domain is very specific, such as transaction processing problems, it can be seen that the programmer's job is greatly simplified. That is, the user of such a methodology is mainly interpreting a concept and selecting a production to refine that concept in accordance with suggestions from the help information. It is also possible that the interpretations of some concepts can be "pre-interpreted" in the productions. For example, the interpretation of `<tkey> mkey>` is always clear and can be pre-interpreted as soon as `<tkey>` and `<mkey>` are specified. In such a case, the writing of a transaction processing program simply means a few keystrokes for selections and a few more keystrokes for interpretations. However, since methodologies for large scale projects were not implemented, this prototype was unable to answer the question of whether such methodologies can be effectively encoded.

3. **Structuring information according to methodologies**: It should be noted that the power of methodologies and high level editing functions was due to the structure imposed on
all related information according to concept grammars. The refinement trees reflected the development of programs and projects, and thus stored all the software-related information throughout the entire life-cycle. However, since methodologies for large scale projects were not implemented, this prototype did not allow us to fully test this point. Another problem was that an unparser, necessary to pretty-print the information stored in the created refinement trees, was not implemented. The rudimentary display functions implemented in this prototype would certainly have been inadequate for the task of handling all of the software-related information for a large project.

4. Information manipulation: In this prototype, no semantic functions to deal with context-sensitive information were implemented. This was partly due to the overly simple methodology we had, and partly because of the time limit imposed. The high level editing functions discussed earlier were probably only enough to manipulate the information generated from small problem domains. However, the manipulation of software-related information is one of the major goals of TRIAD; an approach towards achieving this goal is described in detail in Chapter Six.
5. \textbf{Standardization of information structure}: The LISP-prototype was never available to public users due to its poor user-interface, and thus data could not be collected with which to test this goal. In order to gain data, a pre-experiment was later conducted in a classroom situation. The details and results of this pre-experiment are given in Chapter Five.

6. \textbf{User interface}: Not only did the poor display functions cause the prototype's premature death, but other factors like poor interactivity, insufficient error messages, lengthy function names, unrefined help information, etc. contributed to its demise. We learned from the prototype that the human-engineering of the system should be a high priority task in any system design. This conclusion actually caused the re-design of TRIAD. It has been the case that no design decision could survive without consideration being given to human factors.

7. \textbf{Run-time Support}: The prototype was too simple to allow for any significant insight into the problem of deciding what kind of run-time support was needed. For example, we could not determine what kind of support was needed from the operating system to handle multiple users or memory
allocation.

8. Implementation language: Though LISP's dynamic storage allocation and interpretive capabilities are attractive, there are also many disadvantages to using them as the implementation language for the later prototype or the final system. For instance, they do not support user-defined types, their programs lack structure, their execution time is slow, their storage space is used inefficiently due to the large number of Frames for tree nodes, and finally, their programs are hard to read.

4.3.2. Feasibility of Methodology-Driven Systems

In a previous subsection, the evaluation of the prototype in the context of software environment was discussed. In this section, another evaluation of the prototype, in a more general context, is provided. Generally speaking, many computer systems can be classified, as sketched in the following rough diagram, according to their purposes and their approaches to the management of the information they collect.

<table>
<thead>
<tr>
<th>Data Base Management</th>
<th>Program Development</th>
<th>Software Development</th>
<th>Artificial Intelligence Systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>(DBMS)</td>
<td>(PDS)</td>
<td>(SDS)</td>
<td>(AIS)</td>
</tr>
</tbody>
</table>
As far as the information collected is concerned, the complexity and dynamics increase from left to right. In DBMS the data created and stored is nearly static and has few relationships between data. Therefore, few operations are necessary to create or update the information. The main issue in DBMS is effectively retrieving the collected information. Other issues such as database structure, data queries, and security, are approached with an eye to accommodating this fundamental goal. On the other end of the list, it is not at all clear what information has to be created and stored in the knowledge data bases of AIS. These knowledge data bases store information with very complicated inter-relationships among the data, and these inter-relationships are highly dynamic. The main issue in such systems is how to understand users' requests and automatically perform appropriate tasks. Therefore, their focus is more on what the knowledge is and how to create knowledge beforehand, than it is on the retrieval of information.

In PDS and SDS, the information created is either program or software-related information. Such information contains more inter-relationships than the data stored in DBMS, and has better structure (syntax) than the semantic networks used in AIS. The main issue in these systems is how to support programmers and software designers in the creation of information, on the one hand, and the analysis of this information in order to aid in users'
decision-making, on the other hand. The main difference between these systems and DBMS is that the information to be created and maintained by them has more complicated inter-relationships. In the case of programs, the structure provided by a programming language cannot direct the creation of information. In DBMS, as long as the data base structure is determined (say, hierarchically), the storing of data such as employee records or inventory merchandise is much easier than the writing of a program to accomplish the same task. This means that how to create the information becomes more important in PDS and SDS.

Unlike AIS, automatically generating programs or software for the user is never the intention of PDS or SDS. Even though it is possible that, in the future, a user may only have to feed the specification of a program into a system to obtain the program code in some language, the user still has to create the specification. Since the probability of making errors or changes is always present, the user has to analyze the information in order to make appropriate corrections or modifications. Furthermore, the user still has to be involved in the maintenance and enhancement of existing programs and software. This means that many analyses will still have to be done by the user and cannot be fully automated.

The above contrast might mislead one into thinking that PDS and
SDS are isolated from DBMS and AIS. On the contrary, one should conclude that a satisfiable PDS or SDS must utilize techniques used in both DBMS and AIS. Therefore, the approach proposed in this thesis for achieving such an SDS is the methodology-driven software development system. Methodologies which possess the knowledge of how to create software information are similar to the knowledge databases in AIS but only in a partially automatic manner. Powerful information manipulation functions which retrieve meaningful, methodology-oriented chunks of information are similar to queries in DBMS.

Therefore, as an overall evaluation, the most important thing that the LISP-prototype project demonstrated (or attempted to demonstrate) was the feasibility of a methodology-driven system which incorporates techniques in Data Base Management Systems and Artificial Intelligence Systems into a Software Development System.

4.4. Other Important Experiences

4.4.1. Prototypes Should Last

As was mentioned in the beginning of this chapter, there are many design decisions which cannot be resolved unless a working system is available. The prototyping approach is an effective technique for resolving such problems.
There are two related purposes behind the design of a new software environment. They are the realization and the enhancement of the current state of the art in software development support. The need for realization is obvious since the software development process needs more support. The need for enhancement is also obvious since software environments cannot survive without being influenced by the developments produced in other areas -- hardware, languages, data bases, etc. Therefore, one can consider that the existing software and programming environments are still in the prototype stage. However, a good prototype must contribute the utmost to its next generation rather than merely give a positive or negative evaluation of some design decision. In other words, a good prototype should be the realization of an evolvable design. The value of such a prototype is thus multiplied as it can serve as many prototypes during its lifetime. This is one of the important lessons we learned from the LISP-prototype.

The mistake we made was that the prototype could not evolve in the direction its own feedback suggested. Though it was quickly implemented (probably due to too strict an interpretation of the term "rapid prototyping"), it only fulfilled its minimum requirements before failing us. However, the second prototype, which will be discussed in Chapter Seven, is carefully implemented and thus much more robust and flexible. This new prototype is
expected to be easily extended, modified, and re-configured as the research progresses. It is also expected that this prototype will evolve into a high performance working system, and actually be used to create the next prototype.

4.4.2. LISP and FRL as Prototyping Languages

Though LISP and FRL were disqualified from being the implementation languages of our final system, we found that they have many attractive features of good prototyping languages.

First of all, the simple syntax of LISP and FRL (which is an extension of LISP, providing more sophisticated list structures called Frames and operations to manipulate Frames) makes it easy to learn these languages and write programs in them. All of the necessary language constructs, such as function invocations, recursions, iterations, cases, and block structures, are available within their simple syntax. The uniform treatment of data, functions, and programs eliminates type declarations, type checking, etc. This was convenient when debugging functions because we did not have to worry about type consistency or global variables. Immediate execution through the interpreter also sped up the debugging process without the annoying delays of re-compilation.

The powerful list structure, which can easily implement other
useful data structures such as records, stacks, trees, and certainly lists, is no worse than other multiple type languages. The dynamic storage allocation facility which allowed lists to grow when necessary, was particularly handy in the implementation of our refinement trees. The Frame structure, which is also a recursive list structure and is used in AI systems to store knowledge, was also useful for the implementation of symbols in our concept grammar.

However, this section is not intended to be a testimonial for LISP and FRL. Many features mentioned above could be considered drawbacks according to different programming tastes. In spite of the disadvantages of LISP and FRL mentioned earlier in the evaluation, from an implementor's point of view, LISP and FRL are fairly desirable languages for "rapid" prototype implementation.

4.4.3. Human-Engineering to Support Methodologies

As CPS and other systems have demonstrated, a syntax-directed program editor can release programmers from the nightmare of missed semi-colons, awkward indentation, and un-matched parentheses. A programmer can see exactly what he/she is doing on the screen, and can suppress whole chunks of program code and replace them by ellipses. Errors detected are highlighted at the very positions they occurred rather than in a detached error list. These features
make a programming environment pleasant for its users. The most outstanding feature, though a simple one, is the use of syntax templates in these syntax-directed program editors. Since programmers do not really think of programs as strings of characters mixed in with keywords, templates of the language constructs provide the programmers with a more tangible way to think of programs.

Similarly, it is important to human-engineer software environments, in addition to what has been done in programming environments, such that methodologies can also be presented to users in a way that is close to how they think of methodologies. Though productions in a grammar form or an attribute grammar form are used for both structuring information and encoding concepts, they are not necessarily the most intuitive or effective form in which to present a methodology. More specifically, some productions are too complicated to be handled at one time, while other productions are too trivial and tedious. For example, in order to present the concept of updating the master file when the tkey and mkey are not matched, consider the following productions:

\[
\begin{align*}
\text{<no match> := <while loop>} & \\
\text{<while loop> := WHILE <check control variable>} & \text{ DO <no match process> <update control variable> OD} \\
\text{<check control variable> := <same transaction key>} & \\
\text{<update control variable> := <read next transaction record>} & 
\end{align*}
\]
Each construct is apparently too trivial to be a meaningful concept. Instead, we construct a concept tree from these productions and present it as a whole to users in the following form:

```plaintext
no match
    WHILE same transaction key
    DO no match process:
        read next transaction record:
    OD
```

On the other hand, if a production requires more detailed thinking and the interpretation information is to be large, it can be presented to the user in several forms. Each of these forms only emphasizes an important sub-concept (which might overlap others), and is thus more meaningful to the user.

As discussed earlier, the internal representation of the productions and refinement trees was not suitable for the presentation of the concepts and information collected. The discussion pertaining to the presentation of a meaningful chunk of concept as a whole later led us to human-engineer our environment using a form-based approach. The detail of this form-based approach is discussed in the next chapter.
5. EXPERIENCE WITH FORM-BASED INTERFACE
TO ENFORCE METHODOLOGIES: CASE STUDIES

The great form has no form.

-- Lao Tsu

Form follows functions.

-- Slogan of the Bauhaus school of architecture

As we learned from the LISP-prototype, human-engineering should be one of the top priorities of system design. The LISP-prototype was hard to use because:

- it required that the user deal with too much unnecessary detail,
- its rudimentary user interface could not enforce methodologies effectively,
- it had poor mistake-recovery support,
- it did not have adequate display capability,
- its help information was inadequate,
- it was difficult to modify a methodology description,
- it was slow.

In an integrated environment, the issue of human-engineering should not be separated out from the rest of system design. An architect should have a vision of the human factors related to the users of the building when he/she is designing a building.
Similarly, instead of attacking each problem individually, the form-based approach was proposed to human-engineering the entire environment.

In an interactive, human-engineered, software environment, at least the following features are required: good display capability, freedom from managing unnecessary details, powerful high level tools, and a sophisticated help system. Each of these features is achieved by one or several components in the TRIAD environment. More importantly, their design is based on the notion of "forms". From the user's point of view, "forms" are chosen as a representation of information collected by the environment. Such a representation of information is high level, human-oriented, and flexible. It is our experience that, as soon as the notion of "forms" emerged, the re-design of each component and the integration of different components immediately crystallized. Our thoughts are no longer restricted to low level design vocabulary such as text strings, lists, addresses, trees, symbols, and productions.

It is also our experience that this form-based approach permits us to design and enforce methodologies more effectively in both industrial and classroom situations. Two case studies of enforcing methodologies using this form-based approach are presented in this chapter. The first case study discusses the design and enforcement
of a Data Abstraction Based Methodology in a classroom situation, while the second one discusses this in an industrial environment.¹

The properties of forms and their mechanisms to enforce methodologies are first discussed in Section 5.1. PART-A discusses the first case study, where the design of a Data Abstraction Based Methodology using forms is given in Section 5.2, the usefulness of forms for standardizing design and other advantages are given in Section 5.3, and the summary is in Section 5.4. PART-B discusses the second case study, where the problems of software development in industrial environments are first discussed. The analyses and the solution approaches to these problems are then discussed.

5.1. Aspects of Forms

Forms are widely used to present a set of relevant information and to elicit standard responses from the respondents. Examples are tax forms, application forms and various questionnaires. Many existing "friendly" environments use "menus" to present the user with the minimal information necessary to make simple decisions. These menus are also forms that display all the possible choices to ensure that the users' decisions are well-informed. The term "form"

¹This part of the work was done during consultation with two industrial companies at Ohio and California, and the details are not reproduced here. Only the general concepts are discussed.
is used here in its most general sense. It refers to a mechanism (or "mold") for gathering project-related information in a controlled fashion and structuring it according to certain standards and methods. A typical form is illustrated in Figure 8a. The form contains organizers which generally describe what is to be filled in the blanks. The annotations underneath each blank provide information on how to fill in the blank. We shall say that the annotations provide help information. The text used to fill in a blank will be called an entry. A form does not have to be restricted to English language text. Figure 9 is a blank form which uses graphical symbols. The filled-in graph for a specific problem is in Figure 10.
### DESK COPY REQUEST FORM

**To**

(name of publisher)

(street) (city & state) (zip code)

*Your book,* 

(specify author, title & edition)

publishers book number **34-13308**

is being considered as a

(required or recommended)

text in my course

(description of the course)

with an estimated enrollment of ______. I would appreciate receiving a desk copy as soon as possible.

**Name** ___________________________

**Department** ________________ **University** ________________

**City** ________________ **State** _____ **Zip Code** __________

---

### PROGRAM DESIGN FORM

**Program input specification:**

**Program output specification:**

**Visible lower level modules:**

(describe each lower level module)

**Program driver:**

(give the pseudo code. The pseudo code must refer to all visible lower level modules listed above)

---

**Program-oriented organizer**

**Human-oriented organizer**

---

Figure 8a: Form

Figure 8b: A Program Design Form

Figure 8: Forms
Figure 9: Form for a Transaction Processing Problem Requiring Update of a Master File Using Transaction Records from a Transaction File

Figure 10: Filled in Form
In TRIAD, a useful concept tree derived from the underlying concept grammar is represented as a blank form, but only meaningful and important symbols in the concept tree are reflected in the blank form as organizers. It is also possible that several blank forms may be used to represent a concept tree, or several concept trees may be represented as a single blank form. Therefore, a blank form only presents a reasonable amount of information to the user at one time.

A blank form must be filled in using some notation. This notation is used to record all software-related information specific to the problem at hand. The notation may be formal or informal, programmer-oriented or programming language oriented. If the notation is somewhat formal, it can be automatically analyzed by one of the tools in the tool-box, in a manner similar to that used in CPS.

The organizers help organize and structure the information collected. The help information associated with each organizer can suggest the content and notation (often context sensitive) of the information which needs to be filled in the blanks. Some of the organizers may require machine-oriented detail in some programming language. The organizer program driver, in the form given in Figure 8b, is an example. Other organizers may require human-oriented management and design decisions to be recorded. An example, in the
same form 8b, is the organizer visible lower level modules. Note that this organizer asks the programmer to document information relevant to the program driver. Thus, organizers dictate the nature of entries to be filled in as management decisions, requirement specifications, design specifications or pseudo code, etc.

Figure 8b also illustrates how a form may have documentation judiciously embedded in programming language detail. The organizer visible lower level modules requires the programmer to think about contextual information necessary for implementing the program driver and to decide on the list of modules necessary for the driver. This decision is recorded under the organizer visible lower level modules. This is exactly the information a program maintainer must have immediately before modifying the program driver.

When the information entered for an organizer needs to be further refined, a separate form is used. This process is called the refinement of an organizer. The concept of "refinement" and its dual concept "abstraction" have been recognized as fundamental techniques of solving complex problems. A complex problem is usually abstracted to a more understandable, simplified version by suppressing irrelevant details. This process of abstraction continues until a solution of the abstract problem is available. One can then refine the solution in terms of more specific details to solve his original
problem. In general, the process of abstraction (or theorization) is more difficult than that of refinement (or implementation). Students in computer science are usually asked to write a lexical analyzer using the finite state machine model but not the other way around.

A set of blank forms of a methodology is designed to play the role of conceptual model in order to guide the users' thought processes. A user simply uses the conceptual model provided in a methodology and refines each abstract concept to solve a problem. Each organizer may have several blank forms to refine it, and this set of blank forms is indicated at the left-hand side of the organizer within a pair of curly brackets. The help information associated with each one explains its usage to help the user select an appropriate one. Some organizers need not be refined; these are called the terminals. Some organizers can only be refined using a blank form; in that case, the refinement process can be automated by the system.

To properly design a set of blank forms for enforcing a methodology we must consider

- the **semantic content** of the organizers,
- the **placement** of the organizers,
- the **relationship** between forms,
- the **ordering** of refinement, and
- the content of help information.

Figure 11 summarizes how some of the mechanisms of the form-based approach are used to achieve various effects.

<table>
<thead>
<tr>
<th>Form Mechanism</th>
<th>Purpose of the mechanism</th>
</tr>
</thead>
<tbody>
<tr>
<td>content of human-oriented organizers and its help information</td>
<td>standardizes content and notation of documentation</td>
</tr>
<tr>
<td>placement of human-oriented organizers</td>
<td>standardizes placement of documentation</td>
</tr>
<tr>
<td>placement of program-oriented organizers</td>
<td>standardizes the structure of programs</td>
</tr>
<tr>
<td>content of program-oriented organizer and its help information</td>
<td>standardizes the content and notation of program-oriented text</td>
</tr>
<tr>
<td>the set of blank forms to refine an organizer</td>
<td>standardizes the information structure by enforcing refinement ordering</td>
</tr>
</tbody>
</table>

Figure 11: Form Mechanisms
PART-A: Case Study I

5.2. Design of Forms for a Data Abstraction Based Methodology

In order to illustrate how to enforce a methodology using the form-based approach, we shall first present a set of blank forms which provide concepts of how to specify data abstractions. We would like to point out that the purpose of the discussion here is to emphasize how to use the set of blank forms to enforce a data abstraction based methodology rather than to describe what a data abstraction is. The reasons for choosing the methodology for specifying data abstractions are as follows:

- The techniques of data abstractions are useful in both programming and database design. We would like to design such a methodology which can be used by students to develop medium scale programs in order to obtain programmers' feedback.

- Though there are a number of specification techniques available in published literature (for a comprehensive bibliography see [Dung80]), their emphasis is on "what" a data abstraction is and very little on "how" to apply them.

- There are many new programming languages which support data abstractions, but they can not guide the programmer's thought processes. We would like to demonstrate that the methodology designed using forms can provide better support to programmers than can the programming languages.

This section is divided into three subsections. A set of blank forms to enforce a Data Abstraction Based Methodology and their help forms are presented in Section 5.2.1. In Section 5.2.2, the rationale for the design of these forms is discussed. Section 5.2.3
then discusses how this methodology is synthesized from other existing methodologies.

5.2.1. A Data Abstraction Based Methodology

Figure 12: Blank Forms of a Data Abstraction Based Methodology
Figure 12 (continued)

**Figure 12: Blank Forms of a Data Abstraction Based Methodology**
This is a tutorial for using the eleven blank forms in order to apply a data abstraction based methodology. This tutorial will instruct the user how to fill in the blank forms using the following help forms.

The user can fill out this forms in various orders. However, FORM-1 is the suggested starting form. The information given in the help form are designed to provide instructions, suggestions, check-list, or notation required to fill in each blank. The user can choose to refine a non-terminal organizer in a form using other blank forms whose form numbers are indicated at the left side of the organizer. By refining organizers using other blank forms the user will obtain a form tree. In order to keep track of the structure of your form tree, each filled form is assigned a unique form-use-number. The form-use-number of the root (i.e. the form containing program name) is assumed to be 0. If an organizer in a form needs to be further refined, the form-use-number of the refining form can be obtained by adding a numerical suffix to its parent form-use-number. This form-use-number should be consistent with the number used in the entry.

As the user fills in the forms, he/she will be asked to identify what is "visible" to its top level. This notion of "layers" can reduce side effects and promote security of different modules. The term "module" is used to refer loosely to either a function, a procedure, or a data abstraction. Since the methodology is data abstraction based, the "kernel" for the program will contain a variety of data abstractions.

---

1 This is the tutorial actually used by graduate students and is given verbatim.

2 This fundamental notion underlies many of the newer programming languages such as Ada.
The Help Forms for Data Abstraction Based Methodology

**HELP FORM-1**

<table>
<thead>
<tr>
<th>Form-use-number</th>
<th></th>
</tr>
</thead>
</table>

**program name:**

**program input description:** List the input objects (nouns) to be processed by the program and describe them briefly.

**program output description:** List the output objects (nouns) result of the program and describe them briefly.

(2,4) **visible lower level module description(s):** Write down all verbs (describing function modules) and the lists of the form [noun + verbs] (describing data abstraction modules) which are to be used in the program driver below. That is, list all the module names visible to the program driver.

(10) **program driver:** Specify the main program using pseudo code.

The driver should use only the nouns specified in the input and output descriptions and the verbs and nouns specified in the above entry.

---

**HELP FORM-2**

<table>
<thead>
<tr>
<th>Form-use-number</th>
<th></th>
</tr>
</thead>
</table>

**visible lower level module description(s)**

(2,4) **visible lower level module description(s):** List the nouns (More?) or verbs which will be used by one of the function modules in the function module description below.

(3) **function module description:** You can have several function (More?) modules. Each module is described by a verb followed by a brief description. The verb should be the same as those listed in the entry above. Each module description should be preceded by a form-use-number. This entry is mainly for documentation purpose and is optional.
**HELP FORM-3**

<table>
<thead>
<tr>
<th>Function Module Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input Description: List input nouns of this module with brief description.</td>
</tr>
<tr>
<td>Output Description: List output nouns of this module with brief description.</td>
</tr>
<tr>
<td>Interface Specification: List the external nouns and verbs, i.e., the nouns and verbs listed in the visible lower level module description of the parent form used in this function module.</td>
</tr>
</tbody>
</table>

- Local Data Type(s)/Module(s): List the verbs (function modules) and nouns (data abstractions) used only within this module.

- Function Specification: Give a brief specification of the module function in pseudo code.

**HELP FORM-4**

<table>
<thead>
<tr>
<th>Visible Lower Level Module Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple Type(s): List the primitive data types available in your intended implementation language.</td>
</tr>
</tbody>
</table>

- Data Abstraction: This form is used to describe the data abstraction in the kernel. List the name (noun) the form-use-number of each data abstraction.

**HELP FORM-5**

<table>
<thead>
<tr>
<th>Local Data Type(s)/Module(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple Type(s): List all the local primitive data types that are available in your implementation language.</td>
</tr>
</tbody>
</table>

- Data Abstractions(s): List the nouns describing the local data abstractions of the module.

- Function Module Description(s): List the verbs describing all the local function modules of the module.
HELP FORM-6

Form-use-number[ ]

data abstraction: specify the name (noun) of the data abstraction.

(7) lower level data abstraction(s): List all the nouns which [ ] (More?) will be used to define this data abstraction. These nouns should include all the types used in the I/O of all the operations defined below.

(9) operation(s): List all the useful but basic operations [ ] (More?) (verbs) of this data abstraction (like constructor operations, accessing operations, modifying operations). Operations which are too specific or problem dependent and not generally useful should be avoided. Function modules can be used for problem dependent code.

(8) canonical form: First identify those operations which can [ ] be used to construct any instance of this data abstraction, then write down the expressions using these operations and their parameters. For example, the canonical form of a STACK is either stack=CREATE, or stack = PUSH(...PUSH(PUSH(CREATE,a),b),...),c).
PUSH & CREATE are called constructor operations.

