AN EMPIRICAL INVESTIGATION OF THE IMPACT OF COGNITIVE COMPLEXITY AND EXPERIENCE OF PROGRAMMERS, AND PROGRAM COMPLEXITY ON PROGRAM COMPREHENSION AND MODIFICATION

DISSERTATION

Presented to the Graduate Council of the North Texas State University in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Omar Elnadi M. Khalil

Denton, Texas

May, 1986

The psychological characteristics of programmers are believed to be important determinants of programming productivity. However, little evidence is available to support this contention. This investigation, motivated by the lack of such evidence, was concerned with determining the influence of the programmer's cognitive complexity (differentiation and integration) and experience on comprehending and modifying programs of different levels of complexity.

Data were collected from ninety-three graduate and undergraduate students in a classroom experimental setting. In the first phase of the experiment, a background questionnaire was administered in order to collect experience and other demographic information. Also, a domain-specific Role Construct Repertory (REP) Test was administered to collect cognitive complexity information.

In the second phase, the subjects were randomly assigned to either the program comprehension group or to the program modification group. Both groups used two COBOL
programs of differing levels of complexity to do comprehension and modification exercises.

Three sets of hypotheses were tested. The first set of hypotheses was designed to evaluate the direction and strength of the relationship between cognitive complexity and program comprehension and modification. The second set of hypotheses was designed to evaluate the combined influence of cognitive complexity and program complexity on the comprehension and modification of the programs. The third set of hypotheses was designed to evaluate the moderating effect of experience on the relationship of cognitive complexity to program comprehension and modification.

Cognitive integration was found to have a significant and positive nonlinear relationship only with the relatively complex program modification scores. The subjects who were ranked high in cognitive integration performed better than those ranked low in modifying the relatively complex program; but they performed the same in modifying the relatively simple program. Cognitive differentiation was found to have no significant relationship with either comprehension scores or modification scores. Experience of the subjects did not significantly moderate the relationship of cognitive complexity and program comprehension and modification.
The findings of this study provide strong support for the applicability of the cognitive complexity theory in predicting programmers' performance in program modification situations. This conclusion is tentative as repeated research is necessary to verify further the findings of this study.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>LIST OF TABLES</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST OF ILLUSTRATIONS</td>
<td>v</td>
</tr>
<tr>
<td>LIST OF APPENDICES</td>
<td>vi</td>
</tr>
</tbody>
</table>