HELP FORM-7

Form-use-number[ ]

lower level data abstraction(s)

simple type(s): List the primitive data types available in your implementation language such as INTEGER, REAL, BOOLEAN, ARRAY, CHARACTER, STRING, etc.

(6) data abstraction(s): List the noun which will be designed [ ] (More?) as data abstractions again. This heading allows a hierarchical (or recursive) structure of data abstractions.

HELP FORM-8

Form-use-number[ ]

canonical form

canonical form

canonical form

canonical form

canonical form

concrete representation: Write down the code in the chosen programming language to represent the canonical form. This is not the actual implementation of the data abstraction. It is a mapping from the canonical form to the concrete structure in the chosen language in order to guide the actual implementation of each operation. For example, if you implement a stack using an array STK(n), then the concrete representation will look like stack = {STKPTR=0} or stack = {STK(1)=a,STK(2)=b,...,STK(t)=c; STKPTR=t}. 
HELP FORM-9  Form-use-number[ ]

operation

input type —> output type: Specify the input and output
type(s) of the operation in the format
I1 x I2 x...x Il —> 01 x 02 x ... x Oj

semantic equations: Specified the effect of applying this
operation to the canonical forms. In the example of
STACK, the semantic equations of the operation POP
are: POP(CREATE()) = ERROR;
POP(PUSH(c.f.)) = c.f., where c.f. is any
canonical form of STACK.

operation implementation: Write down the implementation code
for the operation in the chosen programming
language. Each operation should be implemented
as a re-entrant procedure.

HELP FORM-10  Form-use-number[ ]

program driver

program driver implementation: Write down the implementation
code for the program driver (the main program)
specified in Form-1.

HELP FORM-11  Form-use-number[ ]

function specification

function implementation: Write down the implementation code
for the function specification given in the
parent form.
5.2.2. Rationale for the Design of Forms

The general issues related to the design of forms were discussed in Section 5.1. Some specific ways in which issues have been addressed in the design of forms for the data abstraction based methodology are discussed below:

1. **Layered collection of modules**: The overall structure of the resulting program is a layered collection of modules with a kernel of data abstractions. Form-0 can be filled out at any stage of design to obtain a visual overview of the layered program structure. The layers can usefully embody security principles and were therefore incorporated into the overall architecture. This structure has been used effectively to construct parts of operating systems [Madn74], compilers and text editors. Such layer structure is enforced by the organizer **visible lower level modules** appearing in Form-1, -2, and the organizer **lower level data abstractions** appearing in Form-7.

2. **Make relevant documentation visible and hide irrelevant detail**: During the early stages of program design, decisions have to be made regarding the different modules as well as the attributes of each module which must be visible or accessible to other program modules. The organizer
visible lower level modules, in forms-1 and -2, are designed to document visible modules that are used by other modules, as indicated in the remaining entries in each of the forms. For example, the help information of the program driver in Form-1 says that it must use all the visible modules. The placement of the visible lower level modules next to the driver in Form-1, as well as the instructions in the help form, are significant. The instructions in the help form first encourage the programmer to identify contextual information (in this case the names of the visible modules) and then require the programmer to use all of this information in the program driver. It should be easy to see that much of the proper identification and use of contextual information can be checked automatically. The organizer visible lower level modules also requires documentation to be inserted near the program fragment (that is, the program driver) to which this documentation is relevant.

3. Highlight relevant documentation: The help information can be designed to direct the programmer or maintainer to re-examine relevant contextual information. For example, lower level data abstractions are highlighted before designing the operations of a data abstraction. The heading visible lower level modules is designed to document modules
which are visible and which should be used when filling out
other entries in the form. For example, the program driver
in the form must use all the visible module names.

4. Specific guidelines for specifying data abstractions: The
substance of this methodology is, in the design of Form-6,
-7, -8 and -9, to enforce the data abstraction techniques.
The help information of canonical form encourages the
programmer to first identify the "constructor" operations
and then to use them to specify the abstract data objects.
An example is also given in the help information to help the
programmers. Help information also suggests different kinds
of operations (like create, fetch, update, traverse, delete
operations, etc.) which may be useful for any data types in
general. In Form-9, the help information of semantic
equations also provides a very specific instruction about
how to specify the semantic equations of each operation.
Generally speaking, this is the most difficult part of the
specification of a data abstraction. By simply specifying
the effects of applying each operation on all canonical
forms, using the suggested algebraic expressions, the
semantics of each operation is fully captured. These
semantic equations later are important for verifying the
implementation of each operation. Examples are again used in
the help information. In Form-8, some useful instructions are given to help the programmer implement a data abstraction. Note that contextual information plays an important role here. For example, the help information of lower level data abstractions indicates that all the lower level data abstractions should be only the data types used in the operations defined below, and similar information is given in the help information of operations. By iteratively checking the information entered with the instructions provided in the help forms, the programmer should be able to obtain a correct specification of a data abstraction.¹

One should note that most of the organizers are not at all programming language-oriented. Form-6 through Form-9 are to enforce the technique of algebraic specification of data abstractions. The blank forms together with the help forms actually guide the programmers to specify a data abstraction in a systematic way to obtain a correct specification. High level programming languages force the programmers to design a data abstraction only in terms of programming language constructs, while the form-based methodology allows the programmers to design a data abstraction from a high

¹The tool Analyzer, which is not yet implemented, is designed to perform such consistency checking for the user.
level concept to the low level details through a systematic refinement process.

5.2.3. Relationship to existing methodologies and Languages

Tools and methodologies are two complementary ways of supporting the software life-cycle. As methodologies become more precise, tools can be developed to support the methodology. For example, the Ada [DoD80] compiler is a tool for supporting the data abstraction methodology.

Both the programming languages and methodologies have syntactic concepts which may be presented to the programmer by means of forms. In the case of a language-oriented form (or templates), the notation used to fill in the form must be analyzable by a compiler or an interpreter. The CPS [Deme81] and ALOE [Medi82] are examples of such an approach. These forms can be added to the methodology and the programmer can then continue to refine a program all the way down to the implementation code. Currently, the human-oriented information serves only to document design decisions and is not analyzable. As more formal notation is developed, or algorithms for analyzing design documents become available, tools can be developed and incorporated into the tool-box environment to analyze such information.
Methodologies attempt to provide more guidance for the programmer's thought processes than do programming languages. Methodologies suggest techniques for using the syntactic concepts in order to solve a particular problem. These suggestions are not, in general, automated by existing compilers. Certain environments based on Artificial Intelligence techniques attempt to provide more support for the programmer's thought process [Bars79]. Though such systems have potential, they have not yet been developed for large scale projects in different application domains.

The methodology, which is enforced using the forms given in section Section 5.2.1, is synthesized from several sources. The forms emphasize the early identification of module properties that are externally visible, as in Ada, MESA [Gesc77], and MODULA 2 [Wirt81]. The forms also provide support for the design of data abstractions using the algebraic specification techniques developed by [Gutt78], [Gogu78] and a theorem by the author. Finally, the forms enforce a layered structure which is more restricted than DREAM [Ridd80] and, thus, provide techniques for programming-in-the-large. This new, synthesized methodology is the result of our own experience in using existing methodologies for developing software projects.
5.3. Usefulness of Forms for Standardizing Design

In addition to the discussion above, two other major conclusions were reached based on the actual use of the forms for the data abstraction based methodology. Firstly, the forms standardized program structure, and secondly, the forms standardized documentation.

Twenty-eight graduate students were asked to provide very detailed designs for the wordbag problem [Lisk77a] (Figure 13). The students were divided into three groups —

Group I (15 students) used forms for the design.
Group II (6 students) used data abstractions without forms.
Group III (7 students) used no specific methodology.

The wordbag problem was chosen for our first attempt at getting data on the use of forms. It has a fairly straightforward solution, yet it is non-trivial and was new to the students. Also, the problem naturally requires the use of data abstractions.
**Wordbag Problem**

**Input:**
Given some document, we wish to compute, for each distinct word in the document, the number of times the word occurs and its frequency of occurrence as a percentage of the total number of words. The document will be represented as a sequence of characters. A word is any nonempty sequence of alphabetic characters. Adjacent words are separated by one or more non-alphabetic characters such as spaces, punctuation, or newline characters. In recognizing distinct words, the difference between upper and lower case letters should be ignored.

**Output:**
The output is to be a sequence of characters, divided into lines. Successive lines should contain an alphabetical list of all the distinct words in the document, one word per line. Accompanying each word should be the total number of occurrences and the frequency of occurrence. For example:

<table>
<thead>
<tr>
<th>Word</th>
<th>Occurrences</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>2</td>
<td>3.509%</td>
</tr>
<tr>
<td>access</td>
<td>1</td>
<td>1.754%</td>
</tr>
<tr>
<td>and</td>
<td>2</td>
<td>3.509%</td>
</tr>
</tbody>
</table>

Figure 13: Wordbag Problem
5.3.1. Standardization of program structure

The techniques of specifying data abstractions were taught to students in Groups I and II. The number of forms we collected from the students in Group I ranged from 15 to 30 forms for each student. The programs we collected from the students in Group II averaged 5 to 6 pages long. However, the form tree structure precisely recorded the history of each program development. The layer structure and algebraic specification of data abstractions were explicitly reflected in the form trees developed by all students in Group I. All the form trees were very similar and only differed in the way that the students decomposed their modules. (This is because we did not provide any specific method for module decomposition.) In fact, we found many identical subtree structures in these form trees, especially at the data abstractions level. By investigating the form tree structure, the programs developed by students in Group I all fell into three major classes of program structure, while the programs from Groups II and III were all very different from each other. This provides good evidence that programs developed using the data abstraction based methodology had less variation in program structure, and thus were more standardized.

By taking advantage of this standardized program structure, we were able to understand portions of a form tree without
understanding the entire form tree, and this is impossible to do for programs developed by students in Groups II or III. We were also able to understand a program using its form tree in a few minutes, while it usually took twice or three times as long to understand a program from the other two groups. We consider standardization of program structure as crucial to program modification and maintenance. As the size and number of programs increase, it becomes more difficult to understand, modify, and reuse an existing program or software. In order to achieve this, standardizing program or software structure is necessary.

5.3.2. Standardization of documentation

The data collected is given in Table 1.
Table 1: Data for Documentation of Students' Designs for the Wordbag Problem

<table>
<thead>
<tr>
<th>PROGRAMMER</th>
<th>PROGRAM I/O</th>
<th>VLMLD</th>
<th>MAIN PROGRAM (PROGRAM DRIVER)</th>
<th>FUNCTION MODULE</th>
<th>DATA ABSTRACTION/STRUCTURE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>INPUT</td>
<td>OUTPUT</td>
<td>DESCRIPTION</td>
<td>I/O INTERFACE</td>
<td>DESCRIPTION</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>OPERATIONS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GROUP I</td>
<td>All 15</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.</td>
<td>English, on, (15,2)</td>
<td>English, on, (15,2)</td>
<td>English, in terms of noun and verb, on, (15,2)</td>
<td>High level pseudo code + English, on, (15,2)</td>
<td>Specification using pseudo code + English, on, (48,2)</td>
</tr>
<tr>
<td>2.</td>
<td>English, off, (1,1)</td>
<td>English, off, (1,1)</td>
<td>Pseudo code, off, (1,1)</td>
<td>English, off, (1,1)</td>
<td>Cluster (CLU), on, (3,0.2)</td>
</tr>
<tr>
<td>3.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.</td>
<td>English, off, (1,1)</td>
<td>English, off, (1,1)</td>
<td>English, on, (1,1)</td>
<td>English, off, (4,0.9)</td>
<td>English, off, (4,1)</td>
</tr>
<tr>
<td>5.</td>
<td>English, off, (1,1)</td>
<td>English, off, (1,1)</td>
<td>English, on, (1,1)</td>
<td>English, on, (2,1.3)</td>
<td>English, on, (5,1)</td>
</tr>
<tr>
<td>6.</td>
<td>English, on, (1,2)</td>
<td>English, on, (1,2)</td>
<td>English, on, (1,2)</td>
<td>English, on, (2,2)</td>
<td>English, on, (6,1)</td>
</tr>
<tr>
<td>GROUP II</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.</td>
<td>English, on, (1,1)</td>
<td>English, on, (1,1)</td>
<td>English, on, (1,1)</td>
<td>English, on, (1,1)</td>
<td>English, on, (1,1)</td>
</tr>
<tr>
<td>3.</td>
<td>English, on, (1,1)</td>
<td>English, on, (1,1)</td>
<td>English, on, (1,1)</td>
<td>English, on, (6,2)</td>
<td>English, on, (6,1.3)</td>
</tr>
<tr>
<td>4.</td>
<td>English, off, (1,2)</td>
<td>English, off, (1,2)</td>
<td>Flow chart, off, (1,2)</td>
<td>English, on, (6,2)</td>
<td>English, off, (1,1)</td>
</tr>
<tr>
<td>5.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6.</td>
<td>English, on, (1,2)</td>
<td>English, on, (1,2)</td>
<td>English, on, (1,2)</td>
<td>English, on, (6,2)</td>
<td>English, on, (1,2)</td>
</tr>
<tr>
<td>7.</td>
<td>English, on, (1,2)</td>
<td>English, on, (1,2)</td>
<td>English, on, (1,1)</td>
<td>English, on, (5,1.4)</td>
<td>English, on, (1,2)</td>
</tr>
</tbody>
</table>
The table entries focus on the placement, the notation and the content of documentation produced by each group of students. The rows in table 1 are divided into the three groups. The column headings reflect the major program "components" that should be documented.

The simple metric used for the placement of documentation reflects whether the documentation is "on the correct position" (i.e. on) or "off the correct position" (i.e. off). The correct position for a piece of documentation is immediately preceding the program component that it describes. The notation used by each student for documenting each program component is also identified. The notation used ranges from free form English (imprecise) to algebraic specification (analyzable). Finally, the content of the documentation for each program component is ranked as adequate(2), inadequate(1), or absent(0). Meaningful variables contributed to adequate documentation. On the other hand, over-documentation was considered inadequate.

For each student, a table entry reflects the placement, notation and content of documentation for each program component. (A blank entry indicates no documentation was provided.) For example, the entry "Pseudo code, off, (2,1)" for function modules would indicate that pseudo code was used to document function modules. The
pseudo code was not placed immediately before the function module implementation code, and hence, was placed "off the correct position". Finally, two function modules were used in the program as indicated by the first element in the ordered pair (2,1), and the documentation was ranked inadequate, on the average, as indicated by the second element.

There is only one row in Table 1 for Group I. This is because all the 15 designs turned in had standardized documentation as enforced by the forms. This standard is described using the single row. The students in Group III did not use data abstractions. Their documentation and use of data structures were examined instead.

1. Placement of Documentation: The programmer-oriented organizers were designed to force the programmers to document each important program component. The blank entries appearing in the sections of Groups II and III indicate the absence of documentation. The figures given below are the percentage of non-blank entries in table 1. These figures reflect the quantity of documentation present in the program turned in by each group.

<table>
<thead>
<tr>
<th>Group</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Group I</td>
<td>100%</td>
</tr>
<tr>
<td>Group II</td>
<td>46%</td>
</tr>
<tr>
<td>Group III</td>
<td>41%</td>
</tr>
</tbody>
</table>
The other significant placement measure is whether the documentation immediately preceded the code that it documented (indicated as on/off in the table). The following figures reflect the correctly placed documentation in each group.

<table>
<thead>
<tr>
<th>Group</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Group I</td>
<td>100%</td>
</tr>
<tr>
<td>Group II</td>
<td>56%</td>
</tr>
<tr>
<td>Group III</td>
<td>78%</td>
</tr>
</tbody>
</table>

2. Notation of Documentation: Most of the documentation in Groups II and III was in English and not automatically analyzable. On the other hand, the more restricted notations used by Group I to fill out the forms facilitated useful automatic support for program synthesis (for example, data type declarations and function declarations, which can be automatically generated) and for verification, etc. When English was used to document program components, the format of documentation was standardized and thus readily understandable.

3. Content of the Documentation: The average score achieved by each group (recall adequate(2), inadequate(1), absent(0)) is shown below:
Group I --- 2.00
Group II --- 0.98
Group III --- 1.15

This indicates that the help information provided in the help-forms actually guided the students in Group I not only on how to use the notation for an entry, but also on what to fill in as an entry in the blank form. Also, the use of forms resulted in a standard amount of documentation which adequately described all the program components. The content of documentation used in the forms was also uniform; however, we have not yet established the metric to accurately reflect this.
5.3.3. Other Comments

Another interesting conclusion drawn from the data collected is that the program structure can be substantially affected by using forms. Table 2 illustrates how Form-2 and Form-4 affected the programmers' design decisions about modularization.

Table 2: Number of Modules Used in the Program Designs.
An entry indicates the number of students using the number of modules given at the top of the column.

<table>
<thead>
<tr>
<th>Modules</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>Average</th>
<th>Standard deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Group I</td>
<td></td>
<td></td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>7</td>
<td>2</td>
<td></td>
<td>5.3</td>
<td>1.25</td>
</tr>
<tr>
<td>Group II</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3.5</td>
<td>1.21</td>
</tr>
<tr>
<td>Group III</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td>3.4</td>
<td>1.89</td>
</tr>
</tbody>
</table>

Form-2 and Form-4 encouraged the programmer to solve the problem in terms of smaller modules (function modules or data abstraction modules). As a result, the students in Group I used more modules in their program design. The statistics (Table 2) show that students in Group I used about two modules more than those in Groups II and III. This was about 50% more than those used by students in Group II and about 56% more than those used by Group III. This data indicates that organizers and help information together definitely changed the program structure for the given problem. That is,
program quality can be improved by carefully human-engineering the forms.

The notation, content, and placement metrics used reflect commonly accepted practices for evaluating software documentation. However, some of these metrics were, to an extent, subjective. For example, we had determined that the program components should be documented immediately prior to the actual program component. Every manager has preferred software engineering practices. The sample data collected above indicates that forms can be engineered to enforce these practices. Furthermore, as these practices evolve, the design of forms can evolve.
5.4. Summary of Case Study I

The form-based approach required an initial three hours overhead to teach the Group I participants the use of forms. However, considering that the forms enforced a data abstraction based methodology (and thus the designs had all the advantages), this overhead appears to be compensated by the reduced time needed to maintain the program. Based on the experience obtained from the "pre-experiment" described here, other controlled experiments on various form-based issues are currently under design.

Since well-designed methodologies generally provide more support for the programmer's thought process than well-designed programming languages, it is desirable that software engineering environments support both methodologies as well as programming languages.

The interface is based on forms — blank forms and help forms. Such an interface facilitates the standardized enforcement of methodologies (as well as programming languages). The forms must be carefully designed. The data abstraction based forms described in this case study illustrate the issues that must be considered. Carefully designed forms can alter programmer behavior in desirable ways. This is supported by actual data obtained from the documentation provided for a design problem. However, other
experiments must be designed and conducted in order to fully understand how forms can be human-engineered to impact positively on the software engineering process.
PART-B: Case Study II

This case study was done during consultation with two industrial companies in which computers were heavily used to automate many different levels of production processes.\(^1\) In one company, the main products are micro- or mini-computers with special-purpose software which can be used to accumulate data, control (sensor and actuate) processes in the assembly line, perform analysis on the data collected, and provide dynamic communication between managers and manufacturing managers for various kinds of manufacturers. In the other company, the major activity of the department in which we were consulting was to design and produce computer-aided hardware devices which were in turn used to perform various kinds of tests on other products (for example, a tester to test the durability, density, uniformity, etc. of disks).

In order to proceed with our discussion, we will first give a simplified description of the product development in these industrial environments. The problems of such product development and the analysis are discussed next. We will then present the solutions using the concept of form-based software factory.

\(^1\)The actual details related to the management, environments, technology, and resources of these two companies are not to be discussed here. Only very general issues related to software development will be addressed.
5.5. Scenario of Project Development in Industrial Environments

Typically, the main characters in an industrial environment are managers, engineers, technicians, programmers, and end-users. The main activity of the combination of the first four groups of characters is to produce certain products which meet the end-users' requirements. An important characteristic of such activity is that computer software is heavily used in both the design and manufacturing processes. Since we are only interested in the software development, we will define a software product to be any computer software, ignoring its actual use either by the customers of the products developed by the company or by a group of project members to improve the design or manufacturing activity. That is, the users of a software product can be the customers, managers, engineers, technicians, or even programmers themselves.

Usually, a project team is formed when a product design is initiated for a customer. The managers and engineers\(^1\) first specify the general requirements for creating the product. These may include cost/time constraints, resource allocation (including project members' assignments), a description of the high-level functionality of the product, etc. The assigned engineer, with help from the physicists, chemists, mechanics, electric/electronic engineers, etc.

\(^1\)Physicists, chemists, mechanics, electric/electronic engineers, etc.
technicians, specifies the design of the product. The engineer then passes on the design specifications to both the technicians and the programmers. The technicians provide the hardware device needed for this design, while the programmers provide the engineer the software needed. The engineer then uses these hardware and software to test out his design, makes necessary modifications, and gives it back to either technicians or programmers for necessary modifications. This cycle repeats until the product meets all the requirements.

From this simplified scenario of the product development in such industrial environments, the following characteristics are identified:

- there are many different groups of people involved, such as managers, engineers, programmers, technicians, and end-users;

- the production process requires intimate cooperation between these groups of people;

- the instability of requirements and design specifications is unavoidable due to lack of experience in designing new products; and

- engineers and programmers also need design tools and programming tools to improve their productivity.
5.6. The Problems

The current practice of such production development is surprisingly chaotic. There were some efforts made to improve the productivity, but they were done only on a limited local basis. In one company, the poor overall productivity has created intense pressure on the managers, as well as a very unpleasant working atmosphere among project members in different groups. The main objective of the analysis is to identify the root causes for problems typified by the complaints in Figure 14 and determine solutions to the problems.

5.7. Analysis of the Problems

After analyzing the data collected from three sets of questionnaires (for managers, engineers, and programmers, respectively) and consulting directly with many key members of each group, we discovered many interesting facts which, we believe, were the main causes of the problems. The analysis is discussed in the following two subsections according to engineers' and programmers' aspects.
<table>
<thead>
<tr>
<th>Engineer's complaints</th>
<th>Programmer's complaints</th>
<th>Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>Programmer talks computerese.</td>
<td>Engineer does not involve programmer in early design phase.</td>
<td>Programmers cannot pick up the engineer's terminology on-the-fly. Early involvement in the engineer's project will enable the requirements analyst to pick up engineer's terminology.</td>
</tr>
<tr>
<td>Programmer doesn't like to bring up prototypes.</td>
<td>Engineer keeps changing the specification.</td>
<td>To some extent the engineer cannot help this since the process details are constantly refined based on the engineer's intuition. Programmers, by and large, do not have adequate tool support for rapidly bringing up prototype software.</td>
</tr>
<tr>
<td>Support is too complex.</td>
<td>Engineer does not know what is needed.</td>
<td>Programmers, with experience, have learned that in the long run the engineers will need fairly powerful support even though the engineers do not realize this. Programmers, therefore, tend to put together powerful packages for the engineer. These packages are not always engineer-oriented.</td>
</tr>
</tbody>
</table>

Figure 14: Some Typical Comments on the Problems

5.7.1. Engineers' Aspects

1. Engineers must work in a "prototyping mode": Due to lack of experience in designing new products, the engineers' design process is complex. For each new product, various alternative approximations to the final design must be
attempted. During the prototyping phase, "rapid evaluation" of different alternatives is more important than the efficient performance of each alternative. Therefore, changes of the design specifications took place frequently. The system programmers who were responsible for maintaining the software could not keep up with the pace of such change. The engineers thus complained that their poor productivity was caused by the delays of programmers.

2. Engineers under-rate the complexity of software: Though engineers eventually learned about computer-related issues, they were not generally aware of the "science" underlying software design. They tended to under-rate the complexities involved in software engineering. This eventually created the misunderstanding between engineers and programmers. Thus far, the engineers have resisted attempts at standardizing the computer hardware/software that would facilitate the programmers' productivity. This was partly due to the fact that, in the past, the engineers have had unsatisfactory software support from the programmers, and partly due to the engineers being unaware that the complexity of software maintenance is even larger than that of software design.

3. Poor software support: From the engineers' point of view,
the software support provided by the programmers was either too complex or too unreliable to use. On the one hand, programmers tended to design complex software for long term use in order to reduce the future maintenance cost. But, these software were hard to use due to poor human-engineering, and unreliable due to the fact that the programmers themselves did not have sufficient programming tools support. Thus, many engineers decided to create and maintain their own software. Some engineers even decided to isolate themselves from the programmers, and the work of programmers was not at all rewarding.

4. Lack of communication: Engineers designed the processes and systems needed for the product; the programmers were only required to provide programming support for the computers that control the processes and systems. Rarely was the programmer involved in the early design decisions for the design of the processes and the systems. This eventually caused more difficulty for programmers to keep up with the frequency of changes in the specifications. Moreover, the engineer who designed the product might not be the engineer who designed the processes, and the programmer who was responsible for maintaining the software might not be the programmer who developed that software. Such transition
between different project members created many serious human-oriented problems that slowed down the entire project.

5. **Information management**: Experienced engineers agreed that there was about 80% functional commonality among different products they were designing. However, products with similar functionality were usually designed and implemented in totally different ways. For the new engineer or new programmer to perform effectively, there is a need for a central information bank of documentation. Currently, no such information bank exists.

### 5.7.2. Programmers' Aspects

1. **Use of methodologies**: Though most of the programmers agreed on the need to use methodologies, only 60% of the programmers actually used a methodology. However, the methodologies used by the programmers were too general to provide problem-specific support during development as well as maintenance. The methodologies currently used were applicable mainly in the detailed design phase, coding phase, and documentation, where the programmers spent about 40% of their time. No suitable methodology was used for remaining phases where the programmers spent 60% of their time. Programmers currently spent, on the average, about
30% of their time maintaining old code. No specific methodology was used (other than documenting the nature of a change as well as the date) to reduce this time. A well-designed methodology, used during program development, can reduce maintenance time since a methodology can reduce the number of possible solutions by identifying approaches that work and reducing the variations across programs.

2. **Unawareness of the State of the Art:** Various individuals and task forces did make a conscientious attempt to keep abreast of the state of the art. However, these task forces appeared to be focusing on only those productivity aids that have been successful in the 1975 - 1980 time frame. For example, languages and documentation standards were being considered. A database of project information (consisting of requirements documents, specifications, test plans, user manuals etc.) was not being considered. Newer, human-engineered productivity aids (such as customizable editors, customizable debuggers, UNIX-like operating systems, and incremental compilers) which have had a demonstrable impact on programmer productivity, are not being considered. Many of these aids, developed over the last few years, will be necessary to get a productivity edge in the 1982 - 1987 time frame. To illustrate this point by
analogy, a saving of 2:1 in the programmer effort is achieved when a programmer uses a screen editor as compared to a line editor [Robe79]. Noting that in an interactive system, most of the time is spent using the editor, an improvement in the design of the editor can drastically improve programmers' productivity. About 50% of the programmers either questioned the "existence" of these newer productivity aids, or were unaware of the usefulness of these aids.

3. **Experience in the Problem Domain:** There were a few programmers with a wealth of experience, developed over many years, in anticipating engineers' needs. But, no systematic effort was being made to exploit this experience. Experienced programmers also agreed that there was about 80% commonality among the software they developed, and they could at best re-use only 40% of the existing system code in the ideal case.

4. **Poor programming support:** The programmers had only rudimentary programming support, which seriously restricted their productivity. For example, programmers were still mainly writing programs in a pseudo assembly language. There was poor support for program development on their
mini-computers. Poor turnaround time (response time is sometimes as long as 8 - 10 minutes; batch jobs take a day) discouraged most of the programmers from using this support.

5. **Engineer-driven programming**: Since programmers supported engineers who worked in a prototyping mode, their programming specifications were engineer-driven. Engineer-driven programming involved adding enhancements to existing code. But, the programmers could not keep up with the pace of frequent changes of specifications. In other words, the programmers were working in a "fire-fighting mode". Programmers were usually not involved in the early design decisions with engineers and the specifications were usually not well-documented. Therefore, it became more difficult for the programmers to modify the software and meet the schedule.

6. **Information management**: There was no information management control. All project-related information was scattered in various media: engineers' memory, programmers' memory, minutes of meetings, scribbled pieces of papers, program documentation, code, and user manuals. This diversity caused many difficulties, such as inconsistent formats, lack of cross-referencing between related documents, and lack of
accessibility. The management of information should be automated so that it is easy to display or print out audit reports, management reports, and overview documents. Also, important information that reflects the experiences of various engineers and programmers should be collected in a standard, easy-to-use way.

5.7.3. Summary of the Analysis

It seems that the above analysis points to three major problems: communication problem, hardware/software support problem, and information management problem.

The communication problem has created wide gaps between managers, engineers, and programmers. Furthermore, it created gaps between different engineers and between different programmers. Since the productivity of engineers and programmers depended upon each other, these gaps eventually caused the entire project team to be paralyzed. Since engineers and programmers have to work at different levels in the product development, it is not possible, nor desirable, for an engineer to develop and maintain his own software or vice versa. It is, therefore, important to have an environment in which both the engineer and the programmer can work at an appropriate level and still communicate with each other through well-defined, standard terms (that is, neither engineer's jargon nor
computerese). In order to bridge the gap between engineers and programmers, a programmer/engineer is needed to create the requirements analysis document for the programmer.\(^1\) Since this programmer/engineer is skilled in bridging the terminology gap between the engineers and programmers, little time will be wasted by individual programmers in poor communication and imprecise requirements. This programmer/engineer, called requirements analyst, should also participate in the early tester design meetings of engineers. The requirements documents should reflect the fact that about 50% of the effort in a software project will be in continued development and in providing support to facilitate prototyping activities.

The hardware/software support problem also seriously decreased both the engineers' and the programmers' productivity. Such support provided by the programmers for engineers was resisted simply because it was not reliable and human-engineered. This was recursively due to the fact that programmers did not have good hardware/software support. Therefore, in the first place, the programmers should have sufficient programming tools in order to improve their productivity. For example, routines such as scanners, 

\(^1\)That is, specification of software support and maintenance plan for the lifetime of the project
display functions, translators, resource control, etc. made up 40% - 50% of the system code (as opposed to application code) written by programmers. Tools can be constructed to aid in writing this code quickly. Another category of tools can automate a lot of the bookkeeping that programmers must do in order to systematically apply methodologies. Only with the help of these tools, is it then possible for programmers to rapidly bring up prototype software for the engineers. In the meanwhile, engineers should also have tool support to specify, modify, or test a design.

Finally, the information management problem has created many serious problems in software maintenance. Since the project-related information was not recorded, or recorded but not structured, this information was practically inaccessible. Thus, many design and programming efforts were duplicated simply because it was harder to maintain the old code than generate new code. A fully integrated database for organizing all project-related information is necessary. This database should provide different views for managers, engineers, and programmers such that they can all access useful information easily. The information in the database should be structured in such a way that re-usable code or design patterns can be easily identified in order to improve the engineers' and programmers' productivity. The database should also facilitate new tools to support the engineer-driven programming (for example, a
tool which can determine the code affected by a change in a specification).

It can be seen that computer software development in industrial environments still remains in old-fashioned, family-styled industry. Individual efforts can not break through the traditional framework, and this is why the managers have to enter the fracas and seek ways to enterprise the entire development activity in order to improve the productivity and meet the market's demand. In the ideal situation, we believe that the engineers should design the products at fairly high level (as opposed to programmers). There is absolutely no need for an engineer to worry about the details of each process function, the architecture of the databases, or any system implementation details — just as an architect should only work on the blue-prints and not the nails and wires. It should be understood that an engineer is not forbidden to program, but that he/she should not be forced to do so.
5.8. Outline of the Solution Approaches

Referring to the previous systems analysis, we found that productivity problems have their roots in the engineer/programmer interface, the hardware/software tool support and the absence of project information management. These three major reasons are highlighted again in Figure 15.

**Software**

**Cause 2:** computer system not accessible to the engineer, not designed to evolve.

Solution approach 2: easy-to-use interface which makes system accessible and supports evolution.

**Cause 3:** specifications for software change constantly because engineer is working in prototyping mode.

Solution approach 3: improved programmer support environment which includes a prototyping tool-kit.

**Cause 1:** poor communication between programmer and engineer.

Solution approach 1: communication gap bridged by a requirements analyst.

**Engineer**
**Programmer**

Figure 15: Three Reasons for Poor Engineer/Programmer Productivity

In order to provide a solution to these problems, the following
general approaches are suggested:

1. Solution approaches to the communication problem:

   - **Standardization**: Both the information structure and development process should be standardized. That is, the engineers' product design specifications, and documentations, and the programmers' implementation specifications, modules, programs, and documentations, should all be standardized in terms of certain uniform representation such as forms. Moreover, the design development and program development should also be standardized. That is, engineers should identify some standard process of their design development based on their experience in different project domains, but still allow themselves to do prototyping design. Similarly, programmers should standardize the program development and maintenance process in different project domains. This will eliminate the intra-communication problem between different engineers or programmers. **Forms and methodologies** are suggested as mechanisms to achieve such standardization.

   - **Requirements Analyst**: In the beginning, a
requirements analyst who understands both engineering jargon and computerese should be provided between engineers and programmers in order to bridge the gap. This requirements analyst will enhance the standardization in both groups and eventually come up with a unified standardization for both engineers and programmers. This unified standardization, or engineer-driven software standardization, will alleviate the communication problem between engineers and programmers. A tool which can tune the forms and methodologies should be provided for the requirements analyst. However, when a new project evolves to an old project, this requirements analyst is no longer required.

2. Solution approaches to the hardware/software support problem:

- **Prototyping tool-kit:** A set of programming tools suitable for such engineer-driven programming should be provided. Tools which can create, manipulate, retrieve, and analyze all project-related information are necessary. These tools should also support re-usable modules such that a programmer can put together different modules to come up with the desired software support for the engineer. Such capability is extremely useful when programmers are working in a fire-fighting mode.

- **Design tools:** Programmers should provide design tools for engineers to carry out the design process at higher level. From the engineer’s point of view,
useful modules or systems are like black boxes. Thus, the engineer can use the design tools to reconfigure or rearrange these black boxes in order to design various prototypes.

3. Solution approaches to information management problem:

- **Structuring all project-related information:** All project related information — including information collected from different project members and at different phases of its life-cycle — should be well-structured. **Methodologies** for developing software in different project domains are again suggested as a model for such structure.

- **Highly accessible database:** The information thus structured should be made accessible to different project members in a very friendly manner. That is, this database should facilitate high level views and different views for different members. Tools which can retrieve meaningful chunks of information from this database are necessary. This database is a result of structuring information according to methodologies, and the foundation of better software tool support as discussed above.

Highlights of the cause and effect relationships between various factors that affect the productivity of programmers are given in Figure 16. Though a high level language will alleviate some of the programmers' problems, most of the problems stemming from the three major problems discussed earlier will remain. The **Software Factory Concept** is suggested (Figure 17) to remedy these problems.
Project/Problem Characteristics:
Push product technology to the limit
and iteratively refine the process
for developing the product

Engineer Characteristics:
Must work in a prototyping mode
<--------Engineer interface bottleneck
System programmer works in fire-fighting mode and poor environment
Schedule delay
Engineer creates own software
(or goes to vendor)
Software poorly designed/supported
Engineer demands help on poorly designed software

Figure 16: Cause and Effect between Factors that Affect Productivity
Project/Problem Characteristics:
Push technology to the limit and iteratively refine the process for developing the product

Engineer characteristics:
Must work in a prototyping mode
+ Use standardized hardware (computer, controller and interface)

Frequent changes in specifications
+ Communication (between engineer and programmer) which is
  . engineer oriented
  . aimed at anticipating changes in specifications and identifying commonality
  . standardized

System programmers
+ Good factory environment designed to
  . enforce well-designed, standardized methodologies
  . provide prototyping software support

Fine tuning of final system

Figure 17: Scope for Improvement
5.9. Details of the Solution Approaches — Form-Based Software Factory

An overview of the organization of a software factory is given in Figure 18. A Software Factory produces prototype software for the engineers according to the engineers' requirements. In an automated software factory, programmers use tools to improve productivity. These tools permit the programmer to assemble software which is customized for each engineer's needs by gluing together selected pre-existing modules. The resulting software meets the engineer's requirements and is also flexible because component modules can be quickly replaced as the engineer's needs evolve.

5.9.1. Form-Based Requirements Analysis

As indicated earlier, the Requirements Analyst will be critical in bridging the communication gap between the engineer and the programmer within the software factory (Figure 18). For each new project, the Requirements Analyst should identify resources needed as well as the details for
- initial development,
- development while the engineer is using the system, and
- continued support.

As the Requirements Analyst gains experience in various projects, he/she should reflect this experience in the design of blank forms.
Figure 18: Overview of a Software Factory
That is, the Requirements Analyst is responsible for "tuning" the forms to reflect experience in a tangible, systematic way. If experience is thus reflected, up to 90% of the information for the software design can be collected even before the engineer and Requirements Analyst meet to discuss the details of the remaining 10%.

Ideally, the Requirements Analyst should be an engineer-turned-skilled-programmer who is intimately familiar with the projects that need support. Such an engineer would be capable of identifying the characteristics of an engineer-oriented environment that would make the computer system more accessible.

When working with programmers, each engineer should design generic requirements documents (that is, forms and questionnaires), for the type of projects in which the engineer has experience. In other words, the engineer, together with the Requirements Analyst, should examine each type of project and

- carefully design questionnaires and forms that might be used subsequently by the Requirements Analyst in communicating with other engineers in similar projects,

- identify the details of an engineer/project-oriented interface, and

- anticipate the type of support engineers might need for their prototyping activities.
5.9.2. Form-Based Interface for Engineers

The relationship between the engineers' view and the programmers' view is illustrated in Figure 19.

The organizers in forms serve to structure information stored as entries. Structuring information is effective in permitting forms to evolve, and in making information accessible.

Any computer system, designed to provide automated support for evolving technologies, must itself be designed to evolve. Patterns (based on commonality) may evolve, from low-level to high-level, as a new technology becomes an older technology.

A low-level language obscures any pattern that may appear in programs. For example, programs written in pseudo assembly language are extremely difficult to comprehend. Comprehension is necessary before a pattern can be recognized, named as a high-level pattern, and then made accessible to the programmer or the engineer. Comments in programs make it easier to identify patterns since they are generally given in a high-level, English-like language. However, since documentation is not standardized, patterns are still difficult to identify.

In the form-based interface, organizers standardize and structure documentation. Thus it is easy to "see" patterns. Based on
Figure 19: Relationship between Engineer's and Programmer's View
experience with the problem, a perceptive programmer can add organizers (or remove them) to reflect changes in the old form organizers which form a pattern. Thus it is easy to effect evolution of the patterns. The new form can be used as a standard pattern until further tuning makes the form higher-level or more engineer-oriented.

5.9.3. Form-Based Software Support Environment

There can be differences of up to 100 to 1 [Broo77] in the programmer effort needed to implement a module. These differences can be a result of past experience, education, or variation in the use of methodologies. As a general rule, productivity gains are made by recognizing patterns in problems/solutions and by abstracting and exploiting these patterns when faced again with similar problems. In software development/maintenance, the task of recognizing patterns has been impossible because

- project information is not structured and organized, and

- no uniform repository of project information is currently used to make this information accessible.

Experience in the development and maintenance of software leads to the design of methodologies for different phases of the software engineering process. Such methodologies attempt to usefully support the programmer's thought process for re-creating only good, standard
patterns of programming without limiting creativity. However, methodologies, as they are generally used, are limited in their impact on software quality due to the following reasons:

- Methodologies are generally imprecisely described. As a result, programmers may interpret the same methodology in a variety of ways, thus creating very different programs for the same problem. This variation can drastically increase maintenance costs.

- Program documentation is not standardized. Even though a single methodology may in fact be used for a problem, each programmer documents design decisions in a unique way. This, coupled with the fact that documentation is not optimally placed in the program text, makes the program more obscure for the program maintainer.

- Few environments provide comprehensive, semi-automatic tool support oriented toward methodologies.

- Though methodologies evolve with experience, environments are not designed to provide tool support for methodologies as they evolve. Hence, the support provided by an environment for a methodology is often obsolete, and therefore, not optimal for saving programmer effort.

Forms are used here to address some of the problems with existing methodologies. The emphasis is on how to enforce a methodology rather than what a methodology is. The reasons itemized above can be addressed in the following way:

- Methodologies are precisely described by using blank forms and help forms. The blank forms enforce the syntactic concepts of a methodology and the help forms inform the programmer on how to fill in the blank forms. Forms are used to standardize the application of the methodology. The organizers in a blank form ensure that the programmer's design decisions are well-informed.

- The documentation of design decisions, as well as placement of documentation, is standardized. This is achieved by the blank forms, with careful designed organizers, which must be
filled in by the programmer.

- The standardized enforcement of a methodology using forms facilitates the design of software tool support.

- The forms can be tuned by expert programmers. That is, as programmers build up experience, they can change the forms (which they use for a project) to reflect this experience.

5.9.4. Automated Support for the Form-Based Approach

The programmer's task in the software factory is to design new tools. Routine jobs can be done with minimum effort using existing tools. Tools are designed to work on project information. All project-oriented information can be organized in a database by means of filled-in forms. The blank forms can be

- engineer-oriented, in which case they are designed to provide information (specific parameters, etc.) to the programs (also represented as forms) created by programmers;

- programmer-oriented, in which case they are designed to enforce certain methodologies as determined by appropriate committees;

- technician-oriented, in which case they have limited flexibility;

- manager-oriented, in which case they are designed to enforce management policies;

- requirements-oriented, in which case they are designed to elicit the engineers’ requirements.

The forms used for one project will not be suitable for another one. Therefore, many project-oriented sets of blank forms must be carefully developed by committees. Thus we now have two databases:

- Methodology Description Data Base which stores
project-oriented sets of **blank forms**, and

- Form Tree Data Base which stores **filled-in forms** for a project.

When a project is first authorized, the Requirements Analyst must fill out requirements forms. These are examined by the programmer to determine exactly what set of blank forms would be appropriate for the project. The selected forms should enforce a project-oriented methodology. These forms are filled in by the programmer to develop the software for the project. Engineer forms, technician forms, and management forms can also be created by the programmer to **selectively** display only critical information from the forms filled in by the programmer, and receive the responses.

Tools must be provided for the programmer (and the engineer) to manipulate forms and form trees. These tools should work on any set of forms and hence will be called **generic tools**. The list of important tools is given below:

- Keyboard monitor to interpret user's commands and properly activate routines in other tools.

- Information manager to fetch the appropriate set of blank forms and edit the entries to create filled-in forms. It should also provide useful displays of filled-in forms for engineers, managers, or programmers. For example, a single editing function (display status of project X) can print out all the information detailing when the project was started, the engineer who requested the project, etc.

- Analyzer to analyze the information collected (for example,
to check the consistency of related entries, determine side
effects of a change made in the specification, etc.).

- A program synthesizer to synthesize the program by taking
the program fragments in the forms and converting other form
entries to comments. This program is then sent to a
compiler.

- A form tuner to alter forms and create new ones in order to
reflect the experience gained during project development.

- Display module suited for our form-based approach.

- Database management system to organize the databases of sets
of forms and form trees developed by programmers, engineers
and managers.

The form-based approach permits useful methodologies, which are
optimal for a given project, to evolve. An expert programmer can
synthesize new methodologies by creating the appropriate forms,
based on experience gained from other similar projects. Thus sets of
forms can evolve to suit the exact needs of the engineers and the
given project in order to provide support during all phases of the
software-life-cycle. We shall refer to the new methodology as a
tuned methodology.

The form-based approach can be used in all phases of the
software life-cycle, ranging from the requirements phase to the
coding phase. For example, programming language forms for FORTRAN
can be added to an existing set of blank forms. The new set of
forms can then support the coding phase of a particular project. The
use of forms results in a uniform way of recording software-related
information. This uniformity facilitates more powerful tool support such as TRIAD/SIE mentioned before.

5.9.5. Design of Forms for Applying a Methodology

<table>
<thead>
<tr>
<th>PDM-Form-1</th>
<th>Project</th>
<th>PDM-Form-3</th>
<th>Project Design</th>
<th>Form-use-#</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project Name:</td>
<td></td>
<td>Architectural specification:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Project Description</td>
<td></td>
<td>(7) Functional Specification</td>
<td>Form-use-#</td>
<td></td>
</tr>
<tr>
<td>Project Goals:</td>
<td></td>
<td>Reusable Modules</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Project Manager:</td>
<td></td>
<td>Module name:</td>
<td>(More?):</td>
<td></td>
</tr>
<tr>
<td>Project Participants:</td>
<td></td>
<td>(More?):</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Project Proposal</td>
<td></td>
<td>Design Decisions:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Proposer:</td>
<td></td>
<td>(More?):</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Abstract:</td>
<td></td>
<td>Version Control</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(2) Proposal Details</td>
<td>Form-use-#</td>
<td>Version Number:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Project Review</td>
<td></td>
<td>Review</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reviewer:</td>
<td></td>
<td>Date:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Date:</td>
<td></td>
<td>Comments:</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PDM-Form-2</th>
<th>Proposal Details</th>
<th>Communique</th>
</tr>
</thead>
<tbody>
<tr>
<td>Communicate w/Manager:</td>
<td>Communicate:</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PDM-Form-4</th>
<th>Meeting</th>
<th>Form-use-#</th>
</tr>
</thead>
<tbody>
<tr>
<td>Date:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Attendees:</td>
<td></td>
<td>(More?):</td>
</tr>
<tr>
<td>(More?):</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Agenda:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Meeting report:</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Control Algorithm Specification**

<table>
<thead>
<tr>
<th>Control module name:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose:</td>
</tr>
</tbody>
</table>

**Function Module**

<table>
<thead>
<tr>
<th>Function module name: Form-use-#{ } (this will be copied from the parent form)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input specification:</td>
</tr>
<tr>
<td>Output specification:</td>
</tr>
<tr>
<td>Purpose of this module:</td>
</tr>
</tbody>
</table>

**Visible lower level modules**

<table>
<thead>
<tr>
<th>Function module name: Form-use-#{ }</th>
</tr>
</thead>
<tbody>
<tr>
<td>(More?):</td>
</tr>
<tr>
<td>Data abstractions name: Form-use-#{ }</td>
</tr>
<tr>
<td>(More?):</td>
</tr>
</tbody>
</table>

**Function module interface**

<table>
<thead>
<tr>
<th>Imported function: (More?):</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exported function: (More?):</td>
</tr>
</tbody>
</table>

**Module driver**

<table>
<thead>
<tr>
<th>Pseudo code:</th>
</tr>
</thead>
</table>

**Implementation Form-use-#{ }**

<table>
<thead>
<tr>
<th>Implementation Form-use-#{ }</th>
</tr>
</thead>
<tbody>
<tr>
<td>(More?):</td>
</tr>
</tbody>
</table>
PDM-Form-10  Data Abstraction Module  Form-use-φ[ ]
Data abstraction name (this will be copied from the parent form)

<table>
<thead>
<tr>
<th>Purpose of this module:</th>
</tr>
</thead>
</table>

Lower level data abstractions

(10,13) Data abstraction module name:
(More?):

Canonical form:

(14) Operation name: Form-use-φ[ ]
(More?):

PDM-Form-11  Implementation  Form-use-φ[ ]
Name: (will be copied from the parent for)

<table>
<thead>
<tr>
<th>Code:</th>
</tr>
</thead>
</table>

PDM-Form-12  Function  Form-use-φ[ ]
Function name: (this will be copied from)

<table>
<thead>
<tr>
<th>Purpose of this function:</th>
</tr>
</thead>
</table>

Input parameters:

Output parameters:

Side effects:

External functions used:

Pseudo code:

(11) Implementation Form-use-φ[ ]

PDM-Form-13  Data Abstraction  Form-use-φ[ ]
Simple Data Type  Form-use-φ[ ]
Data abstraction name (this will be copied from the parent form)

<table>
<thead>
<tr>
<th>Purpose of this data structure:</th>
</tr>
</thead>
</table>

Data structure:

PDM-Form-14  Operation  Form-use-φ[ ]
Operation name: (this will be copied from the parent form)

<table>
<thead>
<tr>
<th>Purpose of this operation:</th>
</tr>
</thead>
</table>

Input type:

Output Type:

Side effects:

Operations in lower level DA used:

Pseudo code:

(11) Implementation Form-use-φ[ ]
6. THE SOFTWARE INFORMATION EDITOR — TRIAD/SIE

The value of knowledge lies not in its accumulation, but in its utilization.

— E. Green

Men have become the tools of their tools.