**Chapter**

I. **INTRODUCTION**

1.1 Introduction

1.2 Statement of the Problem and Research Framework

1.3 Objectives of the Study

1.4 Background

1.5 Significance of the Study

1.6 The Plan of Research

II. **COGNITIVE COMPLEXITY: ITS ORIGIN AND SCOPE, MEASUREMENT, AND RESEARCH RESULTS**

2.1 The Origin and Scope of Cognitive Complexity

2.2 The Measuring Instruments of Cognitive Complexity

2.2.1 Measures emphasizing the differentiation dimension

2.2.2 Measures emphasizing the integration dimension

2.2.3 Measures emphasizing both differentiation and integration dimensions

2.3 Empirical Results of Cognitive Complexity

III. **AN OVERVIEW OF PROGRAMMER'S EXPERIENCE PROGRAM COMPLEXITY, AND PROGRAM COMPREHENSION AND MODIFICATION**

3.1 Programmer's Experience

3.2 Program Complexity

3.2.1 Volume metrics

3.2.2 Control organization metrics

3.2.3 Data organization metrics
## LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>I.</td>
<td>Research Variable Set</td>
<td>105</td>
</tr>
<tr>
<td>II.</td>
<td>Complexity Metrics for the Experimental Programs</td>
<td>119</td>
</tr>
<tr>
<td>III.</td>
<td>Point Values for Grading PROGRAM 1 Comprehension</td>
<td>122</td>
</tr>
<tr>
<td>IV.</td>
<td>Point Values for Grading PROGRAM 2 Comprehension</td>
<td>123</td>
</tr>
<tr>
<td>V.</td>
<td>Point Values for Grading PROGRAM 1 Modification</td>
<td>124</td>
</tr>
<tr>
<td>VI.</td>
<td>Point Values for Grading PROGRAM 2 Modification</td>
<td>125</td>
</tr>
<tr>
<td>VII.</td>
<td>A Profile of the Subjects' Computer and Programming Background</td>
<td>141</td>
</tr>
<tr>
<td>VIII.</td>
<td>Frequency Distribution for Cognitive Differentiation Scores</td>
<td>143</td>
</tr>
<tr>
<td>IX.</td>
<td>Frequency Distribution for Cognitive Integration Scores</td>
<td>144</td>
</tr>
<tr>
<td>X.</td>
<td>Frequency Distribution for PROGRAM 1 Comprehension Scores</td>
<td>144</td>
</tr>
<tr>
<td>XI.</td>
<td>Frequency Distribution for PROGRAM 2 Comprehension Scores</td>
<td>145</td>
</tr>
<tr>
<td>XII.</td>
<td>Frequency Distribution for PROGRAM 1 Modification Scores</td>
<td>145</td>
</tr>
<tr>
<td>XIII.</td>
<td>Frequency Distribution for PROGRAM 2 Modification Scores</td>
<td>146</td>
</tr>
<tr>
<td>XIV.</td>
<td>Descriptive Statistics for Cognitive Complexity and Programming Performance Variables</td>
<td>147</td>
</tr>
</tbody>
</table>
XV. Spearman Rank Correlation Coefficients (rs)
   for Background and Experience Variables
   Which Significantly Correlated with
   Cognitive Complexity and Program
   Comprehension at P < .10  . . . . . . . . . 151

XVI. Spearman Rank Correlation Coefficients (rs)
   for Background and Experience Variables
   Which Significantly Correlated With
   Cognitive Complexity and Program
   Modification at P < .10  . . . . . . . . . 152

XVII. Spearman Correlation Coefficients (rs)
   for the Association between Cognitive
   Complexity and Programming Performance
   Before and After Sorting out the Effect
   of Subjects' Background and Experience  . . . 154

XVIII. Person Correlation coefficients for the
   Association Between Cognitive Complexity
   and Programming Performance Using a Log
   Transformation of the Residuals  . . . . . 156

XIX. The Results of Kruskal-Wallis Test Where
   Cognitive Differentiation is the Classifying
   Factor  . . . . . . . . . . . . . . . 158

XX. The Results of Kruskal-Wallis Test Where
    Cognitive Integration is the Classifying
    Factor  . . . . . . . . . . . . . . . 159

XXI. The Results of Kruskal-Wallis Test Using
    the Performance Scores Before and After
    Adjusting for the Effect of the Subjects'
    Background and Experience  . . . . . . . . 164

XXII. The Results of Kruskal-Wallis Test Using
       the Portion of PROGRAM 2 Modification
       Scores Which is Due to the Effect of the
       Subjects' Experience  . . . . . . . . . 165
## LIST OF ILLUSTRATIONS

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Research Framework in Programming</td>
<td>9</td>
</tr>
<tr>
<td>2. Person-environment Impact on Individual's Behavior</td>
<td>35</td>
</tr>
<tr>
<td>3. Functioning of Concrete and Abstract Individual in Relation to Environmental Complexity</td>
<td>37</td>
</tr>
</tbody>
</table>
# LIST OF APPENDICES

<table>
<thead>
<tr>
<th>APPENDIX</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A. Domain-Specific REP Test</td>
<td>192</td>
</tr>
<tr>
<td>B. Background and Experience Questionnaire</td>
<td>198</td>
</tr>
<tr>
<td>C. The First Experimental Program (PROGRAM 1)</td>
<td>202</td>
</tr>
<tr>
<td>D. The Second Experimental Program (PROGRAM 2)</td>
<td>205</td>
</tr>
<tr>
<td>E. Program Comprehension Exercise Booklet</td>
<td>210</td>
</tr>
<tr>
<td>F. Program Modification Exercise Booklet</td>
<td>220</td>
</tr>
<tr>
<td>G. Residual Plots of Cognitive Complexity versus Program Comprehension and Modification</td>
<td>228</td>
</tr>
</tbody>
</table>
1.1 INTRODUCTION

This study investigates the relationship of some of the psychological characteristics of computer programmers and program characteristics to the performance of some programming-relevant tasks. In particular, it evaluates the relationship of programmer's cognitive complexity and experience and program complexity to comprehending and modifying computer programs.

Computer programming is a "complex problem-solving process resulting in a carefully-detailed product" (36, p. 26). It is a labor-intensive process which requires the ability to understand a problem and to translate its solution into a language that can be processed by a computer. This process, therefore, demands complex cognitive skills which make psychological study of programming challenging and interesting from both the fundamental and applied points of view (24).

Thadhani (47) emphasized the importance of the human factors in programming when he pointed out that unless major breakthroughs occur to significantly increase programmer productivity, a shortage of programming skills in this decade will severely restrict the development and maintenance of
software applications. This view was well supported a long time ago by Weinberg (52) when he noted the importance of the human factors in programming tasks.