— H. D. Thoreau

The Software Information Editor (or TRIAD/SIE) contained in the TRIAD Tool-Box is a vital component of the TRIAD environment. The structuring of information and enforcement of methodologies are actually performed by invoking functions defined in TRIAD/SIE. Moreover, maintaining the information structure and retrieving information from the databases also rely on the functions defined in TRIAD/SIE. Although it is described as an editor, its functionality far exceeds what this name implies. Briefly, TRIAD/SIE contains procedures which can be invoked by the commands in the User Interface to manipulate the information in the Form Tree Data Base, by the Display Module to fetch appropriate information from the databases for screen display, or by the Tuner to modify a methodology description. The procedures can also be invoked by the Tuner to tune the description of a methodology. More specifically, these procedures are defined using the operations in various data
abstractions (the databases) and are able to perform the following tasks:

- Text editing as in Emacs,
- Tree structure manipulation like creating, copying, removing, and inserting subtrees,
- Derivation tree refinement by using productions in the underlying grammar,
- Form tree refinement by using the user-selected blank form to refine a non-terminal organizer,
- Traversing the derivation and form trees,
- Retrieving hierarchical information from the Form Tree Data Base, and
- Retrieving relational information from the Form Tree Data Base.

The above capabilities allow TRIAD/SIE to manipulate text as a text editor, to manipulate tree structures as a structure editor, to organize forms and manipulate information in forms as a form manipulation system, and to create and retrieve information from the databases as an information manager. One should be reminded that the reason such a powerful tool is possible is mainly because the fact that the information collected is highly structured according to a methodology description which specifies the semantics behind the interpretation of each concept and the relationship between different concepts.

Existing programming environments represent the information
collected as syntax-trees or abstract syntax-trees which store only the information needed by the compiler to translate the programs into assembly code. In the TRIAD environment, the information structure is extended to include all software-related information through the life-cycle of a project, and it is useful for both the compiler and the users. Given such a rich structure for the information collected, it should come as no surprise that TRIAD/SIE can provide better support for software development than can a program editor.

In this chapter, TRIAD/SIE will be presented in terms of the functions it provides. In order to do this, the functions defined within TRIAD/SIE are divided into two classes. One class of functions is for information editing, called editing functions, and the other class is for information retrieval, called query functions. Note that these functions are all generic functions which are independent of any particular methodologies.

6.1. Editing Functions

The editing functions can be further divided into three sub-classes according to the objects which they manipulate. These three sub-classes are text editing, form editing and form tree editing.
6.1.1. Text editing functions

TRIAD/SIE provides the same power for text editing as that of Emacs. They can insert, delete, copy, search, and move the cursor at the level of characters, words, lines, or regions of text. These functions are mainly used to edit the information at the entry level, that is, to fill in the interpretations of organizers. They can also be used wherever and whenever text editing is needed. Therefore, these functions alone make TRIAD/SIE a complete text editor. We assume that readers are familiar with the Emacs editor; thus the details of these functions will not be discussed here.

6.1.2. Form editing functions

Given a form, which is either a newly instantiated form from a blank form or a partially-filled form re-visited, the form editing functions permit the user to move around between different organizers within the form and edit the entries using the text editing functions. From the viewpoint of editing, a form is a tree of nodes in which the text is stored. Recall that each form has its underlying concept tree structure which is displayed as form heading and organizers. As an exception to tuning a form, the concept tree structure is not allowed to be changed. These form editing functions manipulate the information within a form and they also make sure that the underlying structure remains consistent at all times.
A list of form editing functions is given below. Note that these editing functions are not the actual user commands; thus their names are not abbreviated and most of them require long argument lists. The actual editing commands, however, are as convenient as the text editing commands in Emacs.

**StartingForm(bf#)**  
Select and instantiate a blank form "bf#" as the root form of a new form tree.

**Refine(org,bf#)**  
Refine the organizer "org" using the blank form "bf#"; this "bf#" has to be one of the blank form numbers suggested by the organizer. The new form also becomes the current form.

**NextBlankEntry(org)**  
Find the next un-interpreted organizer after "org" in in-order manner.

**NextUnrefinedOrganizer(org)**  
Find the next non-terminal organizer which is not yet refined after "org".

**NextOrganizer(org)**  
Change the current organizer to the next organizer of "org" in in-order manner. The Display Module uses this function to determine the cursor position and to make sure that this organizer is visible on the screen.

**PreviousOrganizer(org)**  
Change the current organizer to the previous organizer in reversed order as that in NextOrganizer.

**More(org,n)**  
Generate n instances of the organizer "org". The "org" has to be a **repeatable** organizer.

**Choice(org,n)**  
Select the n-th organizer (and related organizers) from a set of **alternative** organizers.
DeleteEntry(org)
Delete the whole entry of the organizer "org"
(this can also be done using the text editing
functions).

EraseForm()
Delete the whole entries of all organizers in
this form.

SearchInForm(s)
Search the occurrences of string "s" in all
entries of the form.

CopyForm(fu#1,fu#2)
Copy the entries of form "fu#2" to form
"fu#1". These two forms must have the same
structure.

CopyEntry(org,text)
Copy the "text" into the entry of the
organizer "org". This "text" could be the
entry of another organizer or anything from a
text buffer (for example, a text file of some
documentation).

There are two parameters of these form editing functions, with
default values the current form tree and the current form, are
omitted here.

6.1.3. Form tree editing functions

While the previous two sets of functions deal with the
structures at the entry and form levels, this set of functions deals
with the structure at the form tree level. After an empty form tree
is created, a user can choose a starting blank form, edit it, refine
its non-terminal organizers, and continue this process to obtain a
form tree using only the previous two sets of editing functions.
However, the user can also use the form tree editing functions to insert, delete, clip, combine, and copy subtrees. In other words, the user can develop several form trees separately and then do pruning and grafting to obtain another form tree. Therefore, the development process is not restricted to only top-down refinement. However, these form editing functions would ensure that no such reforming violates the rules specified in the underlying concept grammar. A list of form tree editing functions is given below:

**CreateFormTree(name)**
Create an empty form tree and assign the "name" to it. The name of the methodology chosen by the user is also stored with the form tree.

**DeleteFormTree(name)**
Delete the named form tree from the users' database.

**VisitFormTree(name)**
Make the named form tree the current form tree.

**InsertFormTree(org, name)**
Insert the named form tree as the subtree of the organizer "org". This is useful for refining an organizer using an existing form tree.

**CopyFormTree(fu#, name)**
Copy the entire named form tree to the form "fu#". The original content of "fu#" is replaced by the root form of "name2". The effect of this function is equivalent to first deleting the subtree and then inserting the new form tree.

**DeleteForm(fu#)**
Delete the form "fu#" and all the forms in its subtree.
ClipFormTree(fu#, name)  
Make a new form tree from the subtree of the  
form "fu#" and assign the "name" to it.

VisitForm(fu#)  
Visit the form with the form-use-number  
"fu#". The cursor is placed at the entry of  
the first organizer.

NextUnfilledForm()  
Find the next form with blank entry.

NextUnrefinedForm()  
Find the next form which contains unrefined  
organizers.

NextForm(fu#)  
Change the current form to the next form of  
"fu#" in in-order manner.

PreviousForm(fu#)  
Change the current form to the previous form  
of "fu#" in the reversed order of that in  
NextForm.

ParentForm(fu#)  
Change the current form to the parent form of  
"fu#".

ChildForm(fu#, org)  
Change the current form to the form refining  
the organizer "org" in the form "fu#".

RootForm()  
Change the current form to the root form of  
the form tree.

SearchInFormTree(s)  
Search the occurrences of string "s" in the  
entire form tree.

EraseFormTree()  
Delete entries of all organizers in all forms  
of this form tree.

Together with the previous two sets of editing functions,  
TRIAD/SIE is as capable as a tree structure editor. Many useful  
applications, such as text processing and form-based systems, can be
easily implemented in the TRIAD environment.

6.2. Query Functions

From the database's point of view, the organization of the Form Tree Data Base has both hierarchical and relational structures. It is hierarchical because the tree structure is present in forms and form trees. It is also relational because the relations between organizers and forms are specified in terms of attributes and semantic functions. Thus we can take advantage of the presence of the hierarchy and relations to provide useful functions for information retrievals. However, the implementation of FTDB in the C-prototype is hierarchical.

Two types of information retrievals are considered here. One type of information retrieval is called fact retrieval which scans through the information collected and selects an answer. This answer is usually stored somewhere in the database (like documentation or values of variables), and the problem is to locate the answer through a certain path using a set of keys. The other type of information retrieval can be named deductive retrieval, which examines the logical relations among different pieces of information and prepares the answer from them. This answer is usually constructed dynamically based on the queries and the relations specified in the arguments. In general, deductive retrieval is
harder and less efficient than fact retrieval. In hierarchical databases, the fact retrieval queries can be made even more efficient. However, both types of information retrieval should be supported in an information management system. Readers will see that the query functions introduced later include both types of information retrieval.

Note that these two types of information retrieval can be supported by any kind of database implementation. Hierarchical structures lend themselves to rapid fact retrieval. Since the relations in hierarchical structures are bound at specification time, the search effort is reduced and thus the deductive retrieval is also more efficient. Relational database structures, on the other hand, are more flexible because the relations are bound at the query execution time. Though less efficient, they have more potential to support more sophisticated queries than hierarchical database structures.

In order to present the query functions, we will divide them into three sub-classes according to the information required to process each query function. These three sub-classes are: help query functions, structure-based query functions, and semantics-based query functions.
6.2.1. Help query functions

This class of query functions apparently belongs to the fact retrieval type. They access help information stored with various concepts at various levels so that the Display Module can display them to the user. A user can request help information about the system, available commands, details of each command, available methodologies, details of each methodology, use of each blank form or organizer, or even examples. Though the help information is scattered everywhere in the databases, the actual commands for these query functions have a uniform format:

HELP(key1, key2, ..., keyn).

The sequence of keys is used to identify the path of the requested help information stored in the hierarchy. Some keys are optional as long as the Keyboard Monitor can identify the path using the default values. Since the volume of help information is large, they are stored with the objects they are describing, instead of being kept together in another database. The hierarchy of help information is shown in Figure 20.

For example, the query HELP(DA,6,EXAMPLE) would cause the Example Form of Form-6 in the Data Abstraction Based Methodology to be displayed (See Figure 21).
System Help
  ┌──────────────────────┐
  │ Keyboard Help       │ Methodology Help
  │ Monitor Help        │ Library Help
  └──────────────────────┘
  │ Command Set Help    │
  │ How to define new commands │ All Methodologies Help
  │ Command Help        │ Each Methodology Help
  │ Customized Commands Help │ All Blank Forms Help
  │ Command Help        │ Each Blank Form Help
  │ Each Organizer Help │ Each Example Pattern Help
       ┌──────────────────┐
       │ Example Form Help │

Figure 20: Hierarchy of Help Information
Example Form-6

\begin{verbatim}
\textbf{data abstraction: Stack of Items}

\textbf{lower level data abstraction(s): Item}

\textbf{operation(s):}

\begin{itemize}
  \item [n.2] Create /* create an empty stack */
  \item [n.3] Push /* push an item to the stack */
  \item [n.4] Pop /* pop the top-most item from the stack */
  \item [n.5] Top /* read the top-most item of the stack */
\end{itemize}

\textbf{canonical form:} For each instance of Stack,

\begin{itemize}
  \item [n.6] Stack = Create(), or
    \begin{itemize}
      \item Stack = Push (Create(), a, a, \ldots, a), where
        \begin{itemize}
          \item a's are instances of Item.
        \end{itemize}
    \end{itemize}
\end{itemize}
\end{verbatim}

Figure 21: Display of the Result of the Query HELP(DA,6,EXAMPLE)

6.2.2. Structure-based query functions

This class of query functions retrieves information based on the structures provided by a form tree or a form. Since form trees and forms have hierarchical structures, both fact retrieval and deductive retrieval are efficient. These query functions heavily use the fetch operations and get-next (or traversal) operations to retrieve requested information in the hierarchy. The parameters required by these functions are keys needed to identify the location of the desired information. In the case of query functions for fact
retrieval, the implementation is fairly straightforward. Consider the following queries:

- What is the interpretation of organizer X in form F?
- What is the form refining organizer X in form F?
- What is the Form-use-number of the form refining organizer X in form F?
- What are the forms refining form F?
- What are the leaf forms of this form tree?
- Show me the form with Form-use-number 123?
- What are the attributes of organizer X in form F?
- Find all occurrences of organizer X in the subtree of form F.
- Find the form with form heading "Operation" in the subtree of form F.
- Find all the forms with form heading "Operation".
- Find the "Attendees" of "Meeting" on "Date: 1/1/83".
- Who is the "Reviewer" of the "Proposal: TRIAD" on "Date: 1/2/82"?
- What are the "Function Modules" defined in "Project: A"?
- What are "Operations" defined in "Data Abstraction: Stack"?

etc.

Actually, this list is endless. But, the functions needed to implement these queries are no more than fetch operations, traversal operations, get-next operations (to go down the hierarchy), and simple pattern-matching operations. Moreover, most of these operations are already defined with the data structures; thus, our
task of implementing these query functions is substantially simplified. Consider the following structure-based query function:

```
GetInterpretation(FT, FH, HC, ORG)
```

**Input:** form tree name "FT", form heading "FH", heading content "HC", and organizer "ORG".

**Output:** the interpretation of the organizer "ORG" in the form with form heading "FH" and heading content "HC" in form tree "FT".

**Action:**
- Get the form tree "FT";
- Find the first form with form heading "FH";
- Loop: Compare "HC" with heading content of this form heading;
  - If (not equal), find next form with form heading "FH" and continue the loop;
  - Else find the organizer "ORG" in the form;
    - fetch the interpretation of this "ORG";
    - exit the loop;

The last query example given in the above list is simply performed by this query function. That is, the result of the query: What are the "Operations" defined in "Data Abstraction: Stack"? is obtained by calling the function

```
GetInterpretation(current_form_tree,
  "Data Abstraction",
  "Stack",
  "Operations")
```

Note that in the actual user's commands, the system would prompt the user to supply arguments needed by the function interactively, so that the user has no need to memorize the argument list.
In addition to the fact retrieval type of query functions, the structure-based query functions also support deductive retrieval. This type of query function is also based on the hierarchical structure but the information requested is not stored in a single location in the hierarchy. Consider the following queries:

- What are the Form-use-numbers which contain interpretation A of organizer X?
- What are the forms using form F for refinement?
- How many forms are there in this form tree?
- Generate a list of interpretations of all occurrences of organizer X in the subtree of form F.
- Generate a list of all organizers whose interpretations contain the name "ABC".
- Show me the tree structure with only Form-use-numbers of the subtree of form F.
- Show me the tree structure with only Form Headings of the subtree of form F.
- Generate a list of all organizers which use the attribute A.
- What are the leaf forms which are not terminal forms?
- What are the "Dates" of "Meetings" that "Attendee: Jeremy Kuo" attended?
- What are the "Function Modules" which use "Operation: Push"?
- Generate a list of "Function Specifications" of all "Function Modules" defined in "Project: A".
- Generate a list of the "I/O Types" for all "Operations" defined in "Data Abstraction: Stack".

The information requested by these queries is usually scattered at more than one location in the hierarchy; thus the result has to
be prepared in some intermediate form. In the C-prototype, we use a
text buffer to prepare the result and display it to the user. Now,
consider the following query function:

```
---

GenerateInterpretationList(FT, FH, HC, ORG1, ORG2)
---

Input: form tree name "FT", form heading "FH", heading
content HC, and organizer ORG.

Output: a list of interpretations of the organizer "ORG2"
in the refinement form of all the organizers
"ORG1" in the subtree of the form with form
heading "FH" and heading content "HC" in the
form tree "FT".

Action: Get the form tree "FT";
Find the form with form heading "FH" and heading
content "HC";
For (each "ORG1" in this form) Do
    Get the refinement form of "ORG1";
    Fetch interpretation of "ORG2";
    Write it to the text buffer;
End Do;
Return the text buffer;
```

The last query given in the above list, Generate a list of "I/O
Types" for all "Operations" defined in "Data Abstraction: Stack", is
performed by calling this function

```
GenerateInterpretationList(current_form_tree,
    "Data Abstraction",
    "Stack",
    "Operation",
    "I/O Types")
```

The Display Module would then take this text buffer (the
intermediate form) and display it as follows:
**Form Tree Name:** current\_form\_tree name  
**Data Abstraction:** Stack

<table>
<thead>
<tr>
<th>Operation</th>
<th>I/O Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create</td>
<td>(\rightarrow) Stack</td>
</tr>
<tr>
<td>Push</td>
<td>Stack (\times) Item (\rightarrow) Stack</td>
</tr>
<tr>
<td>Pop</td>
<td>Stack (\rightarrow) Stack (\cup) ERROR</td>
</tr>
<tr>
<td>Top</td>
<td>Stack (\rightarrow) Item (\cup) ERROR</td>
</tr>
</tbody>
</table>

Similarly, the query next to the last one in the above list is also performed by calling this function, but only the arguments supplied are different:

\[
\text{GenerateInterpretationList}(\text{current\_form\_tree,}
\text{"Project",}
\text{"A",}
\text{"Function Module",}
\text{"Function Specification"}).
\]

It can be seen from these examples that the implementation of these query functions is no harder than that of the fact retrieval type of query functions. The pictures which contrast these two types of information retrieval in a form tree are shown in Figure 22.

Since most of the form tree structure is bound at the time of methodology specification, these structure-based query functions can be used to implement customized commands very efficiently for a particular methodology. For example, the project manager may want to define a customized command called MeetingReport(project name) which would automatically generate a list of all meeting reports in a
Figure 22: Information Retrieval of Structure-Based Query Functions
project. A programmer may want to define a customized command called DA-Operation(DA name) which automatically generates a list of documentation (without implementation code) of all operations defined in a data abstraction.

However, queries such as: Generate all "Meeting Reports" of "Meeting" from "Date: 1/1/83" to "Date: 3/1/83", can not be performed by any query functions discussed so far. Such queries use methodology-dependent semantics (about the "Dates"); thus the structure-based query functions are not able to answer this kind of query. The query functions needed to implement this kind of query are called semantics-based query functions, and these are discussed next.

6.2.3. Semantics-Based Query Functions

The main difference between semantic-based and structure-based query functions is that the locations of the information requested can not be identified using the form tree structure alone. Therefore, attributes and semantic functions are employed. Recall that an organizer corresponds to a symbol in the concept grammar. Any finite number of attributes and semantic functions may be associated with each symbol. For example, the organizer "Date" corresponds to the symbol with the same name "Date". One attribute of the symbol "Date" is "Value", which takes integer values. The
interpretation, say, 1/1/83, is first converted into an integer
830101 and stored in "Value". There may be at least three semantic
functions to test this attribute:

1. Before(Date,Value,n): returns 1 if Value is less than n;
2. After(Date,Value,n) : returns 1 if Value is larger than or
equal to n;
3. Between(Date,Value,n,m): returns 1 if Value is between n and m.

With the help of these attribute and semantic functions, we can
traverse the form tree and select the organizers, but we can fetch
the interpretations only if the result of a semantic function is 1.
In other words, the semantics-based query functions are similar to
the structure-based query functions, except that they can
selectively retrieve the information.

More specifically, consider the following semantics-based query
functions:

$\text{GetForm}(FH,f(\text{ORG},a,\text{arg1},\text{arg2},...))$
Find all forms with form heading "FH" provided that
the result of semantic function $f$ is nonzero. "$f$" is
a semantic function defined on the attribute "$a$" of
the organizer "ORG". Arg's are other arguments
required by "$f$".

$\text{GetOrganizer}(FH,HC,\text{ORG},f(\text{ORG1},a,\text{arg1},\text{arg2},...))$
Find all organizers "ORG" in the form with form
heading "FH" and heading content "HC", provided that
the semantic function "$f$" has nonzero result.

$\text{GetFormView}(FT,f(a,\text{arg1},\text{arg2},...))$
Get all forms in the form tree "FT" provided that the
semantic function "$f$" returns certain value. Where
"a" is an attribute common to all form headings. This
attribute is called **Form-View attribute** which can have values such as "Manager", "Engineer", "Programmer", "End User", "Requirements", "Design", "Specification", "Code", etc.

\$GetOrganizerView(FT, f(a, arg1, arg2, ...))

Get all organizers "ORG" in the form tree "FT", provided that the semantic function "f" returns certain value. Where "a" is called **Organizer-View attribute** and common to all organizers.

These functions are generic to all methodologies and can be used to selectively retrieve information from a form tree. For instance, the function

\$GetFormView(current_form_tree, f("Manager"))

would retrieve only "Manager"-related forms from the current form tree; while the following function

\$GetOrganizerView(current_form_tree, f("Code"))

would only retrieve all organizers which contain implementation code from the current form tree.

However, since most of the semantics of attributes are methodology dependent, many semantics-based queries are implemented as customized commands for each specific methodology. In TRIAD/SIE, only useful, **generic** functions are defined. An expert can design a very rich set of customized commands by invoking these generic structure-based and semantics-based query functions. As an example, the implementation of the customizer command to answer the query:
Generate a list of all "Meeting Reports" of "Meeting" from "Date: 1/1/83" to "Date: 3/1/83", is given below. Note that both the semantics-based and structure-based query functions are used to implement this command.

---

**MeetingReportFromTo(FT,n,m)**

Input: form tree name "FT", integers n, m.

Output: a list of interpretations of "Meeting Reports" in all forms "Meeting" with "Date" between n and m.

Action: Get form tree "FT";

Loop: form = $GetForm("Meeting", Between("Date","Value",n,m));
      Fetch interpretation of "Meeting Reports" in this form;
      Write this interpretation to text buffer;
      Continue the loop;

Return text buffer;  
---

6.3. Summary

As promised earlier, and as demonstrated in the previous two sections, the Software Information Editor TRIAD/SIE is able to create, edit, and manage all software-related information.

Both the editing and query functions defined in TRIAD/SIE can be used for all methodologies to edit and retrieve information in and from a form tree. Therefore, TRIAD/SIE is a **generic** tool in the environment.
Query functions can be used to implement friendly query commands which can retrieve methodology-oriented chunks of information (like Meeting Reports of all Meetings, I/O Types of all Operations in a Data Abstraction, etc.), provide different views of the information (like management-related or programmer-related forms of a form tree), or selectively retrieve information through the use of attributes and semantic functions. Therefore, all information collected is made accessible to the users.

With the power of these editing and query functions, many immediate applications are possible:

- A document editor with simple methodologies,
- An office automation system which uses forms to organize schedules, calendars, application forms, etc.,
- A personal database which uses forms to organize any kind of information,
- A programming environment which also uses forms to organize both the documentation and language constructs,
- A special-purpose workstation with a smaller set of methodologies with highly-customized commands, or
- A general-purpose software environment such as TRIAD itself.

One should be reminded again that the power of TRIAD/SIE is mainly due to the following three facts: firstly, the information is highly-structured according to methodology descriptions; secondly, the techniques of information retrieval developed in the database area are incorporated into the design of query functions; and
finally, attributes and semantic functions are much more flexibly used to deal with all software-related information instead of being restricted to the level of programming languages.
7. THE DESIGN AND IMPLEMENTATION OF THE C-PROTOTYPE

... the only thing that is important is whether a result can be achieved in a finite number of elementary steps or not.

-- J. Von Neumann

In this chapter, the design and implementation of a human-engineered prototype of TRIAD will be discussed. This prototype is designed to be a working prototype which reflects a more up-to-date status of the TRIAD environment. It is expected that this prototype can be used by graduate students to develop useful software projects and can be enhanced to become a high-performance software environment for a larger user community in the future. In contrast with the LISP-prototype, this C-prototype has the following improved features:

- A highly human-engineered User Interface using the form-based approach as described in Chapter Five.

- A powerful Software Information Editor which can edit, manipulate, and retrieve all software-related information as described in Chapter Six.

- A self-supported Methodology Tuner which can create new
methodologies, or modify, specialize, and generalize existing methodologies.

- A multi-windowed Display Module which can dynamically display various types of information in multiple windows on the screen.

The above mentioned features are all tailored using the form-based approach as described in Chapter Five. For example, databases are organized as blank forms and form trees; tools can manipulate forms and form trees; even the Tuner also uses forms to tune other forms. Through the whole design and implementation of this prototype, it can be seen that the fundamental design philosophy is the integration of three inter-related main scenes:

1. The structuring of information according to methodologies: This includes the design of the Form Tree Data Base, the Methodology Library, as well as the design of the methodologies themselves.

2. Tool support for manipulation of information: This includes the design of TRIAD/SIE as a constructive and maintenance tool to manipulate the structured information, as well as the design of Methodology Tuner as an enhancement tool to manipulate the methodology descriptions.

3. A human-engineered user interface using the form-based approach: This includes the design of Keyboard Monitor, user's commands, as well as the design of blank forms, help information, and display strategies.

Since the design specifications and implementation documentation are far too lengthy to be included here, only the
important issues related to the design and implementation of this prototype will be discussed.

7.1. Design of the C-Prototype

7.1.1. System Organization of the C-Prototype

The prototype is first decomposed into smaller components. An overview of the organization of these components is given in Figure 23.

Figure 23: Overview of the C-Prototype's System Architecture
The functionality of each component is briefly described below:

**Keyboard Monitor**

It is the responsibility of the Keyboard Monitor to interpret and execute each of the user's commands. A user's command can be a keyboard command, a macro definition, or an extended command. Each one of these corresponds to a C-procedure which invokes routines defined in other components. Keyboard Monitor is the only component that directly interfaces with the user, and the rest of the system is actually hidden behind the Keyboard Monitor. However, all the commands can still be logically grouped together to reflect the system architecture.

From the system's point of view, the commands can be divided into two sets. One set of the commands is generic to all methodologies and can be used to edit any form trees or to request the system status. The other set of commands consists of a collection of customized commands for different methodologies which can be used to manipulate the methodology-oriented information more effectively. This set of customized commands can be defined by either a user or a methodology designer, and is loaded at the same time that methodology is loaded. From the user's point of view, there is no need to distinguish whether a command invokes routines defined in one tool or another. However, the Keyboard Monitor
always informs the user about its status such as the name and content of the active buffers. Other detailed system information can also be made available to the user through help commands.

The monitor is also designed to be flexible and extensible so that the binding between a command and a routine can be easily customized for the end-user's environment, and the set of commands can also be easily extended by the user for different applications.

**Buffer Management Module (BMM)**

This module is responsible for maintaining the user's working space for creating, editing, or manipulating information in the databases. A buffer, or a form tree buffer, has the same hierarchical structure as a form tree. It contains the run-time information about the form tree and a list of form buffers which correspond to the forms in this form tree. Similarly, a form buffer contains the run-time information about a form and a list of organizer buffers which correspond to the organizers and entries in this form. Note that a buffer is not a copy of the information in the data bases; it is only an intermediate structure that contains the run-time status about the information and a pointer to the actual data in the databases. The run-time status is to keep track of the active buffers, as well as the information needed by the Display Module to display various kinds of information properly.
Form Tree Data Base and Methodology Library

These two databases are similar to those in the LISP-prototype except that methodologies are represented as a set of blank forms instead of productions, and that the information collected is represented as form trees instead of refinement trees. However, the productions of a concept grammar are still stored with the methodology, and the refinement tree can still be easily constructed from a form tree. The additional information needed for forms and form trees is mainly used to human-engineer the external representation of the collected information in order to interface with users in a more friendly way. The attributes, semantic functions, and derivation trees are still required for information manipulation and analysis. In Methodology Library, the methodology-oriented patterns and customized commands are also stored with that methodology. As soon as a methodology is selected by a user, the system automatically loads the methodology, the patterns, and the customized commands to customize the environment itself.
TRIAD/SIE

This is the Software Information Editor which can create, edit, and manipulate form trees as an information editor, and retrieve useful information from FTDB as an information manager. While the methodologies provide a way of structuring information, TRIAD/SIE provides the functions to construct and access this structured information. The functionality of this tool is described in Chapter Six.

Methodology Tuner

This module is designed to create new methodologies or enhance an existing methodology when more experience is gained. Enhancing a methodology means that one can add new blank forms, combine or divide existing blank forms to make new blank forms, make a well-developed pattern into new blank forms, modify the display specification of blank forms, or modify attributes and semantic functions in the grammar. In the Methodology Library, there are methodologies, called meta-methodologies, which can be used to tune other methodologies. The tuner can make use of such a meta-methodology and the routines provided by TRIAD/SIE to perform the tuning. This is a highly efficient approach to extending or customizing an environment in the sense that the tuning process is self-supported, and the consistency of the environment is thus
guaranteed.

**Display Module**

The Display Module is responsible for displaying the information properly on the screen and organizing the screen into smaller units so that different pieces of information can be displayed at the same time. Three types of information need to be handled by this module. The first type is the static information stored in the Methodology Library such as help information, blank forms, and other information associated with the methodologies. The second type is the dynamic information which resides in the various buffers such as form trees, forms, and various reports generated by the query functions. The third type is also dynamic information about the system's run-time status such as the messages to be displayed on the modeline and the error messages to be displayed on the bottom lines of the screen.

### 7.1.2. Other Design Decisions

The following are some of the design decisions which are important to the detailed design, project management, or implementation strategies of this prototype:

1. **Project decomposition**: The entire prototype project is decomposed into sub-projects based on the system
decomposition discussed earlier. Three sub-projects are identified as the databases and TRIAD/SIE, Buffer Management Module and Tuner, and the Display Module. The detailed design and implementation of these three sub-projects are developed and implemented by different programmers. These components are integrated later at the level of Keyboard Monitor. The idea is to reduce the inter-dependency of components with different functionalities in order to obtain more flexibility for future modification and enhancement. Each sub-project is responsible for implementing the data structures needed, as well as establishing the set of high level routines needed to implement the command procedures.

2. **Uniform data representation:** In order to reduce the complexity of integrating different components, it has been decided that the data structures of forms and form trees will be the common internal representation of the information collected. This approach is different from having the databases maintain the refinement trees and having the Display Module display them as forms and form trees. Therefore, all routines provided by various components can communicate with each other through this common representation, and thus, there is no need for data conversions between different routines.
3. **Generic functions:** All the high level routines defined in each component should be generic in the sense that they can be invoked by Keyboard Monitor to manipulate the information of any form tree created by using any methodology. Moreover, these high level routines should also be general enough so that they can be used to implement **customized commands** for a particular methodology. In other words, no programmers should make naive assumptions based on the implementation to restrict the generality of a high level routine. However, lower level routines which are local to a component and not visible to other components are not affected by this decision.

4. **Documentation standard:** Standards for documenting design decisions, implementation decisions, and C-programs are also established before the actual coding begins. Basically, the standards for documenting the design and coding of data structures and functions are the same as the blank forms defined in the Data Abstraction Based Methodology given in Chapter Five.¹

¹Though the programmers still have to use the blank forms manually, the documentations of this prototype have been impressively well-structured.
5. **Implementation language**: The language C, a so-called "system programming language", was chosen for the implementation language due to its availability and its many attractive features such as its efficiency and its ability to deal directly with memory addresses [Kern78].

### 7.2. Implementation Strategies of the C-Prototype

In this section, the implementation strategies of each component are discussed. Since the prototype has to support text editing at the lower level, the possibility of adopting some implementation techniques in Emacs editor [Gosl80] was first studied. This study concluded that the implementation of keyboard monitor, buffer structures, and display module in Emacs can be extended to support the C-prototype. For instance, the editing functions provided by Emacs can be used to implement part of TRIAD/SIE, and the keyboard structure and command interpreter in Emacs can also be used to implement part of the Keyboard Monitor. Therefore, at the level of Keyboard Monitor or text editing, this prototype has characteristics similar to those of Emacs. However, the databases and tools in the prototype are completely different. The buffer structures have to be extended from text buffers to much larger buffer structures in order to handle forms and form trees. The display module has also to be extended from displaying only text
to displaying forms and form trees. In the following sub-sections, the implementation strategies of each component are discussed in terms of data structures and routines defined.

7.2.1. Keyboard Monitor

A major data structure defined in the Keyboard Monitor is the Command Table. A Command Table is a table-like C-structure which stores each of the command names and the command procedures bound to this command name. Each entity of a Command Table is called a Command Description Block which stores the command name, command type, a pointer to the command procedure, and other information. A command type can be a keyboard command which is bound to a key on the terminal keyboard, a keyboard macro which is a macro definition defined using existing commands and is also bound to a key, or an extended command which usually takes a longer string.

In the monitor, a number of command tables are maintained. When a command is issued by a user, the keys typed in by the user are first interpreted by a lexical analyzer, and the monitor will look up the appropriate command table to find the actual command procedure bound to this command and then execute the command procedure. A number of command tables are necessary for several reasons. Firstly, the frequently-used commands are usually bound to the keyboard commands, while other less frequently-used commands are
usually bound to the extended commands. These two types of commands have different prefix keys (such as the control-key "^" for the keyboard commands and the "M-X" key for the extended commands), Therefore, the keyboard interpreter uses these prefix keys to identify the appropriate command table in order to speed up the searching for the command procedures. Secondly, commands are organized into groups according to their logical functionalities. Each group corresponds to a command table and uses the same prefix keys. Therefore, it is easier for the user to memorize the command names. Thirdly, different command tables are necessary to distinguish generic commands and customized commands. The command tables for generic commands always reside in the memory, while the command tables for the customized commands are loaded only if a particular methodology is loaded or under the user's request. Finally, this organization also simplifies the task of maintaining the Keyboard Monitor. For instance, the user-defined commands are kept separate from the others; therefore, deletion or modification of the user-defined commands will not jeopardize the rest of the table structures in the monitor.

A command table for keyboard commands is called the Key Map Table, and a command table for extended commands is called the Extended Command Table. Both the generic commands and customized commands (for each methodology) have a key map table and an extended
command table. The organization of command tables in the Keyboard Monitor is given in Figure 24.

Another data structure needed in the monitor is the **Abbreviation Table**. This abbreviation table stores all the abbreviations of the names which can be recognized by the system (for example, names such as command names — including the user-defined command names — methodology names, and even the organizers). Whenever a string of characters supplied by the user is sufficient to identify a specific name, the user can hit the "Esc" key to let the monitor complete it.\(^1\) Since the speed of searching is critical here, this table is implemented as a binary tree which reduces both the time of searching as well as the time of insertion and deletion.

Still another data structure needed in the monitor is the **Mini-Buffer**. A mini-buffer is used to display the messages shown on the bottom lines of the screen. Since the messages to be displayed here are simple character strings such as error messages or system prompts, this mini-buffer is implemented as a large character array instead of a text buffer.

---

\(^1\) The monitor can also accept some identifiable incomplete names followed by the "Return" key.
Figure 24: Organization of Command Tables in the Keyboard Monitor
The following routines are defined in this component:

- routines to interpret a keyboard command (that is, the lexical analyzer) and find the associated command procedure from the appropriate command table,

- routines to manipulate the command table structures such as binding, inserting, deleting, searching, etc. so that the system programmer and the users can add, delete, or re-bind the commands easily,

- routines to manipulate the abbreviation table such as creating, inserting, deleting, searching, etc.,

- routines to load or unload command tables from or to the Methodology Library,

- routines to execute the command procedures initiated by a user's command, and

- routines to fetch the system prompts or error messages to be displayed in the mini-buffer.

The most important part of the Keyboard Monitor is a large collection of command procedures. Each command procedure corresponds to a user's command which usually invokes routines defined in other components. The implementation strategies of these command procedures are discussed in the next section.

7.2.2. Buffer Management Module

The data structure of the buffers is defined as a hierarchy of C-structures for maintaining the form tree structure which is currently been working on. More specifically, the top level of the hierarchy is the form tree buffer which contains a pointer to a list of form buffers. Each form buffer corresponds to a form in the form
tree and contains a pointer to a list of organizer buffers. An organizer buffer is the lowest level of the hierarchy and is implemented as a text buffer in Emacs. The organization of buffer structures defined in BMM is shown in Figure 25.

In addition to maintaining the hierarchical structure as that of a form tree, these buffers also maintain the run-time information about the form tree. This run-time information includes the display information of each buffer, a hierarchy of current buffers, the cursor position, the position markers created by the user, the links between each buffer and the database, as well as the links between each buffer and the virtual screen. Since all the user's commands are working on the information maintained by these buffers, the data structures defined in BMM have to contain all information needed by the high level routines defined in different tools.

Currently, there is no limit to the number of form tree buffers or organizer buffers which reside in BMM. But, BMM only maintains a constant number of form buffers. This is because the number of forms in a form tree can be arbitrarily large, while the number of form trees and the number of organizers in a form are usually very small. Whenever a form tree is initially accessed by a user's command, BMM instantiates and initializes a form tree buffer for that form tree. In the meanwhile, a fixed number of form buffers and all organizer
Figure 25: Organization of Buffer Structures in BMM
buffers of each form buffer are also instantiated and initialized. Since the number of form buffers is a constant, not all the forms in the form tree will have a corresponding form buffer. Therefore, BMM also keeps track of the access statistics of each form buffer. When a new form is accessed which does not reside in the buffers, BMM re-initializes the content of the least-recent-used form buffer for this new form and constructs the organizer buffers for the organizers in this form.

The routines defined in this component include:

- routines to instantiate and initialize (including links to the database and virtual screen) a hierarchy of various buffers for a form tree,
- routines to set up the current buffers at different levels of the hierarchy,
- routines to maintain the marker positions and the cursor position in the current buffers,
- routines to keep track of the least-recent-used form buffer,
- routines to manipulate the list of form buffers (such as fetch, search, insert, and delete routines),
- routines to manipulate the list of organizer buffers,
- routines to free the memory when a buffer is deleted, and
- routines to maintain the killed buffers for later recovery.

Note that the initialization routines have to bind the buffers to appropriate data structures in the database, on the one hand, and bind the buffers to the appropriate virtual screens and panels, on
the other hand. Except for these initialization routines, the rest of the routines are independent of the other components.

7.2.3. Form Tree Data Base and Methodology Library

Since C supports user-defined data types, the data structures in the databases are defined as data abstractions. The relationships between these data abstractions are illustrated in Figure 26.

The data abstractions defined for a methodology description are based on the attribute grammar form model and are similar to the MDDB in the LISP-prototype. Blank_Form is a new data abstraction which specifies a form structure for the underlying concept tree. It contains a pointer to the concept tree and a pointer to a set of blank forms to refine a non-terminal organizer. A blank form also contains the Display Specification which is used by Display Module to display the blank form and all the filled form instantiated from this blank form. The detail of the display specification is discussed in the sub-section of Display Module.

The data structures in FTDB are organized as filled forms and form trees. A filled form is a recursively defined C-structure which contains a pointer to its parent form and a pointer to its children forms. A filled form also contains a pointer to the blank form from which it is instantiated. When a filled form is instantiated, a
<table>
<thead>
<tr>
<th>Data Abstractions Defined in Methodology Library</th>
<th>Data Abstractions Defined in Form Tree Data Base</th>
</tr>
</thead>
<tbody>
<tr>
<td>SET OF METHODOLOGY</td>
<td>DIRECTORY</td>
</tr>
<tr>
<td>METHODOLOGY</td>
<td>FORM TREE</td>
</tr>
<tr>
<td>CONCEPT GRAMMAR</td>
<td>TEXT FILE</td>
</tr>
<tr>
<td>SET OF BLANK FORM</td>
<td>FILLED FORM</td>
</tr>
<tr>
<td>BLANK FORM</td>
<td>LIST OF FILLED FORM</td>
</tr>
<tr>
<td>PRODUCTION</td>
<td>RENEFEMENT TREE</td>
</tr>
<tr>
<td>CONCEPT TREE</td>
<td>NODE</td>
</tr>
<tr>
<td>LIST OF SYMBOL</td>
<td></td>
</tr>
<tr>
<td>SYMBOL</td>
<td></td>
</tr>
<tr>
<td>SET OF ATTRIBUTE</td>
<td></td>
</tr>
<tr>
<td>ATTRIBUTE</td>
<td></td>
</tr>
</tbody>
</table>

Figure 26: Relationships between Data Abstractions Defined in FTDB and Methodology Library
refinement tree is also instantiated from the concept tree and stored with the filled form. Each node of the refinement tree corresponds to an organizer in the filled form, and a pointer to its entry is also contained in the node. The data structure of a node is also a recursively defined C-structure which is used to implement both the concept tree and the refinement tree.

On top of FTDB and Methodology Library, the data abstraction DIRECTORY specifies the structure of a user's directory. A directory is implemented as a tree where each node can be either a form tree or an ordinary text file.

In addition to the data abstractions mentioned above, the library also has a tree structure to maintain the help information. Again, all the actual text of help information is stored with each methodology, blank form, command, etc. This tree only contains pointers to the actual help information. The reason for having such a tree in the library is to speed up the searching for all help commands. The help information is still kept together with the methodologies and commands so that it is easier to maintain the consistency.

These data abstractions are basically trees, lists, sets, and records, which can be easily implemented in terms of C-structures. Each data abstraction contains a set of operations which can create,
initialize, insert, delete, traverse, search, and fetch the structure or its sub-structures.

7.2.4. TRIAD/SIE

The data structures needed in this component are already defined in FTDB, Methodology Library, and BMM as described above. The routines needed in this component have also been discussed in the previous chapter. Note that the difficult part of this component is that it must provide routines for instantiating a filled form from a blank form and generating a refinement tree from a concept tree. In other words, it is the responsibility of this component to take care of the interface between FTDB and Methodology Library.

7.2.5. Methodology Tuner

Since Methodology Tuner also uses methodologies (or meta-methodologies) to tune the existing methodologies, all the data structures needed here are already defined except that of the display specification of a blank form. A display specification is also a table-like C-structure. Each entity in this table describes how each visible organizer and its entry in a blank form are to be displayed. The details of the display specification will be discussed below in the sub-section on Display Module.
One thing which needs to be clarified here about the display specification is that the data structure of a display specification is stored with a blank form in Methodology Library and the content of the data structure is used by Display Module. However, neither of these is authorized to modify the data structure of a display specification. It is the responsibility of Methodology Tuner to maintain and manipulate this data structure.

Many routines needed by the tuner are already defined in TRIAD/SIE. However, the following routines still need to be defined in this component:

- routines to create a brand new blank form and its display specification,
- routines to create a new blank form from an existing form tree and its display specification,
- routines to modify a blank form and its display specification,
- routines to create or modify a methodology by adding or deleting blank forms,
- routines to create or modify a grammar during creating or modifying a methodology as mentioned above, and
- routines to add an authorized methodology to the Methodology Library (or replace it).

Note that most of the routines are the same as the routines provided by TRIAD/SIE, but TRIAD/SIE does not maintain the consistency of a methodology or the library. For example, a blank form number should be unique, the blank forms to refine a new
organizer should be correct, the display specification should remain correct, all the help information should remain accurate, and the attributes and semantic functions should also remain consistent. It is the responsibility of Tuner to make sure that any change made on a methodology or the library will not violate the consistency.

7.2.6. Display Module

The data structures defined in Display Module are Virtual Screens, Panels, Windows, and the Desired Screen. These data structures are organized as a hierarchy to reflect the form tree structure. A blank form or a filled form corresponds to a virtual screen which is constructed according to its display specification. It is only a "virtual" screen because it ignores the limited size of the physical screen. Within a virtual screen there are small panels, each of which corresponds to an organizer and its entry.

Display Module also maintains a list of Windows. Each virtual screen is bound to a window, and a window is in turn bound to a portion of the physical screen. Among the windows in the list, there are a certain number of visible windows. All the visible windows are displayed on the physical screen so that more than one form can be made visible to the user at one time. Among the visible windows, there is only one active window. The active window corresponds to the active virtual screen and the active form buffer on which the
user is currently working.

There is a strong relationship between BMM and Display Module. When a form buffer is created, a virtual screen is also created and bound to this form buffer. Similarly, the panels of the virtual screen are also created and bound to the corresponding organizer buffers. In order to construct the virtual screen for each form buffer, the Display Module has to fetch the display specification from that blank form. More specifically, the display specification of a blank form specifies the size of the virtual screen, the size and the relative location (using coordinates) of each panel resides in the virtual screen, and pointers to the actual form heading, organizers, and the entries to be displayed. The active virtual screen is also bound to the active form buffer; therefore, the information in the active form buffer is always ready to be displayed. This module also keeps track of the visible windows. Only the virtual screen bound to these visible windows will be displayed. Since the window size is usually smaller than the size of a virtual screen, only a portion of the virtual screen can be displayed. Therefore, the size and the location of the visible portion of a virtual screen is also contained in the virtual screen.

Display Module also maintains a data structure called Dot Positions which is similar to the Markers defined in BMM. In
particular, the dot position of the visible active panel is exactly the cursor position displayed on the physical screen.

The organization of the data structures in this module is given in Figure 27.

Except when a command is issued to change a window or a panel, Display Module is unaware of the information changes made within the entries. From the Display Module's point of view, it simply accepts whatever content is provided by BMM (through pointers) and dumps the information on the indicated region of the physical screen. However, repeatedly dumping the entire information is very inefficient, especially since most of the time, the changes are as simple as inserting a character or deleting a word, in which case only one line of the screen or a small region of the screen needs to be re-displayed. The Desired Screen is thus used to optimize the re-display effort. The desired screen is an exact image of the physical screen. Whenever a re-display is necessary, Display Module compares the changes made after previous display and the content of the desired screen to check if any optimization is possible. If the change made only affects one line on the desired screen (and this is usually the case), then only the new content of that line is re-displayed. Therefore, it substantially speeds up the re-display routine.
Figure 27: Organization of Data Structures in Display Module
The routines defined in this module can be divided into three groups. The first group is the interface routines which set up appropriate virtual screens and panels for various buffers in BMM. The second group is a large number of routines for screen management and is independent of the rest of the system. The third group is the physical display routines. These three groups of display routines are briefly described below:

1. Interface routines:
   - routines to construct a virtual screen for each form buffer,
   - routines to construct panels for organizer buffers in a form buffer,
   - routines to bind the active virtual screen to the active form buffer and to bind the active virtual screen to a visible window,
   - routines to bind the active organizer buffer to the active panel and to make sure that the active organizer buffer is visible,
   - routines to update the cursor position according to the current dot position obtained from BMM,
   - routines to erase or delete a virtual screen when the corresponding form buffer is erased or deleted,
   - routines to maintain the pointers to the actual information for the content of all virtual screens and panels, and
   - routines to find the virtual screen or panel for a form buffer or an organizer.

2. Screen management routines:
   - routines to create and initialize a virtual screen, a panel, or a window,
- routines to delete a virtual screen, a panel, or a window,

- routines to maintain the existing virtual screen list and window list (such as screen numbers, window numbers, and the last displayed virtual screen),

- routines to change the visible portion of a virtual screen,

- routines for window management such as splitting a window, changing window size, scrolling a window, or making a window visible or invisible, and

- routines for panel management such as splitting a panel, changing panel size, scrolling a panel, or making a panel visible or invisible.

3. Physical display routines:

- routines to optimize the re-display,

- routines to fetch the needed amount of actual information from the database (through BMM),

- routines to make sure that the information to be displayed can fit in the physical window, and

- routines to dump the actual information on the physical screen.

So far, the implementation strategies of displaying a form tree have not been discussed. In order to reduce the implementation time of this component, a form tree is displayed in the same way as a form. That is, a form tree structure to be displayed is also bound to a virtual screen, and each form in this form tree is bound to a panel. From the Display Module's point of view, the virtual screen bound to a form tree has exactly the same structure as other virtual screens. The form tree name is displayed as a form heading, each
form heading is displayed as an organizer, and the information about a form (such as the interpretation of the form heading and the Form-use-number) is displayed as an entry. Currently, all form trees have default display specification. Therefore, no extra routines are needed to construct such a virtual screen for a form tree. However, it is the responsibility of BMM to bind, re-bind, or un-bind a form tree to such a virtual screen. In the future, better display specifications and display routines to pretty-print a form tree should be designed.

7.3. Integration of Different Components

Now, the data structures and high level routines needed to implement this prototype are already available. That is, the bones and meat are already there, and the only thing left is the blood to make this prototype work. The "blood" consists of a large set of command procedures and a set of methodologies. Most of the command procedures invoke routines defined in more than one component; thus, implementation of these command procedures is the major task of integration.

The general strategies of integrating the data structures and routines defined in different components are discussed below:

1. Integrating the Routines at the Highest Level: As mentioned earlier, each component has to provide a set of high level
routines. The purpose of these high level routines is to reduce the complexity of each command procedure. For example, many different kinds of tree traversal routines have already been defined in TRIAD/SIE so that a command procedure can simply invoke such a routine instead of traversing the tree by itself. In other words, the functionality of the high level routines provided in each component have been designed as close as possible to a command procedure. These high level routines permit us to implement the command procedures at the high level; therefore, the complexity of implementing a command procedure is greatly reduced.