Because of the complex nature of the programming task, the programmer's personality—his individuality and identity—are far more important factors in his success than is usually recognized...there seems to be evidence that critical personality factors can be isolated and associated with particular programming tasks—at least in the sense of their possession rendering one incapable of performing that task well. Consequently, attention to the subject of personality should make substantial contributions to increased programmer performance—whether that attention is paid by a psychological researcher, a manager, or the programmer himself (p. 158)

Although programming is a more complex problem-solving endeavor than most of the tasks studied by cognitive psychologists (43), little is known about the cognitive abilities and processes necessary to perform programming functions. Recognizing the lack of such information, Brooks (5), Bunde (6), and Shneiderman (41) have recently proposed cognitive models to explain programmer behavior. The proposed models are based on the problem solving theory of Newell and Simmon (33). They treated the task of computer programming as an information processing problem; but these models have not been extensively tested, and their explanations of individual differences in programming tasks remain unsatisfactory.

Scott et al. (39) suggested that cognitive contents—"a person's idea about the world" (p. 7)—and cognitive
process—"the mechanism by which ideas arise, are ma-
nipulated and transformed" (p. 7)—may be viewed as
components of personality and may be useful in explaining
individual differences. They (39, p. 33) also define cog-
nitive structure to include the application of the entire
range of structural concepts from personality theories to
both cognitive contents and cognitive processes. The authors
further explain that cognitive structure is the way a person
combines the information perceived from the outside world
or generated internally. The combination of cognitive
structure with cognitive ability (contents and process)
create unique individual differences in problem-solving
behavior.

The cognitive complexity theory is suggested to ex-
plain such individual differences in behavior; hereinafter
"cognitive complexity" is used to refer to cognitive dif-
ferentiation and integration. This theory was originally
developed by Kelly (27) to explain the individual dif-
ferences in the interpersonal relations domain. The
theory was further extended by Schroder et al. (38) and
their followers to explain individual differences in
information processing and decision making. The thrust of
the theory is that behavior is guided by one's personal
construct system. This system provides the means of under-
standing, anticipating, and predicting events. The theory
also characterizes man as capable of representing the
environment, as well as merely responding to it, and posits that different presentations lead to different behaviors (23, p. 104; 36).

It would logically follow that when a task requires the perception of a large number of pieces of information and the integration of these pieces into a comprehensive component, the cognitively complex person would be expected to perform better than a cognitively simple person. If programming is a problem-solving task which involves the decomposition of information into pieces and integrating these pieces into chunks (44), then the cognitively complex programmer would be expected to better understand a program that requires a high level of differentiation and integration than the cognitively simple programmer.

Moreover, the cognitive structure approach employed in this study would lead one to suggest that as the programmer gains experience in the domain of programming, the ability to perceive and integrate more complex program structures may increase. This view is based on the results of many of the studies on cognitive complexity (e.g., 16; 28; 30; 34; 35; 46; 49) and on programming experience (e.g., 5; 12; 17; 24; 26; 50). Based on the view of the researchers who support the generality of the concept of cognitive complexity (e.g., 4; 8; 37; 48) it would also appear that programmers of equivalent cognitive complexity would have similar dif-
difficulties in comprehending similar programs, regardless of their level of experience.

Although no published research relates cognitive complexity measures to programming performance, several studies (e.g., 2; 7; 13; 14; 20; 28; 46) have investigated cognitive complexity hypotheses in problem solving tasks in the design and use of computer-based information systems. This study explores the extension of the theory to explain individual differences in program comprehension and modification as problem-solving tasks within software development and maintenance environment. It investigates the relationships of cognitive complexity and programming experience to the performance of programmers in comprehending and modifying programs with different levels of complexity.
1.2 STATEMENT OF THE PROBLEM AND RESEARCH FRAMEWORK

The problem that motivates this study is that although the psychological characteristics of programmers are believed to be important determinants of programming productivity, little evidence is available about such characteristics and their specific influence on programming (40; 45, p. 235). In particular, evidence is lacking on the influence of individual differences in terms of the cognitive complexity and programming experience of programmers on comprehending and modifying programs with different levels of complexity. This problem is set forth in the following three questions:

(1) What are the relationships of the cognitive complexity of programmers and program complexity to program comprehension and modification?

(2) Does the experience of the programmer influence the relationships between his/her cognitive complexity and program comprehension and modification?

(3) What is the significance of cognitive complexity theory in predicting the performance of programmers in programming related tasks?