2. Identify the Common Routines: Although the routines provided in each component are sufficient to manipulate the data structures at a higher level, some of them may be redundant or partly overlapped. In fact, there are many command procedures which perform very similar functions. Therefore, it is desirable to identify common routines which are useful to implement many command procedures so that the number of routines in each component can be reduced. In order to do this, the functionality of each command procedure is first specified in a pseudo language, and then many common routines are identified by studying the specifications of
similar command procedures. For example, the routine to set up a panel was first defined only for the purpose of constructing a panel during the creation of a new form buffer. Later, it was found that this routine was also useful during the creation of a form tree buffer (because a form tree is also displayed as a form), and whenever a form was inserted to a form tree or a new organizer was generated from a repeatable organizer. Instead of defining three routines to perform these three similar tasks respectively, the original routine is slightly modified to perform these three tasks.

3. **Reduce the Degree of Data Coupling:** As discussed in the previous section, it should be clear that no component has to be aware of any local data structures defined in other components. For example, even though the buffer routines have to set up various kind of buffers for each form tree, BMM does not manipulate the structure of a form tree or a form directly. It simply invokes routines defined in TRIAD/SIE or FTDB to set up the binding properly. In other words, the communications between different components are mostly through function invocations rather than through data transmissions. Therefore, even when the local data structures of a component are modified, the other components
will not be affected. Only very few global variables are defined outside all components. These global variables are mostly simple variables which keep track of the current buffers and the dot positions. Command procedures can directly access these global variables without going through BMM or Display Module and, thus, are more time-effective.

It can be seen that every effort has been made to reduce the degree of data coupling (and increase the degree of function coupling) between different components in order to improve the flexibility and maintainability of this prototype. Apparently, such design and implementation strategies can not guarantee that the performance of this prototype will be efficient. However, the flexibility and maintainability of an evolving prototype deserve a much higher priority than the performance efficiency. It is expected that after the design phase and implementation phase of this prototype, the performance issues will be the major task during the testing and optimization phase of this prototype.

7.3.1. Command Procedures

The command procedures, according to their functionality from the user's point of view, can be divided into the following categories:

1. User-directory command procedures: to create, display, and
manipulate a user-directory.

2. Library command procedures: to load/unload methodologies, extended commands, and other system routines from/to the library.

3. Editing command procedures: to edit entries, forms, and form trees.

4. Query command procedures: to retrieve information from a form tree and generate reports based on a query command.

5. Tuning command procedures: to tune an existing methodology description using a meta-methodology.

6. Display command procedures: to manipulate the windows and panels which are visible to the user, such as redisplay, scrolling up and down, creating a new window or panel, etc. These procedures are independent of buffers and databases, but useful for the user to have a control of the physical screen.

7. Buffer command procedures: to manage extra working buffers created by the user, in addition to the form tree buffers, form buffers, etc. For example, a user may request a temporary buffer to store some information which is not part of a form tree or he/she has not yet decided where to store it.

In the beginning, the routines defined in Keyboard Monitor (that is, the command interpreter) interpret a user's command. It may have to call certain routines to look up the abbreviation table or read in the arguments supplied by the user, and then find the appropriate command procedure from a Command Table. From then on, the command procedure is executed until it is completed or aborted, or whenever errors are detected. This process is repeated when a new command is issued, but the user does not have to be aware of this.
Due to the large number of command procedures, it is not possible for us to present all command procedures here. However, a general idea of how a command procedure works is provided below:

1. Determine what and where the input data are.
2. Make sure that the data are in the buffers.
3. Make sure that the data are in the current buffers.
4. Make sure that virtual screen and panels are also ready.
5. Execute routines defined in TRIAD/SIE, Tuner, Display Module or BMM to manipulate the data.
6. Update modeline and prepare prompts or error messages to be displayed.
7. Redisplay the current window (possibly optimized).

In order to illustrate the implementation of the command procedures, the specifications of two command procedures are presented below. These two examples are randomly chosen: one is a simple editing command procedure, called NEXT_ORGANIZER, which moves the cursor to the beginning position of the entry of the organizer next to the current organizer; the other is a query command procedure, called ORGANIZER_LIST, which retrieves the entries of a given organizer existing in a form tree and appends them to a temporary text buffer.

EXAMPLE-1

Command Procedure: NEXT_ORGANIZER()

1. Get the current panel in the current virtual screen bound to
the current window;

2. Get the next panel from the panel list in the current virtual screen; (Note that the next organizer is the organizer visible to the user, which is not necessarily the next node in the refinement tree. Therefore, it searches the panel list instead of the tree in the database.)

3. Get the pointer to the next organizer from the panel structure;

4. Set the current organizer buffer to the organizer buffer bound to this organizer; (Current form tree buffer and form buffer remain the same.)

5. Set the current panel to this new panel; (Current window and current virtual screen remain the same.)

6. Set the current dot position and the cursor to the beginning position of the entry of this organizer;

7. If this panel is not visible, scroll the virtual screen;

8. Check if one-line optimization or half-window optimization is possible;

9. Dump the information which can not be optimized on the screen;

Note that the organizer buffers and panels of the current form are always available in BMM and Display Module after the form is visited. Therefore, this command procedure only re-initializes some global variables at the organizer level and text level. In fact, if the new organizer is already visible on the screen, the only effect apparent to the user is that the cursor is moved to the beginning position in the next panel.

EXAMPLE-2

Command Procedure: ORGANIZER_LIST(form_tree, org)
1. If the "form_tree" is not the current form tree, check if it is already in the list of form tree buffers;

2. If it is not in the list, set up the buffers, virtual screen and panels for this form tree; also set the current form tree buffer to this form tree buffer;

3. Set up a temporary text buffer to store the report;

4. Copy the form tree name and the organizer name in the text buffer;

5. Traverse the form tree to prepare the report;

(Get the root form of the form tree;
For each form in the form tree do:
  Search the organizer list to find the given "org";
  If found, write the form heading and form-use-number in the text buffer;
  Get the pointer to the entry of this organizer;
  Copy the entry (and append it) to the temporary text buffer;
  Continue on next form in the form tree;)

6. Set the current window to this text buffer;

7. Change the modeline to indicate that this is a report and the form tree name;

8. Check if screen optimization is possible;

9. Dump this text buffer on the screen;

Note that step 5 is actually a TRIAD/SIE routine which prepares the text buffer for this command procedure. The responsibility can be roughly divided in the following way: steps 1-3 are done by routines defined in BMM, steps 4-5 are done by routines defined in TRIAD/SIE, and steps 6-9 are done by routines defined in Display Module.
The above two examples illustrate that each step performed in a command procedure roughly corresponds to a high level routine defined in one of the components. As pointed out earlier in the general strategies, this not only simplifies the implementation of each command procedure, but also promotes the modifiability and maintainability of the command procedures.

7.3.2. Methodologies and Meta-Methodologies

There are three methodologies to be implemented in this prototype. They are the Transaction Processing Methodology (TPM), the Data Abstraction Based Methodology (DAM), and the Project Development Methodology (PDM). The blank forms of DAM and PDM are given in Chapter Five and will not be discussed here.

In the beginning, an extremely simple but useful meta-methodology, called ALPHA-Methodology, is implemented. This meta-methodology consists of only three blank forms which are designed to create a blank form of a methodology. As a matter of fact, these three blank forms in ALPHA are merely the form representation of the data structures defined in the data abstraction of BLANK_FORM. Each organizer in the blank forms corresponds to a field of the data structure, which needs to be specified by the blank form designer.
The actual creation of a blank form is done by invoking the operations defined in the data abstraction of BLANK\_FORM. However, it is much more desirable to create a blank form using the meta-methodology instead of invoking an operation to enter the value of each field manually. Without using this meta-methodology, the form designer each time has to write a C-program which has to invoke more than twenty operations in order to create a blank form. This is apparently error-prone and time-consuming. Since the Display Module can display forms, the entire data structure of a BLANK\_FORM is represented as three blank forms on the screen (see Figure 28), and the form designer simply enters the value of each field or refines an organizer in the same way as editing a form. When all the entries are filled in, the form tree thus constructed contains all the information needed to create a blank form. A tuning command can then be issued to process this form tree, construct the data structure of the blank form, and store this blank form in the specified methodology description. Certainly, these three blank forms of ALPHA have to be created manually.
Figure 28: Blank Forms of the Meta-Methodology: ALPHA
7.4. Coding of the C-Prototype

The coding of this prototype began in August, 1982. Though it has lasted more than six months, the programmers were intensively coding the programs on a full-time basis for only three months. Originally, three programmers participated in coding all the components. Later, two more programmers joined the programmers' team to help implement the command procedures on a part-time basis. This prototype is expected to be completed in June, 1983 and will be running on VAX 11/780.

The progress of coding is actually three months behind the planned schedule due to various factors. At least two of them are worth being mentioned here:

- Since part of the implementation is based on the Emacs editor, a considerable amount of time was wasted in trying to understand and modify the poorly-structured and poorly-documented Emacs code. It is our common opinion that, in spite of its correctness and efficiency, this particular implementation of Emacs is extremely difficult to understand, modify, and extend.

- A large amount of time is also invested in producing only well-structured and well-documented programs. Since all of these still have to be done manually, the programmers'
productivity is not as impressive as that of implementing the LISP-prototype. However, it is our belief that such an investment in time and labor will be repaid in the near future.

By following the implementation strategies discussed earlier, the functions needed in each component have been modularized, using the layer approach, into small routines. In fact, the sizes of all the C-functions implemented range from only a few lines to two pages with an average of less than a page. Such small-size programs not only are easy to implement, but also are easy to debug, modify, and integrate.

The design of the prototype was developed in a top-down manner, but the implementation has been done in a bottom-up manner so that the program debugging and testing can be performed at the same time as coding. Finally, C is also a fairly pleasant language for implementing the prototype. For example, the user-defined types, macro definitions, function parameters, and partial re-compilation are particularly useful.
7.5. Future Enhancement of the C-Prototype

A list of possible tasks to enhance this prototype has been identified. The reason for identifying such a list for future enhancement at an earlier stage is to make sure that no implementation of the prototype will hinder the enhancement. As pointed out earlier, many efforts have been made to promote the extensibility and flexibility of this prototype. Therefore, it can be seen that the enhancement suggested by the following list still fits into the entire framework of this prototype, as discussed in the implementation strategies.

1. **More Methodologies**: More methodologies should be encoded and implemented in Methodology Library. It is important to have a variety of methodologies in order to attract more users of this prototype and thus assist the evolution of methodologies. A methodology can only evolve after it has been widely used. Three categories of methodologies have been brought to our attention:

   - The first one is the methodologies for writing small programs. These methodologies will be programming language-oriented as well as problem domain-oriented. The potential users of these methodologies will be the undergraduate students in the programming language courses in which PL/I or Pascal is taught. There have been well-developed patterns in the problems assigned to these students. Moreover, these students have been taught to use some basic methodologies like Top-Down Design and Jackson's methodology. Therefore, it is desirable
to include these methodologies and patterns in the library so that the user community of this prototype can be substantially expanded.

- The second category is the contemporary programming methodologies such as function decomposition methodologies or program testing methodologies. Since the Project Development Methodology described in Chapter Five is still fairly general, it is important to include these methodologies in order to provide better support for detailed design or maintenance during each phase of the life-cycle. When more experience is gained, it is possible to merge all of these methodologies to obtain a comprehensive methodology which can support the entire life-cycle of a software project at both the higher and the lower level.

- The last category is the methodologies for developing larger projects in industrial environments. These methodologies will be more application-specific than those in the previous category. Some of the TRIAD project members are currently consulting for a company which produces a variety of computer software. The engineers and programmers of this company are also the potential users of the TRIAD environment, and many important management-oriented issues of software development can only be identified and studied in such a real industrial environment. A more comprehensive methodology suitable for this company's environment, based on the PDM, is currently under design by two of the project members.

2. **More Meta-Methodologies:** Both the tuning commands and meta-methodologies should be expanded. In this prototype, only the tuning commands to create new blank forms are supported by the ALPHA meta-methodology as described earlier. Similarly, other tuning commands (see Section 7.2.5) should also be created through the use of meta-methodologies. Generally, a methodology is designed and
tuned by a project manager who is usually not a TRIAD expert. Therefore, it is important to have a set of easy-to-use meta-methodologies so that the project manager can tune a methodology easily.

3. **More Human-Engineering**: Many issues of further human-engineering of the prototype will probably become clear only after the users' feedback is available. However, at least two of these have been identified and were pointed out earlier. The first is to design a better display specification of a form tree, and the second is to design a set of more sophisticated error/prompt messages.

4. **Command Extension Language**: In this prototype, Keyboard Monitor is designed to be extensible. Unfortunately, only keyboard macros can be defined easily by a general user, and new keyboard commands or extended commands still require a new command procedure written in C. A keyboard macro is defined in terms of existing commands. Similarly, many useful extended commands can be defined in terms of existing command procedures. But, a command procedure is a C-procedure which can not be interpreted by the keyboard interpreter. In order to relieve the user from writing these C-procedures, a higher level extension language based on the MLISP in Emacs should be designed.
8. CONCLUSIONS

It is for us to make the effort. The result is always in God's hands.

-- Mohandas K. Gandhi

As discussed in the Introduction, the software produced today is costly and unreliable. In general, the software productivity is so low that hardly any software product can be delivered on time; the overall costs of software design and maintenance are never within the budget; the quality of final products rarely meets the requirements specifications; and very often they can not even operate correctly. It is predicted that, by 1985, the software cost will be 90% of the total cost of an application system, and 70% of the software cost will be spent in software maintenance [Boeh79]. These problems, or the so-called "software crisis", have been widely recognized as major research issues in software engineering.

The work described in this thesis provides an approach to supporting both software design and maintenance by organizing and structuring all software-related information according to methodologies. The structured information data base serves to integrate various tools in the environment. Furthermore, the structuring of information facilitates human-engineered commands
which invoke the tools to manipulate methodology-oriented chunks of information. In earlier chapters, many issues related to the above-mentioned software crisis have been addressed. In particular, the issues of software methodologies, information structuring, tools integration, human-engineered interface, documentation standardization, and software tool design have been discussed in detail. The concepts and techniques presented here can effectively improve the software productivity and reduce the design and maintenance costs. A prototype is also carefully designed and implemented. This prototype is expected to be completed in June, 1983 and to be used by graduate students at the Ohio State University.

In the following three sections, we will discuss the approach presented in this thesis, in a broader context, in order to identify some future research directions. Some of these future research topics are then discussed in the final section.

8.1. Relationships to Other Research Areas

No technology developed in one area of computer science can mature or become practical without being influenced by technology developed in other areas. During the last thirty or forty years, we have seen that research and development in areas like Very High Level Languages, Operating Systems, Artificial Intelligence,
Databases, and Computer Architecture all have benefited from one another to a large degree. Software engineering, although adapting many techniques developed in these areas, has its focus on identifying and formulating concepts and techniques of software development. It is useful to examine the research and development in these areas, within the context of software development, so that the significant results and future research directions in software engineering can be identified.

8.1.1. Very High Level Languages

Very High Level Languages (VHLL) have been an evolutionary research area since the birth of the first high level programming language. It has successfully promoted the activity of human-programming from the level of machine-oriented language constructs to a level at which the data structures, control structures, and operations are more abstract and are not necessarily computer-related. The goal of VHLL is to provide more and more non-procedural features in a programming language such that the programmer can specify his/her program more in terms of "what" he/she wants instead of "how" it is to be done. Though such evolution will never be complete, the work accomplished in VHLL research has substantially improved the effectiveness of using computers, as well as the productivity of programmers in producing
large software.

The difficulty in designing a VHLL is that a VHLL is usually designed to be a general-purpose language such that it can be used to write a large number of programs for various types of applications. A comprehensive set of abstract programming concepts has to be first identified before it can be incorporated into the language. Most, if not all, of these concepts, are not necessarily procedural and are thus difficult to implement in terms of existing programming languages. Even if they can be implemented using some particular algorithms, the implementation code generated from such algorithms (by a compiler) is usually very inefficient. The traditional optimization techniques used in a language compiler are no longer adequate. This is because the gap between non-procedural language constructs and executable code is much wider, and thus, it requires a global optimization which includes selecting data structures, algorithms, and operations for this non-procedural specification. At the same time, the traditional optimization techniques can only recognize small control structure patterns to achieve the optimization locally.

As far as software development is concerned, the approach presented in this thesis can identify useful abstract concepts, methods, and patterns of software design and implementation. These
will contribute to the design of a VHLL. However, the research in VHLL does not address issues like software life-cycle which include requirements specification, design specification, implementation, maintenance, and verification phases of a software project. In other words, software development is not only the activity of coding programs. Though VHLL will simplify the task of coding and maintenance of programs to some degree, an environment which supports only a VHLL (with its editor, compiler and debugger) is not at all a satisfactory software environment. Furthermore, VHLL does not explicitly support programming methodologies. The art of programming is more in its "design" rather than in the actual "coding" [Dijk77]. A VHLL does not really get involved in the activity of programming during the early design phase (which is usually language-independent) and the computer behaves like a "slave" to the programmer. A software environment, such as TRIAD, can interactively work together with the programmer to produce a program or software. However, a VHLL is a necessity of a successful software environment. Without it, the programmers would have to work in darkness before automatic programming becomes possible.
8.1.2. Operating Systems

It is probably safe to say that software engineering is substantially motivated by research in programming methodologies and operating systems. Operating systems used to be the only interface between the computer users and the computers. An operating system itself is one of the largest software products ever written, which requires a thorough understanding of many software-related as well as hardware-related issues. Many problems have been resolved, such as time-sharing systems, resource allocation for CPU scheduling, core memory and virtual memory management, and control processes management for deadlock and synchronization problems. These research results have successfully improved the efficiency and effectiveness of computer performance. It has been the case that many useful and optimized functions supported by operating systems in one generation of computer systems have become hardware or firmware features in the next generation [Denn79]. These issues have also stimulated computer scientists' interest in many other related issues such as distributed systems, concurrent programming, and computer architecture.

Current research in operating systems focuses on issues like distributed operating systems, operating system languages (so that operating system facilities can be made more accessible through
language notations rather than supervisor calls), and new operating system organizations due to the now reduced hardware costs. There are also efforts to identify a list of general functions which can provide better support for the software or subsystems running under an operating system. These functions have to be realizable according to the existing computer architecture, on the one hand, and be sophisticated enough to support a wide range of software applications on the other. This issue is directly related to the current research interest in software engineering. Researchers in software engineering have tried to identify a list of required support from the operating systems such that a software environment can access the computers more efficiently and effectively in order to improve the overall performance of the environments. For instance, the TRIAD project has been trying to identify some runtime support from the operating system such as multi-users, input/output routines, and memory management. TRIAD is also seeking an operating system to handle the "form" structure (on top of traditional file structure) in order to shift some tedious task to the operating system level.

The above-mentioned achievements in operating systems research are more oriented toward the mechanisms to solve problems, such as CPU scheduling and memory management. As far as software development is concerned, the quality of a software design comes from both
concepts and techniques. If we consider the design of an operating system as an activity of software design, it is not clear what the concepts are behind such an ingenious design. Implicitly, the concepts of virtual memory, abstract machines (i.e. the processes), and hiding machine details are all useful concepts for software design in general. It will be even more useful if the overall concept of operating systems design (such as the concepts of integration of different components, communication between processes, system organizations, and error or exception handling) can be identified.

Software engineering, on the other hand, addresses many issues of a quite different nature. For example, this thesis has addressed issues of software information structure, methodologies, human-engineered interface, abstract concepts of software design, software maintenance, and tools design. As software systems become more and more intelligent, and, in the meantime, hardware costs become less and less expensive, research in software engineering will have a definite impact on the design of new operating systems. In other words, a computer system which has many intelligent software applications, but which is controlled by an overwhelmingly powerful operating system, will not be a satisfactory solution. More democratic policies in the operating systems which would allow more autonomy for the subsystems and more freedom and flexibility of
communication among them will be necessary.

8.1.3. Data Base Management Systems

Data base management systems (DBMS) are a well-defined application area in computer science. In DBMS, important research issues like query languages, software implementation, hardware architecture, and data representation problems have been clearly characterized. The traditional DBMS usually supports only large data bases for industrial, commercial, or governmental applications. In general, the data base of a system should reflect the image of the system applications. No matter whether the application is information retrieval/manipulation (as in airline reservation systems), software systems, or diagnosis systems in Artificial Intelligence, the data base should always provide an integrated, up-to-date, consistent, convenient, and high level view of the information for users or tools of that application system. The research of information organization and information retrieval mechanisms in DBMS has had impact on the design of many large-quantity data bases in other application areas. For instance (as discussed in Chapter Six), the techniques of information retrieval from Form Tree Data Base in TRIAD are similar to those in DBMS.

It is extremely important for a data base to reflect the user's
views of the information in a problem domain. It has been an important research issue that a DBMS should permit the user to have different logical views of the information stored in the database in spite of its representation. Many existing DBMS can provide views independent of the physical implementation of the data structure, but cannot provide higher level views independent of its logical structure. This issue is also one of the most important research topics in software information databases. As discussed in Chapter Six, the tool TRIAD/SIE is designed to provide alternative views of all software-related information for different project members. It is true that the query functions provided by TRIAD/SIE are still very much representation-dependent. This is because the databases in the TRIAD prototype are implemented as hierarchical databases. Though they are more efficient, only limited queries can be expressed non-procedurally.

Research in relational databases has solved this problem to a certain degree. In a relational database, the queries can be like a very high level language which allows queries to be expressed by means of abstract relations. Since the relationships among different units of software information are much less understood, and the data structures are less uniform than those in DBMS, it will be more difficult to design a relational database for all software-related information. However, TRIAD/SIE also shows that the attributes
associated with each symbol, and the semantic functions associated with each production, can be used in a similar way to the relations and schemas used in relational data bases. Therefore, as the attributes and semantic functions become matured in a more specific methodology, it is possible to achieve non-procedural queries in a particular problem domain.

Currently, the logical data structure (i.e. the data unit, not the data base organization) used in most DBMS are still designed based on fairly low level data structures such as records or tables. Consequently, the descriptions of these data structures and the mechanisms for manipulating these data structures are too complex and can not reflect the high level views of the information. Efforts are being made to adapt the notion of data abstraction and design methodologies from software engineering to the data base environments [Hamm79]. This characterizes that a DBMS is also a complex software system. In other words, DBMS can be considered a well-defined problem domain in software development, in which the concepts of design methodology, tools design, and abstract data structure can be employed.
8.1.4. Artificial Intelligence

In contrast with the research in VHLL, Artificial Intelligence (AI) has been a revolutionary research area in computer science. The goal of AI is a qualitative expansion of computer capabilities. Most researchers in AI are more concerned with developing conceptually new approaches to using computers in new problem areas, rather than with increasing speed or lowering the cost of existing uses of computers [Duda79]. The range and scope of research issues in AI cover almost every application problem of computer uses. Some typical examples are in understanding continuous speech, chemical analysis, medical diagnosis, mathematical symbol manipulation, automated manufacturing, expert systems, computer vision, robots, natural languages, and automatic programming. The results of AI research are also fruitful. Many techniques have been developed in the areas of knowledge representation, control structures, heuristic search, planning, deduction rules, induction, and AI systems and languages. Many of the efforts mentioned above are attempts to understand the relationship between AI, psychology, and learning. Other efforts are more engineering-oriented and should benefit software technology more directly. Among them, the research in automatic programming probably has the most immediate impact on software engineering.
The advantages of automatic programming are clear. If part of the software development can be automated, then more efficient use of resources (time, man and machine) is possible over the software life-cycle. Though the complexity of system implementation will increase, the overall gain in the long run is evident. In an AI programming system, the programmer provides a high level specification of his/her program, which can be inconsistent or incomplete. The system utilizes the "knowledge" in that problem domain to make "intelligent" decisions in order to resolve the ambiguity in the specification. Then the specification is transformed into a functional specification and, with the approval of the programmer, transformed into a program (probably with some optimization to improve the efficiency). The work done so far in this area has only addressed the issue of small program synthesis. This is because the transformational process of a software (or program) from concept to code is still unstructured and imperfectly understood.

There has been evidence that the AI techniques adequate for problems of small or modest size are completely ineffective for problems of practical size (such as software development in general). This implies that more general methodologies and techniques for program or software development should be identified before they can be automated. The methodology-driven system, as
described in this thesis, is an alternative approach to achieving the same goal. The idea behind software methodology is to take a semi-automatic approach by breaking down the entire software development activity into different phases of the life-cycle, and then further breaking down each phase into manageable portions based on its underlying design or implementation concepts (as corresponding to the methodology-oriented concept trees in TRIAD). Moreover, by explicitly imposing syntactic structure on these concepts, we have a better chance to examine their patterns and behavior and to better understand the development process in a particular problem domain. Eventually, these concepts can be effectively algorithmized and automated. It is, therefore, important that AI and software engineering go hand in hand in this research area so that each can benefit from the other.

Currently, the AI programming systems are interested in only creating new programs, while research in software engineering also addresses other important issues such as maintaining, customizing and extending existing software. However, due to the versatility of techniques developed in AI research, there is a good possibility that software engineering can benefit from AI research much more than was suggested above. For example, the research in learning capability of AI systems should be useful for "methodology tuning" in TRIAD in the future. On the other hand, TRIAD will not only
improve the productivity and maintainability of software, but also help to identify the structures and methodologies of software development which can be eventually automated.

8.2. Research Concepts of TRIAD in Software Engineering

Research and development are two main streams in any discipline of application science. Systematic development results in the enhancement of existing technology in all respects and thus pushes the technology to its limit. In the meanwhile, researchers constantly try to invent revolutionary approaches which give themselves and others a new dimension of the technology. Therefore, both the "will of doubt" and the "ability of enhancement" of researchers guarantee that the wealth of science has no limit. It has been the case that many new approaches emerge simply because the researchers are able to see the light by studying the problems at a higher level of abstraction. Sometimes it might be necessary for a researcher to abandon all existing technology in order to envision a higher level of abstraction. Usually, a new approach can not be immediately as practical as existing ones. Researchers have to refine the abstract notion, through a long period of development, to promote the usability and feasibility of the new approach. Many ingenious new ideas remain undeveloped because they did not receive deserved recognition.
In the previous section, we discussed the TRIAD project and its relationship to other research areas from the viewpoint of development techniques. In this section, we will discuss the approach presented in this thesis from the viewpoint of research concepts in the context of software engineering.

In programming languages, many well-known abstraction techniques have been successfully applied to the design of a high level language. Some examples are the subroutine and function invocations which are abstractions of control structures; macro expansions which are abstractions of both syntactic and semantic structure; programming methodologies like Jackson's methodology or top-down design which are abstractions of program design; many other language constructs such as recursion and case statements which are abstractions of control branching in a machine; and finally, the data abstractions. In operating systems, virtual memory is the abstraction of the auxiliary memory in disks or drums; processes are the abstractions of the program execution in the CPU in a time-sharing system; and the operating system itself is the abstraction of the computer architecture. In DBMS, the key fields of data records in a hierarchical data base are the abstractions of the data base; the relations in a relational data base are the abstractions of the data base; and the queries are the abstractions of the schemas which perform the actual retrieval or updating. In
Artificial Intelligence, all representations of knowledge are based on several levels of abstractions (such as Frames and Plans), and all AI techniques of using this knowledge are also based on these multi-leveled abstractions via gradual transformation (or refinement). This long list of examples illustrates that abstraction techniques are indeed one of the most promising conceptual tools in research and development. The more we can abstract the underlying concepts, the better opportunity we have to solve a problem.

Software life-cycle is an important abstraction of the general activity of software development. It helps researchers to describe and understand the general concepts of software development. However, there is a huge gap between the life-cycle concept and the current software technology. That the concept of life-cycle in software engineering can not be used as effectively as in hardware-engineering or architecture is mainly due to the fact that software development involves many management-oriented issues. Another reason is that the quality of software is not as physical as that in hardware and architecture. That is, unlike constructing a building or a bridge, it is difficult to "see" and "test" the design of a software. A software should be able to perform a variety of functions and evolve, unlike a hardware product which can only perform a set of specific functions and can not evolve. These unique
characteristics make software development difficult.

In relation to all research concepts in software engineering, the concept of software life-cycle is at one end of this continuum, and the traditional programming concepts are at the other end. In between these two ends are many efforts to identify the research concepts in issues of software management, software quality, software methodologies, testing and verification, optimization and enhancement, specification languages, and various software tools. These efforts usually concentrate on one phase of the life-cycle and eventually make the concept of life-cycle more concrete. In other words, research in software engineering has been, and should be, an evolutionary process to build up more levels of abstractions between the concept of life-cycle and the current software technology.

There is, however, a lack of computer-aided environments to support these useful concepts or techniques. As illustrated in earlier chapters, these concepts and techniques can be encoded as TRIAD's methodologies which can evolve along with experience. That is, TRIAD has been designed as an adaptable software environment to support these concepts and techniques in software life-cycle. Therefore, TRIAD is a useful application environment for software development and maintenance.

From the system's point of view, the Methodology Library
represents the abstract knowledge of the environment and the generic tools in TRIAD which are able to manipulate information at the conceptual level. In this way, the tool Tuner can create, specialize and generalize a methodology to customize the environment in a uniform and consistent manner. From the user's point of view, a software methodology in TRIAD first identifies the abstract concepts of software design in a particular problem domain, and then suggests some refinement techniques to solve the problem. In the beginning, the refinement techniques suggested are likely to be incomplete or inefficient when the problem domain is not fully understood. However, TRIAD allows a methodology to evolve. A methodology can thus become more and more intelligent and eventually will bridge the gap between the highest level of abstraction and its implementation (see the section, "Automatic Methodological Support," below). From the methodology designer's point of view, the development history of each software project using a methodology is also maintained in TRIAD. Therefore, the logical quality of the software is made more visible to the designer. Since TRIAD uses the concept attribute grammar to encode a methodology, it allows the designer to identify the underlying concepts of similar software design at a more abstract level (instead of at the level of programs and documentation) than their development history. These three views together illustrate that TRIAD is not only an application
environment, but also a research environment.

8.3. The Impact of Methodology-Driven Systems on Programming

It was a well-known lesson of the 70's that programmer's programming habits must be changed because the costs of maintaining an unstructured program are much higher than those of writing a new program. Therefore, many programming methodologies have been proposed and programmers have been taught to write only "structured programs". The notion of "structured programming" (or the metaphor of "Eliminating Goto" [Dijk68]) has become the new trend of programming in the 80's. Though many programmers have to sacrifice their freedom to create "smart" programs, the overall cost is reduced.

It is also likely that methodology-driven systems and automatic programming systems, in addition to structured programming, will have similar impact on current programming habits. Especially when the costs of an application system mainly come from the design (where methodologies and automatic design are important) and from software maintenance and enhancement (where methodologies and standardized structures are important), the activity of software development can no longer be considered as simply creating a collection of structured programs.
Programmers and software designers will probably have to sacrifice more freedom to create "smart" software (like most of the existing software). In fact, it will probably be necessary for programmers or software designers to create only the kind of programs or software which can be supported by methodologies (as discussed in this thesis) or automated by certain AI techniques. Both the activities of programming-in-the-small and programming-in-the-large will be equally affected. However, this is not to say that current programming techniques are incorrect or unimportant. Rather, this simply means that new programming concepts and techniques have to emerge such that the overall software development can be more practical and cost-effective. One of the most important responsibilities of the TRIAD prototype, in the near future, is to identify such changes in programming habits.

8.4. Future Research Directions

8.4.1. Automatic Methodological Support

Currently, computer-aided methodological support for software development and maintenance is not available. This thesis provides a uniform approach to support methodologies in a software environment. Most of the desirable characteristics of software methodologies have been captured through the model of attribute grammar forms, form-based uniform interface, and the tool Tuner to tune
methodologies.

Although such methodological support in a software environment can be very effective in a more specific problem domain, it can only have a marginal value in a more general problem domain. This difficulty comes from the conflict between two goals: the *generality* and the *effectiveness* of a methodology. It has been shown in Chapter Four that a methodology as specific as Transaction Processing Methodology can be fully automated, while in a more general problem domain, a methodology and tools provided in a software environment is much less effective. This is analogous to providing an assembly language programmer with a high level language (without compiler) and tools to generate a symbol table and cross reference, but requiring that the programmer still generate the assembly code. Though this might sound ridiculous at present, such a transition period is necessary (at least at the conceptual level) before the compiler for a high level language can be fully automated. Automating a general methodology will be like automating the compilation process for a general-purpose high level language. However, in order to substantially reduce the software cost, it is necessary to provide automatic methodological support in a software environment in the future. It will be easier to automate a more specific methodology, and the techniques of automating specific methodologies will eventually lead to the automation of more general
methodologies.

8.4.2. Software Maintenance Methodologies and Tools

Most of the current research in software methodologies (including this thesis) focuses only on design methodologies. In the case of hardware-engineering, most of the hardware products are basically "black boxes" to the user. The maintenance cost of a hardware product is usually a constant or is proportional to its size, while software products usually need to be modified for customization, enhancement, and extension, or they may have errors which need to be corrected. Therefore, the maintenance cost of a software is a result of its functionality rather than its size. A software environment should also provide methodological support for software maintenance in addition to software design. Another important and related issue is the need for maintenance tools in a software environment. In one sense, the purpose of a maintenance tool is similar to that of a language debugger to programs. The difficulty of designing such tools is that software-related information is not structured and the notation used to document this information is not formalized. There are efforts being made to formalize the notations for requirements specifications and design specifications such as SADT [Ross77] and FSL/FSA [Teic77], but more research needs to be done in this area.
The structuring of all software-related information, as presented in this thesis, maintains the entire development history of a software project. Furthermore, it standardizes the project structure as well as the documentation. The task of maintaining a software can thus be simplified by taking advantage of the presence of its development history and standardized structure and documentation. A maintenance tool called Analyzer is currently being designed in the TRIAD project. This Analyzer is to provide support for version control, consistency checking (for both the content and the structure of a form tree), data dependency (in order to identify the global effect of a modification), and the relationships between different form trees. This Analyzer can also utilize the information structure provided by a methodology and make use of associated attributes and semantic functions to generate useful information for the user. However, the notation used to interpret the organizers of a general methodology is not yet formalized in TRIAD.

8.4.3. Software Adaptability

There are some classical techniques, such as macro expansion and parameterization, to improve the software adaptability. The former one allows the user to define his/her application-specific functions at the time of command language definition. As a matter of
fact, should the command language be interpretable, macros can also be defined during the execution time (such as Mini-LISP in EMACS). The latter one allows the user to customize a generic function to his/her needs. In an interactive software system, the user can supply appropriate parameters to a generic function in order to adapt his/her specific applications. A generalized technique based on parameterization is the subroutine library. A software system with such a subroutine library allows the user to select an appropriate subroutine and plug it into his/her module at the time of compilation or execution.

As discussed in Chapter Seven, the above techniques have been incorporated into the design of the C-prototype. The adaptability of the TRIAD prototype is again highlighted below:

- The extensible User-Interface which facilitates user-defined commands in terms of either macros or Mini-LISP,

- The methodology-driven tool-kit which can be customized by a specific methodology to adapt the user’s needs, and

- The evolvable methodologies in which more "intelligence" can be embedded using the Tuner.

Since the command language provided in the C-prototype is fairly simple, many useful extension and customization still have to
be done at the level of implementation language. From the user's point of view, this seriously restricts the adaptability of this prototype. In order to make the TRIAD prototype more adaptable, one has to pull up all the useful generic functions (defined in each tool) to the level of Mini-LISP so that the user can modify existing functions or define new functions using Mini-LISP. Furthermore, it is necessary to design a command extension language as mentioned in the last section of the previous chapter. However, a Mini-LISP-like command extension language will also soon reach its limit. Therefore, in the long run, an extensible language will be the key to a truly adaptable software system. More research in this area is required.

It is the author's opinion that, as soon as a simple extensible language becomes available, the process of software adaptation and maintenance can get arbitrarily complex. In spite of the fact that the extensible language itself can get arbitrarily complex, the techniques of software adaptation will become required lessons for all the software designers and the software users. It can also be predicted that, along this progress, there will be "computer school" where the "computer professors" will be teaching personal computers how to adapt their owners' needs in the foreseeable future.
BIBLIOGRAPHY


Boch74 BOCHMANN, G.V. "Attribute Grammars and Compilation: Program Evaluation in Several Phases", 1974, Document de Travail #54, Department d'Informatique, Universite de Montreal, Montreal, Aout

If you don't find it in the index, look very carefully through the entire catalogue.

-- Consumer's Guide, Sears Co. (1897)


Gos180 GOSLING, J. Implementation of Emacs in C, 1980,


STALLMAN, R. "EMACS, The Extensible, Customizable, Self-Documenting Display Editor", June 1979, AI Memo 519, Artificial Intelligence Laboratory, MIT, Cambridge, Mass.


I don't want to talk grammar. I want to talk like a lady.

— G. B. Shaw

**Grammar Form Definition**

A context-free grammar $G = (V, N, S, P)$ is a four-tuple, where

1. $V$ is a finite set of vocabulary,

2. $N$ is a finite set of non-terminal vocabulary and $N$ is contained in $V$,

3. $S$ is the starting symbol contained in $N$, and

4. $P$ is a finite set of production rules of the form $X \rightarrow a$, where $X$ is in $N$ and $a$ is in $V^*$.

Since $G$ has a finite vocabulary it is, by itself, inadequate for modeling a program derivation tree whose nodes may be labeled by strings belonging to an infinite set — the design language. Hence we augment the context-free grammar as shown below. A grammar form is composed of a concept grammar that is a "template" for other grammars, called the interpretation grammars, that have similar
form. The interpretation grammars are supplied by the programmer by responding to prompts of the concept grammar symbols in some language.

More formally, a (context-free) grammar form \( G \) is a four-tuple \((G, M, V, S)\) where

1. \( G = (V,N,S,P) \) is a context-free grammar called the concept grammar,

2. \( M \) is an infinite set of substitutions, and

3. \( V \) and \( S \) are infinite vocabularies with \( S \) in \( V \).

For each \( u \) in \( M \),

1. \( u(a) \) is a finite set in \( S^* \), \( a \) in \( V - N \),

2. \( u(X) \) is a finite set in \( V - S \), \( X \) in \( N \), and

3. if \( X, Y \) are in \( N \) and \( X \neq Y \), then \( u(X) \) and \( u(Y) \) are disjoint.

We will follow the convention that a grammar form will simply be referred to by its underlying concept grammar \( G \), while \( M, V \) and \( S \) are understood.

An interpretation grammar \( G(I) = (V(I), N(I), S(I), P(I)) \) of \( G \) is a grammar such that

1. \( u(V) = V(I) \),
2. \( u(N) = N(I) \),

3. \( u(V-N) = V(I) - N(I) \),

4. \( P(I) \) is contained in \( \{u(X) \rightarrow u(a) \mid X \rightarrow a \text{ in } P\} \), and

5. \( S(I) \) is contained in \( u(S) \).

In order to develop a program the programmer must use the productions of the concept grammar and interpret them using the vocabularies in \( V \) and \( S \). The derivation sequence constructed using the concept grammar productions will define the concept tree. A production can also be viewed as a single-level concept tree. The corresponding tree defined using the interpretation grammar will be called the refinement tree. There are infinitely many interpretation functions \( u \), therefore, corresponding to each concept tree there are infinite set of refinement trees each of which is simply a relabeled version of the concept tree. It is obvious that the concept tree provides the abstract structure for all the corresponding refinement trees, and the infinite set of substitutions allows one to record all software related information (not necessary program related information) into such structure.

There are three important but subtle distinctions between grammars and grammar forms:

1. The notion of interpretation permits a grammar form to
provide high level and language independent syntactic concepts. For example, the following production

\[
\text{<program>} : = \text{<input/output specification>} \text{<initial statement>}
\text{<program body>} \text{<final statement>}
\]

suggests that a program must be composed of distinct initial, body, final parts, and must be preceded by input/output specification. This concept can be eventually interpreted into any programming language as comments and language syntax like type declaration, begin-end block, etc. But, no existing programming languages can enforce such concept using their syntax. By repeatedly using such concepts, a user can interpret the concept using some notation (not necessary programming language) to develop a refinement tree that would also bear the same concepts.

2. A pre-defined concept tree reflects a plan, a pattern, or a template which can be used in the development of software projects or programs. By using a concept tree as a whole, we can suggest the user good patterns which are known to be useful in similar problems. In programming languages, we can also find some very low level patterns. For instance, iteration can be written in several different ways in a language and each one of them can be thought of as trying to encode certain patterns of iteration. Grammar forms can
extensively encode useful patterns for much higher level concepts and suggest them in a much more direct and explicit manner than can the programming languages.

3. The interpretation of a concept grammar can be repeated. That is, an interpretation grammar can again become a more specific concept grammar, the substitution rules can also become more and more specific. As experience grows, a more complete pattern is possible. Hence, while the concept grammar "evolves", less and less work is required from the user. In most programming languages, the abstract concepts and the concrete syntax are indistinguishable and all buried inside the compiler. Since the only way for a language to enforce a concept is through its concrete syntax, evolution of the abstract concepts is equivalent to rewriting of the compiler.

However, the structure and patterns alone are by no means the methodology yet. One of the most important characteristics of methodologies, in guiding the user's thought process, is still missing. An extended model, called the attribute grammar forms, which can be used to facilitate this (and many other advantages) is described in next section.
As in the definition of a grammar form which is derived primarily based on its underlying concept grammar, an attribute grammar form is defined based on its underlying concept attribute grammar. Since the notion of attribute grammars is not as well known as context-free grammars, we will first give a definition [Knut74, Raih77] of the attribute grammars and then the attribute grammar forms.

Definition of Attribute Grammars

An attribute grammar AG is a four-tuple \( AG = \{G, A, F, R\} \), where

1. \( G = \{V, N, S, P\} \) is a context-free grammar,

2. \( A = \{\text{Syn}, \text{Inh}, \text{Ran}, \text{Val}\} \) is a specification of attributes,

3. \( F \) is an attribute associator function for \( G \) and \( A \), and

4. \( R \) is a set of collections of semantic rules.

A specification of attributes \( A \) for a grammar \( G \) is a four-tuple \( A = \{\text{Syn}, \text{Inh}, \text{Ran}, \text{Val}\} \), where

1. \( \text{Syn} \) is a finite set of synthesized attributes, disjoint from \( \text{Val} \).
2. Inh is a finite set of inherited attributes, disjoint from Val and Syn.

3. Ran is a collection of sets of allowed attribute values.

4. Val is a mapping from Syn and Inh to Ran.

The set Syn U Inh is denoted by A and called the set of attributes. That is, A includes both the synthesized and inherited attributes. For each attribute a in A, Val(a) specifies the set of values in Ran that any instance of attribute a may assume.

To each symbol x in V, we associate a finite set F(x) of attributes. This set F(x) is partitioned into two disjoint sets, the synthesized attributes Fs(x) and the inherited attributes Fi(x). We also require Fi(S) to be empty (that is, the starting symbol S has no inherited attributes) and Fs(x) to be empty if x is in V-N (that is, the terminal symbols have no inherited attributes). The mapping, from V to the power set of A, which satisfies the above conditions is called the attribute associator for G and A.

Let 
\[ P: x_0 := x_1 x_2 \ldots x_n \]
be a production of G. We say P has an attribute occurrence \((a, j)\) if a is an attribute in \(F(x_j)\) and \(0 \leq j \leq n\). That is, a is an attribute belonging to the set of attributes associated with the \(j\)-th symbol
in the production P. A **semantic rule** for an attribute occurrence \((a, j)\) in production \(P\) is a function 
\[ r : \text{Val}(b_1) \times \text{Val}(b_2) \times \ldots \times \text{Val}(b_n) \rightarrow \text{Val}(a) \]
, where each \((b_i, i)\) is an attribute occurrence of \(P\). In other words, each semantic rule maps values of certain attributes of \(x_1, x_2, \ldots, x_n\) into the value of some attribute of \(x_j\). We require that each attribute occurrence of a production \(P\) must have one and only one semantic rule. Each production thus has a collection of semantic rules for all of its attribute occurrences. The set of collections of semantic rules for all productions is denoted by \(R\).

The usefulness of attribute grammar forms are as follows:

1. Grammar forms can be used to provide the structure for the final products of software development, as well as, encode higher level concepts, good patterns, or experience in developing similar projects.

2. Attributes and semantic rules can be used to reinforce the encoding of concepts, patterns, etc. and provide suggestions and feedback at the evaluation time.

3. A semi-automatic algorithm of evaluating attributes (similar to those described above for attribute grammars) can be used as a general strategy or guidelines for software development based on the information provided by 1 and 2 above.
It should be clear at this point that methodologies require that, if not more, all the features facilitated by both grammar forms and attribute grammars to be present. This cause us to merge the grammar forms and attribute grammars into an extended model called the attribute grammar forms. The formal treatment of this extended model and some of its formal properties are described in [Soni83]. In this section, author shall use it only for the purpose of obtaining a working definition of software methodologies. Potential applications of this model and its formal properties will not be discussed in this thesis.

**Working Definition:**

A software methodology is a four-tuple \( M = \{ \text{AGF}, \ C, \ L, \ E \} \),

where

1. \( \text{AGF} = \{ \text{AG}, \ldots \} \) is an attribute grammar form,

2. \( C \) is a set of concept trees defined using the concept attribute grammar \( AG \),

3. \( L \) is a set of interpreted trees derived from the interpreted attribute grammar, and

4. \( E \) is a set of functions, each of which maps the semantic
evaluator for the concept attribute grammar to the semantic evaluator for each interpreted attribute grammar.
APPENDIX B. AN ANNOTATED EXAMPLE

Examples are important. A farmer who has never seen a ship in the ocean will be helped if told that the ship plows the water as it sails through it, for example.

-- M. F. Rubinstein

It is the purpose of this appendix to demonstrate the power of TRIAD/SIE by using a comprehensive example of project development. This example is taken from the design and implementation of the database FTDB which is part of the prototype itself. We will firstly show how the editing commands are used to construct the form trees for these two databases using the Project Development Methodology given in Chapter 5. Secondly, we will compare the project related information thus structured with the original documentation and programs which were kept as Emacs text files. Thirdly, we will show how the TRIAD commands can be used to retrieve the information in a variety of ways to support high level views, different views, etc. which are useful for both information development and maintenance.

As for the format of presenting this example, we will first show the user's command, the effect of that command as displayed on the screen, and annotations to explain the purpose of that command
as well as other important actions behind the display. In order to make this example more readable, we will not worry about the size of the physical screen. Before we start the example, readers should be reminded that there are three levels of editing functions. The commands start with the control key "^" are usually working at the text (entry) level, the commands start with "~y" are working at the form level, and those start with "~z" are working at the form tree level. Commands start with "M-X" are extended commands which can be either generic or methodology-dependent. Finally, the cursor position is displayed as the underline "_".
1. Login the TRIAD Environment

1.A Command: TRIAD

1.B Display:

TRIAD:DEMO.OSU.VERSION-1(11/11/82)

TRIAD_DEMO is the prototype of the TRIAD System designed and implemented by the TRIAD Research Group at OSU.

<table>
<thead>
<tr>
<th>Methodology</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM:Transaction</td>
<td>for developing simple file transaction programs</td>
</tr>
<tr>
<td>Processing</td>
<td></td>
</tr>
<tr>
<td>Methodology</td>
<td></td>
</tr>
<tr>
<td>DA: Data</td>
<td>for developing programs using data abstractions</td>
</tr>
<tr>
<td>Abstraction</td>
<td></td>
</tr>
<tr>
<td>Methodology</td>
<td></td>
</tr>
<tr>
<td>PDM:Project</td>
<td>for developing larger scale project</td>
</tr>
<tr>
<td>Development</td>
<td>through the whole software life cycle</td>
</tr>
<tr>
<td>Methodology</td>
<td></td>
</tr>
</tbody>
</table>

To select a Methodology, type in its short name.
Methodology Name: PDM

1.C Annotation:

"TRIAD" is the login command to the TRIAD environment from the top level monitor (say, from the operating system). The system responds to this command by displaying a set of methodologies supported by TRIAD and prompts the user to select a methodology. As soon as the user types in the methodology name, say, PDM, all methodology-related system routines, such as the blank forms, help information, extended commands defined for PDM, patterns, etc., are loaded from the Methodology Library. The command "TRIAD PDM" will have the same effect except that the above display is omitted.
2. Create or Visit a Form Tree

2.A Command: M-X FT (Form Tree Name:) FTDB

2.B Display:

TRIAD (Methodology:PDM) Buffer: (Form-Tree) FTDB [ ]
M-X FT (Form tree name:) FTDB
(New form tree)

2.C Annotation:

The string "Form tree name:" within the parentheses in the command line is the system's prompt after the user types in "FT" followed by the "Esc" (escape) key. After the form tree name is entered, the system will respond in one of the following two ways: If FTDB is a new form tree, the screen will be blank and a message "New form tree" is displayed on the message line; if FTDB is an old form tree, then the root form is displayed. The first line in the bottom region is the mode line which displays the system status, the second line is the command line and the rest of the area is the message lines. In order to save the space, we will not repeat this bottom region in the display of the following commands.
3. Select a Starting Blank Form

3.A Command: ^Z-START (Form #:) 1

3.B Display:

<table>
<thead>
<tr>
<th>PDM-Form-1 Project Form-use-#[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project Name:</td>
</tr>
<tr>
<td>Project Description</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Project Goals:</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Project Manager:</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Project Participants:</td>
</tr>
<tr>
<td>(More?):</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Project Proposal</td>
</tr>
<tr>
<td>Proposer:</td>
</tr>
<tr>
<td>Abstract:</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>[2]Proposal Details Form-use-#[]</td>
</tr>
<tr>
<td>Project Review</td>
</tr>
<tr>
<td>Reviewer:</td>
</tr>
<tr>
<td>Date:</td>
</tr>
<tr>
<td>Comments:</td>
</tr>
</tbody>
</table>

3.C Annotation:

^Z-START is the command to select a starting blank form as the root form for a new form tree. If the user can not remember which starting blank form he wants, he can enter ^Z-START followed by the "Return" key, then the system will display all the starting blank forms with brief descriptions and ask the user to make the selection. Note that the cursor is placed at the first blank character of the first entry and ready to be edited.
4. Edit Entries in a Form

4.A Command: (Text editing and form editing commands)

4.B Display:

<table>
<thead>
<tr>
<th>Project Name: Form Tree Data Base</th>
</tr>
</thead>
</table>

Project Description

- Project Goals: Design and implementation of the Form Tree Data Base which organizes the software related information into form trees according to blank forms defined in a methodology description.