Multiple factors are thought to affect computer program development and maintenance. This view is reflected in the literature describing research on methods to improve program quality and programmer productivity (e.g., 11; 12; 17; 18; 29;
However, it becomes difficult to select which factors to test and which factors to control in the absence of a theory as a guide to identify the basic factors and their relationships. A short-run substitute is a tentative framework of the factors that are thought to affect the underlying phenomenon.

Chrysler (11) suggests a framework which encompasses six factors believed to affect programming performance: organizational operations characteristics, computer hardware, source language, programmer characteristics, programming mode, and programming problem. Benbasat and Vessey (3) add software engineering to Chrysler's framework as a factor. The problem with Chrysler's model, however, is that it ignores the nature of the relationships of the independent variables and programming performance as well as the interaction effects. This problem is recognized in the programming framework suggested by Chisamore (10), who classifies the independent variables which may influence programming performance into programmer, program, and environmental factors. The Chisamore's framework also shows the possible interaction effects of these variables on programming performance.

Recently, Gilmore and Smith (22) suggested a generalized framework for programming performance which includes program factors (complexity, language, structure, representation), contextual factors (purpose, task, activity),
context-program interaction, and programmer characteristics (motivations, relevance, cognitive style, level of ability). Although these authors recognized the interaction effect of some of the factors, they ignored the possible interaction effect of programmer characteristics and the other factors in their framework.

Figure 1 comprises a graphic representation of the research framework assumed for this study. The framework is derived from the research framework of Chervany et al. (9), which was used in the Minnesota experiments, and the programming model of Chisamore (10). The suggested framework synthesizes the various factors believed to affect programming performance. The performance of computer programming (development and maintenance) may result from the joint effects of the characteristics of the programmer, program characteristics, and environmental characteristics. In addition, it is quite likely that these factors influence one another as well.

The suggested framework is not intended to constitute a theory of programming, nor should it be regarded as complete. It is, however, advanced as being valuable in the essential synthesis of the results of previous research and by delimiting areas of investigation and their relative interdependence as a guide for ongoing or future research.
1. Programmer Characteristics
   * - Experience
   - Personality
   * - Cognitive structure
   - Innate ability
   - Motivation
   - Others

2. Programming Environment
   - Physical aspects
   - Technological aspects
   - Social aspects
   - Management aspects
   - Others

3. Program Characteristics
   - Problem domain
   - Source language
   * - Complexity
   - Structure
   - Representation
   - Documentation
   - Others

4. Programming Activity
   - Designing
   - Coding
   - Testing
   - Debugging
   - Documenting
   * - Comprehending
   * - Modifying
   - Learning
   - Others

5. Programmer Performance
   - Time
   - Errors
   - Memorization
   - Lines of code coded and/or modified
   - Others

6. Program Quality
   - Complexity
   - Reliability
   - Maintainability
   - Efficiency
   - Effectiveness
   - Others

Fig. 1—Research framework in programming
(the variables to be addressed in this study are marked with asterisks)
So far, most of the research in the suggested framework has been primarily normative (50); that is, the studies stated what should be done to improve the quality of program development and maintenance. Normative models have rarely been subjected to empirical testing, and their value remains uncertain. Furthermore, most of the empirical research studies (i.e., 17; 18; 31; 50) have experienced problems with the statistical models or the measurement instruments they applied. This investigation, therefore, will be directed at a portion of the research framework in Figure 1. The factors followed by asterisks in Figure 1 are those included in this study; the other factors are recognized in the framework but are assumed to be constant for the purpose of the study.
1.3 OBJECTIVES OF THE STUDY

The ultimate goal of this study is to extend the knowledge base that exists as to the influence of individual differences and program complexity on the performance of some programming relevant tasks. This goal includes three primary objectives. The first objective is to evaluate the direction and strength of the association between cognitive complexity (differentiation and integration) and the performance in comprehending and modifying computer programs. The second objective is to evaluate the influence of the individual differences of the programmers in terms of their cognitive complexity (differentiation and integration) on their performance in comprehending and modifying programs of different complexity. The third objective is to assess the moderating influence of experience on the relationship between cognitive complexity and program comprehension and modification. These objectives will be accomplished when the following are determined:

1. How the performance of programmers in comprehending and modifying a program is associated with their cognitive complexity.
2. How the performance of programmers in comprehending and modifying a program is affected by the level of complexity of the program.
3. How the performance of programmers in comprehending and modifying a program is affected by the combined effects of cognitive complexity and program complexity.