- Project Manager: Jay Ramanathan

- Project Participants: Jeremy Kuo
- Project Participants: Chin Li

Project Proposal

- Proposer: Jeremy Kuo

Abstract: This database will be designed as a hierarchy of data abstractions based on the attribute grammar form model. This hierarchy contains data structures SET_OF_FORM_TREE, FORM_TREE, FILL_FORM, NODE and BLANK_FORM defined in MDDDB.

(2) Proposal Details

Project Review

- Reviewer: 
- Date: 

Comments:

4.C Annotation:

Emacs-like text editing commands are used to fill in the entry for each organizer. Form editing commands such as "Y"-"N (move to next organizer), "Y"-"P (move to previous organizer), "Y"-"M ("more" for a repetition organizer), "Y"-"K (erase an entry) etc. are used to edit a form. The display above shows a partially filled-in form of PDM-Form-1.
5. Refine an Organizer Using a Blank Form

5.A Command:  `Y~F`

5.B. Display:

<table>
<thead>
<tr>
<th>PDM-Form-2 Proposal Details Form-use-#[11]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concept Communication</td>
</tr>
<tr>
<td>Communicate w/Manager: _</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Communicate w/Programmer:</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Project Plan</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>General strategy:</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Special strategy:</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Preparation</td>
</tr>
<tr>
<td>(4)Meetings Form-use-#: [ ]</td>
</tr>
<tr>
<td>(More?):</td>
</tr>
<tr>
<td>(5)Presentations Form-use-#: [ ]</td>
</tr>
<tr>
<td>(More?):</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>(6)Resource Required: Form-use-#: [ ]</td>
</tr>
<tr>
<td>(3)Project Design: Form-use-#: [ ]</td>
</tr>
</tbody>
</table>

5.C Annotation:

The command `Y~F` is to refine a non-terminal organizer in the current form. The organizer should also be the current organizer. That is, the organizer where the cursor resides. The cursor in the above display is at the end of the organizer Proposal Details and since this organizer has only one blank form to refine it (i.e. PDM-Form-2), no blank form number need to be specified. If there is more than one blank forms can be used for refinement, the system will prompt the user with (Using:) after `Y~F`. The Form-use-# specified in that organizer is also copied to the new blank form.
6. Help Information

6.A Command: M-? PDM FORMS

6.B Display:

Project Development Methodology (PDM)

The following blank forms are defined in the PDM.
Form-1, -7, -10 can be the starting blank forms. Use command M-? PDM FORMS # to display the help information of each form.

Form-1 Project
Form-2 Proposal Details
Form-3 Project Design
Form-4 Meeting
Form-5 Presentation
Form-6 Resource Required
Form-7 Functional Specification
Form-8 Control Algorithm Specification
Form-9 Function Module
Form-10 Data Abstraction Module
Form-11 Implementation
Form-12 Function
Form-13 Simple Data Type
Form-14 Operation

Pattern-1 DA-LIST-OF-ITEM
Pattern-2 DA-STACK-OF-ITEM

6.C Annotation:

All the help commands are preceded by the "M-?" key. The argument list is used to identify the help information in the hierarchy. For example, "M-? COMMAND "Y-"F" will cause the help information of the command "Y-"F to be displayed. Similarly, "M-? PDM FORMS 10" will cause the help information of PDM-Form-10 to be displayed, and so forth. Note that there are two patterns for designing data abstractions of generic singly-linked lists and stacks are available in this methodology.
Now, suppose that we are at the point to specify the data abstraction of a singly-linked list of tree nodes LIST_NODE, and we would like to use the pattern DA-LIST-OF-ITEM. This pattern is actually a form tree in which many entries are pre-filled. The following two commands show how this pattern is used to obtain the data abstraction of LIST_NODE. The command 7.A displays the form tree structure of DA-LIST-OF-ITEM, and the command 8.A uses this pattern to refine the organizer

**Data Abstraction:** LIST-NODE Form-use-#[233].

The form tree DA-LIST-OF-ITEM contains the specification of the list structure, useful operations, and even the implementation of some operations. It is actually a singly-linked list structure of pointers to another structure ITEM. The user of this pattern can change the pointers to other data structure and modify only the code which access to the internal structure. For instance, the organizer **Lower level data abstractions** has to be refined by the user, the declaration of ITEM at the beginning of the implementation of each operation should be replaced by a pointer points to this new structure. The help information (through the command "M-? PDM PATTERN 1") of this pattern clearly specifies what organizers have to be interpreted by the user, as well as what implementation code of each operation has to be modified.
7. Display a Form Tree Structure

7.A Command: \[^{Z-^T}\] (Form tree name:) DA-LIST-OF-ITEM
7.B Display:

```
Form tree structure
|
Data Abstraction Module: LIST-OF-ITEM [1]
Lower level data abstraction
  Simple data type: [ ]
  Implementation [111]
Operation: HeaderNodeAlloc [12]
  Implementation [121]
Operation: CreateListItem [13]
  Implementation [131]
Operation: KillListItem [14]
  Implementation [141]
Operation: IsEmptyListItem [15]
  Implementation [151]
Operation: SizeOfListItem [16]
  Implementation [161]
Operation: FirstItemListItem [17]
  Implementation [171]
Operation: NextItemListItem [18]
  Implementation [181]
Operation: MoveRightListItem [19]
  Implementation [191]
Operation: GoBeginListItem [20]
  Implementation [201]
Operation: GoEndListItem [21]
  Implementation [211]
Operation: InsertListItem [22]
  Implementation [221]
Operation: DeleteListItem [23]
  Implementation [231]
Operation: DeleteNextListItem [24]
  Implementation [241]
Operation: FindListItem [25]
  Implementation [251]
```

7.C Annotation:

The command \[^{Z-^T}\] displays the form tree structure of any exiting form tree. The tree structure shows the form heading and Form-use-\# of each form in the form tree. Note that an un-refined organizer has a blank Form-use-\#. 

8. Refine an Organizer Using a Pattern

8.A Command: \texttt{~Y~F \textsc{pattern-1}}

8.B Display:

```
| Form-10 Data Abstraction Module Form-use-\#[2331] |
| Data abstraction name: LIST-NODE |

Purpose of this module: |

Lower level data abstractions
(10,13) Data abstraction module: NODE Form-use-\#:\#[2331]

(More?):

{14} Operation name: ItemNodeAlloc Form-use-\#:[23311]  
{14} Operation name: HeaderNodeAlloc Form-use-\#:\#[23312]  
{14} Operation name: CreateListNode Form-use-\#:\#[23313]  
{14} Operation name: KillListNode Form-use-\#:\#[23314]  
{14} Operation name: IsEmptyListNode Form-use-\#:\#[23315]  
{14} Operation name: SizeOfListNode Form-use-\#:\#[23316]  
{14} Operation name: FirstNodeListNode Form-use-\#:\#[23317]  
{14} Operation name: NextNodeListNode Form-use-\#:\#[23318]  
{14} Operation name: MoveRightListNode Form-use-\#:\#[23319]  
{14} Operation name: GoBeginListNode Form-use-\#:\#[23320]  
{14} Operation name: GoEndListNode Form-use-\#:\#[23321]  
{14} Operation name: InsrNodeListNode Form-use-\#:\#[23322]  
{14} Operation name: DelNextListNode Form-use-\#:\#[23323]  
{14} Operation name: DelNodeListNode Form-use-\#:\#[23324]  
{14} Operation name: FindNodeListNode Form-use-\#:\#[23325]  
(More?):
```

8.C Annotation:

This command is the same as the command 5.A except that a pattern is used to refine the current organizer instead of using a blank form. The root form of the pattern is displayed, and the Form-use-\# specified in the current organizer becomes the prefix of all Form-use-\#'s in the pattern. Note that the lower level data abstraction name "ITEM" is replaced by "NODE". The user should simply follow the instructions given in the help information to obtain a correct form tree for the data abstraction LIST-NODE. More operations can be added or some operations can be deleted if desired.
9. Visit a Form

9.A Command: \(^{Z^-}{^V}\) (Form-use-#: 23318)

9.B Display:

```
<table>
<thead>
<tr>
<th>FORM-14 Operation Form-use-#: [23318]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation name: NextNodeListNode(list)</td>
</tr>
<tr>
<td>Purpose of this operation: get a pointer to the next node.</td>
</tr>
<tr>
<td>Input type: a pointer to a LIST_NODE</td>
</tr>
<tr>
<td>Output type: a pointer to the next tree node, if the next node at the tail node, a null pointer is returned.</td>
</tr>
<tr>
<td>External operations used:</td>
</tr>
<tr>
<td>NextItemListProc()</td>
</tr>
<tr>
<td>Side effect: none</td>
</tr>
<tr>
<td>Pseudo code:</td>
</tr>
<tr>
<td>get next item node from the list;</td>
</tr>
<tr>
<td>/* use NextItemListProc() and convert the pointer */</td>
</tr>
<tr>
<td>if (at the end of list)</td>
</tr>
<tr>
<td>return null pointer;</td>
</tr>
<tr>
<td>else</td>
</tr>
<tr>
<td>return the pointer to the structure in that item-node;</td>
</tr>
</tbody>
</table>
```

9.C Annotation:

The command \(^{Z^-}{^V}\) can visit a form in the current form tree by its Form-use-#. There are other commands which can also visit a form in different ways. For example,

\(^{Z^-}{^N}\) (visit the next form in in-order),
\(^{Z^-}{^P}\) (visit the previous form in in-order),
\(^{Z^-}{^A}\) (visit the root form), etc.

The user can also use multiple windows to display more than one form on the screen at one time.
Improvement of the Project Related Information Structure

The original documentation and programs for the design and implementation of FTDB and MDDB were kept in about 30 text files which were all edited using Emacs. The design and implementation of these two databases were the main responsibility of a sub-project of the entire C-prototype project. The documentation and programs written in C were accumulated up to more than 500 pages. These documentation and program files were manually structured as a rough hierarchy. During the design, coding, and maintenance phases of this sub-project, there have been many problems encountered by different project members. Some of them are described below.

Problems

1. The structure of all project related information was manually constructed and maintained, thus, part of the documentation was to document the documentation itself. For example, the relationship between different files, the content of each file, the structure of the content, etc. were all manually maintained by the programmer.

2. Programmers, including author, have had very often experienced in scanning through the entire documentation, line by line and word by word, just in order to maintain the
consistency of the documentation after each design modification.

3. The project manager, different programmers, and new project members have been having difficulties in understanding this 500-page-thick documentation and code. The difficulties were mainly due to the fact that they all had different perception of this project, and it was impossible for them to digest the entire documentation. Therefore, the programmer had to manually extract information from the documentation in order to produce higher level documentation for different project members.

4. The situation was even worse since the entire project was in concurrent development by three sub-projects. Other project members had to be informed by the up-to-date documentation of each sub-project. Therefore, such higher level documentation had to be reproduced again and again whenever any modification was made. Such a time-consuming and annoying task has become the most-hated burden on each project member.
Improvement

As we have shown earlier in this appendix, all the project related information can be structured into form trees according to the Project Development Methodology (PDM). Though the entire documentation in Emacs files has not yet been completely transferred to the form tree structure, we estimate that there will be no less than 1000 forms in the form trees of both FTDB and MDDB. However, this is no concern of the user because TRIAD does most of the work and maintain the structure for the user automatically. As illustrated earlier, the structure of the entire project related information is explicitly provided by the blank forms of the methodology, thus the programmer no longer has to construct the information structure, nor manually maintain this structure. Also, the placement, content and notation of the documentation are standardized to a large degree, thus useful patterns can be used to develop similar modules.

In addition to the advantages mentioned above, we will next show how the TRIAD commands can support a variety of information retrieval such that useful views of the entire information can be easily obtained instead of manually producing different documentations for such views.
10. Collect Programming Code from a Form Tree

10.a Command: M-X CODE (Form tree:) LIST-NODE
(Output file:) LIST-NODE-CODE

10.b Display:

```
#include "listch.h"
#include "node.h"
#include "symbol.h"
#include "bform.h"
#include "prod.h"

extern struct ITEM_NODE TAIL_ITEM_NODE;
extern struct ITEM_NODE *NULL_ITEM_NODE;
extern char *NULL_TEXT;
struct NODE *NULL_NODE = 0;
LIST_NODE NULL_LIST_NODE = 0;

LIST_NODE CreateListNode()
{
    LIST_NODE new_list;
    new_list = (LIST_NODE) CreateListChar();
    return(new_list);
}

KillListNode(list)
LIST_NODE list;
{
    KillListChar((LIST_CHAR) list);
}

#define IsEmptyListNode(list) IsEmptyListChar((LIST_CHAR) list)
#define SizeOfListNode(list) SizeOfListChar((LIST_CHAR) list)
...
```

10.c Annotation:

The command "M-X CODE" is an extended command defined in PDM, which will traverse the given form tree in in-order and collect the code in all the Form-ll Implementation into the output file. Usually, if the code is collected for the entire form tree, it may not be in the format ready for compilation. Therefore, the user is encouraged to collect the code locally and then put them together in proper order (maybe by adding some information needed for linking) such that it is ready for compilation.
### 11. Generate Report for High Level Views

**11.A Command:** M-X OPERATION LIST (DA name:) LIST-NODE  
(Output file:) LIST-NODE-OP-PURPOSE

**11.B Display:**

<table>
<thead>
<tr>
<th>Data Abstraction Module: LIST_NODE</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>CreateListNode()</td>
<td>Create and initialize a LIST_NODE</td>
</tr>
<tr>
<td>KillListNode(list)</td>
<td>Kill a list of nodes</td>
</tr>
<tr>
<td>IsEmptyListNode(list)</td>
<td>return true if the list is empty</td>
</tr>
<tr>
<td>SizeOfListNode(list)</td>
<td>return the no of nodes in the list</td>
</tr>
<tr>
<td>FirstNodeListNode(list)</td>
<td>return a pointer to the first node</td>
</tr>
<tr>
<td>NextNodeListNode(list)</td>
<td>return a pointer to the next node</td>
</tr>
<tr>
<td>MoveRightListNode(list)</td>
<td>advance the pointer to the next tree node</td>
</tr>
<tr>
<td>GoBeginListNode(list)</td>
<td>move the pointer to the beginning of the list</td>
</tr>
<tr>
<td>GoEndListNode(list)</td>
<td>move the pointer to the last tree node of the list</td>
</tr>
<tr>
<td>InsrNodeListNode(list, pnode)</td>
<td>insert the tree node to the right of the previous node and move the pointer there</td>
</tr>
<tr>
<td>DelNextListNode(list)</td>
<td>delete the next tree node</td>
</tr>
<tr>
<td>DelNodeListNode(list, pnode)</td>
<td>delete the item-node containing the pointer to the tree node</td>
</tr>
<tr>
<td>FindNodeListNode(list, pnode)</td>
<td>find the pointer to the item-node which contains the pointer to the tree node</td>
</tr>
</tbody>
</table>

...  

**11.C Annotation:**

This command traverses the subtree of the given data abstraction, extract the entries of **Purpose of this operation** of all **Operation** forms into the output file. There are similar commands to generate different documentations. For example, "M-X OPERATION I/O" can produce a list of operations with their input/output types, "M-X OPERATION USE" can produce a list of operations with the external function names used, "M-X DA PURPOSE" can produce a list of data abstraction modules with their purpose, etc. These commands are useful in obtaining **high level views** of some particular modules in a form tree.
12. Generate Report for Different Views

12.A Command: M-X MANAGER (Form tree name:) FTDB
               (Output file:) FTDB-MANAGER

12.B Display:

<table>
<thead>
<tr>
<th>PDM-Form-1</th>
<th>Project</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project Name:</td>
<td>Form Tree Data Base</td>
</tr>
<tr>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>Project Goals:</td>
<td>Design and implementation of</td>
</tr>
<tr>
<td></td>
<td>the Form Tree Data Base which</td>
</tr>
<tr>
<td></td>
<td>organizes the software related</td>
</tr>
<tr>
<td></td>
<td>information into form trees</td>
</tr>
<tr>
<td></td>
<td>according to blank forms</td>
</tr>
<tr>
<td></td>
<td>defined in a methodology</td>
</tr>
<tr>
<td></td>
<td>description.</td>
</tr>
<tr>
<td>Project Manager:</td>
<td>Jay Ramanathan</td>
</tr>
<tr>
<td>Project Participants:</td>
<td>Jeremy Kuo</td>
</tr>
<tr>
<td></td>
<td>Chin Li</td>
</tr>
<tr>
<td>Project Proposal</td>
<td></td>
</tr>
<tr>
<td>Proposer:</td>
<td>Jeremy Kuo</td>
</tr>
<tr>
<td>Abstract:</td>
<td>This database will be designed</td>
</tr>
<tr>
<td></td>
<td>as a hierarchy of data</td>
</tr>
<tr>
<td></td>
<td>abstractions based on the</td>
</tr>
<tr>
<td></td>
<td>attribute grammar form model.</td>
</tr>
<tr>
<td></td>
<td>This hierarchy contains</td>
</tr>
<tr>
<td></td>
<td>data structures SET_OF_FORM_</td>
</tr>
<tr>
<td></td>
<td>TREE, FORM_TREE, FILL_FORM,</td>
</tr>
<tr>
<td></td>
<td>NODE and BLANK_FORM defined</td>
</tr>
<tr>
<td></td>
<td>in MDDB.</td>
</tr>
</tbody>
</table>

12.C Annotation:

This command is defined to extract only the manager-related information from a form tree. That is, only the entries of manager-related organizers in the manager-related forms are extracted and output to the file. This command should be defined according to the manager's interest and can be altered if necessary. For example, here we suppress the information below the data abstraction module level and function module level, but the purpose of these modules are included. Similarly, we can define extended commands such as "M-X PROGRAMMER" or even "M-X NEW-MEMBER" to provide different views of the entire project related information.
13. Generate Reference Lists

13.A1 Command: M-X REF (Organizer:) External operations used
   (Text:) IsEmptyListNode (Output file:)

13.B1 Display:

| Reference list: Organizer: External operations used |
| Text: IsEmptyListNode |
| [23325] Operation: FindNodeListNode |
| [23326] Operation: IsInListNode |

13.C1 Annotation:

The current form here is assumed to be Data Abstraction Module:
LIST-NODE [233]. This command generates a list of Form-use-#s and
form headings of all forms (but only in the subtree of the current
form) which contain the organizer External operations used and the
text string "IsEmptyListNode" occurs somewhere in that entry. This
command is particularly useful for identifying the affected forms of
a change.

13.A2 Command: M-X REF (Organizer:) (Text:) NULL_LIST_NODE
   (Output file:)

13.B2 Display:

| Reference list: Text: NULL_LIST_NODE |
| [211] Data Abstraction: NODE (Organizer) Purpose of this module |
| [21113] Operation: CreateNode (Organizer) Pseudo code |
| [211131] Implementation: CreateNode (Organizer) Code |
| [21120] Operation: GetIndexChildNode (Organizer) Pseudo code |
| [211201] Implementation: GetIndexChildNode (Organizer) Code |
| ... |

13.C2 Annotation:

The current form here is assumed to be Data Abstraction Module:
NODE [211]. If the organizer is omitted, this command will look for
all organizers whose entries contain the given text string. The
reference list will show the Form-use-# and heading of the form as
well as the organizer where the text is found. This command actually
generates a cross reference if the text is a variable or function
name.
14. Search Using Pattern Matching

14. A Command: `\~Z`\~S (Pattern:) TAIL-ITEM-NODE

Display:

```
PDM-Form-11 Implementation Form-use-#[23325]
Operation: FindNodeListNode

Code:
struct ITEM-NODE *FindNodeListNode(listy pnode)
LIST-NODE list;
struct NODE *pnode;
{
    LIST-CHAR clist;
    struct ITEM-NODE *inode;
    clist = (LIST-CHAR) list;
    if (IsemptyListChar(clist))
        return( &TAIL-ITEM-NODE__);
    else
        { GoBeginListChar(clist);
            inode = clist -> First-Node;
            while (inode != &TAIL-ITEM-NODE)
                { if (pnode == (struct NODE *)(inode -> Item))
                    return(inode);
                    else
                        { clist -> Prev-Node = inode;
                            inode = inode -> Next-Node;
                        }
                } /* return tail node if search fails */
        return(&TAIL-ITEM-NODE);
    }
}
```

Annotation:

The search command "\~Z\~S" can search for occurrences of a string pattern in the entries of all organizers in the subtree of the current form in in-order manner. The cursor will be placed at the end of next occurrence after the same command is issued. The command "\~Y\~S" can do the same searching but only within the current form. Similarly, command "\~S" works only in the current entry. There are also three parallel commands: "\~Z\~R", "\~Y\~R" and "\~R" which do the same searching but in the reversed order.
15. Search Using Form Headings and Organizers

15.A Command: M-X SEARCH (Form heading:) Meeting
(Organizer:) Meeting reports

15.B Display:

PDM-Form-4 Meeting Form-use-#[111]

Date: 11/15/82

Agenda:
1. Jeremy presents the demonstration
2. Documentation
3. Status report

Attendee:

Meeting reports:
1. Demonstration postponed to Wednesday, 3pm. Make sure Del would install network switch on VT100 before then.
2. Should we move the prototype to UNIX?
   Problem: there are not enough terminals available.
3. Start testing implementation code. Using "ifdef" to trace the variable values so that new programmers can maintain the code more easily.
4. Documentation should be kept updated. Each programmer should turn the completed documentation in hard copy to Jay in an already-to-stock format.
5. Plan to write 4 technical reports:
   a. SOPREX — Jay, Soni, Suni — deadline?
   b. John Hopkins Conf. — Jay, Soni — deadline 1/14/83
   c. Tools Conf. — Kuo, Jay, Soni, Suni — deadline 1/15/83

15.C Annotation:

This command will search for the first form (in the subtree of the current form) with the given form heading and place the cursor at the entry of the given organizer. Either the form heading or the organizer is optional. Together with the commands discussed in 9, a user can access a form by its relative position to the current form, Form-use-#, form heading, or even the organizers contained.
Summary

It is not the purpose of this example to serve as a user's manual for the TRIAD prototype, therefore, the commands illustrated here are only a small subset of the entire TRIAD commands. However, we have tried to selectively present some typical commands and their effects using screen display and annotations to give readers some flavour of using this prototype to develop a real project. Also, we have focused only on demonstrating the power of the tool TRIAD/SIE by taking advantages of the structure provided by the blank forms of PDM. Reader should be reminded that the power of these commands can be better demonstrated if the size of the project-related information becomes larger.

The selected commands illustrated earlier are again highlighted below:

1. Login the TRIAD Environment
2. Create or Visit a Form Tree
3. Select a Starting Blank Form
4. Edit Entries in a Form
5. Refine an Organizer Using a Blank Form
6. Help Information
7. Display a Form Tree Structure
8. Refine an Organizer Using a Pattern
9. Visit a Form
10. Collect Programming Code from a Form Tree
11. Generate Report for High Level Views
12. Generate Report for Different Views
13. Generate Reference Lists
14. Search Using Pattern Matching
15. Search Using Form Headings and Organizers