4. How the performance of programmers in comprehending and modifying a program is affected by differences in cognitive complexity as moderated by their programming experience.
1.4 BACKGROUND

Programming is a complex task that can be further decomposed into subtasks (43). It is therefore appropriate to isolate certain aspects of the task for more focused analysis (40). As subtasks of the programming activity, program comprehension and program modification are the two dependent variables in this study.

Although the possible impact of the programmer's psychological factors on programming relevant tasks is well recognized in the literature (e.g., 41; 52), little has been determined concerning the nature of these factors and their significance in explaining individual differences in programming. Sommerville (45, p. 235) explains, "Because of the lack of concrete evidence, any attempt to assess the influence of distinct personality traits on programmer performance is conjectural." He adds, however, that it seems likely that some personality factors might play a more important role than others in programming. For instance, the ability to withstand the stress of meeting project deadlines and the ability to adapt to changes in work environment might influence the performance of programmers.

Withstanding stress and handling complex and unstructured input from the outside world are central to cognitive complexity theory. Rooted in the work of Kelly (27) and Schroder et al. (38), cognitive complexity
maintains that people may be ordered along a continuum from concrete to abstract, depending on their ability to differentiate and integrate information (23, p. 136). Differentiation is the ability to determine the number of dimensional units of information when a person is presented with a stimulus; integration is the ability to utilize complex rules or programs to combine the differentiated dimensions (23, p. 136). It would logically follow that, when a person is involved with a task, the more complex that person is the more ways a stimulus can be perceived and the more complex ways that relevant information is located.

Thus far, there are no reported studies on testing hypotheses derived from cognitive complexity theory in the programming domain. However, there are a few reported cognitive complexity investigations in the other domains which utilized cognitive structure models that could be used to predict decision making behavior and task performance. Carlisle (7), Larreche (28), and Stabell (46) utilized the approach of Schroder et al. (38) to decision making, in which the subjects were decision makers involved with computer-aided decisions. In general, the results of these studies supported the tested hypotheses relating information processing behavior to the level of cognitive complexity. The outcomes of the tasks themselves, however, were not formally investigated in these studies.
Clark (13, 14), on the other hand, conducted two experiments in which the subjects were students involved in system analysis and design tasks. The results showed a relationship between the subjects' cognitive complexity and the characteristics of the assigned task outcomes.

Following the work of Claunch (1964) cited in (23, p. 162), Clark (13, 14), and Amernic and Beechy (1), this study stresses the gross outcomes of task performance rather than the information processing behavior utilized in such performance.

Relating to the characteristics of work environment, Scott et al. (39, p. 23) emphasize the importance of the interaction between the psychological characteristics of persons and their environment in determining their behavior. They stated that "the determination of behavior is only in part personality-based; it is also dependent on the situation and the interaction between personality and situation."

In the approach of Schroder et al. (38) to cognitive complexity, the interaction between the individual's cognitive complexity and environmental complexity is well recognized. They (38) explain that environment is complex if it provides excessively diverse and/or numerous dimensional units of information; i.e., information that requires substantial integration of thought can be described as multidimensional and therefore complex.
Schroder et al. (38) and their associates hypothesized and empirically tested the interaction effect of the individual's cognitive complexity and environmental complexity on that individual's performance. Explained in more detail in the next chapter, this relationship is an inverted U-shaped curve; i.e., increases in the complexity of the information processor's environment beyond an optimal point result in a lowering of the processor's ability to make judgments. In addition, the level of optimal performance of a highly cognitive individual is hypothesized to occur at a high level of informational complexity.

The results of reported studies (e.g., 7; 21; 28; 46) concerning the decision-making environment support the hypotheses of Schroder et al. (38) regarding the behavior of individuals in information processing. In addition, the data reported by Amernic and Beechy (1) and Claunch (1964), cited in Goldstein and Blackman (23, p. 162), provide some evidence that high and low cognitive individuals perform equally well in highly structured tasks whereas highly complex individuals perform significantly better than less complex individuals in less structured tasks.

Reading and comprehending a program comprise a task that requires a sequential approach which decomposes the program one statement at a time. Dijkstra (19) suggests that, in doing so, the programmer is trying to discover assertions which describe the program's operations. As
the programmer attempts to uncover these assertions, the potential to generate new attributes of information and discriminate between stimuli is determined by the number of dimensions he or she perceives in the program.

Drawing on the hypotheses derived from the cognitive complexity theory, in comprehending and modifying a program an increase in environmental complexity, measured by program complexity, is expected to lead to an increase in the information load required to perform these tasks. This increase in information load increases the level of the programmer's abstractness—the ability to differentiate and integrate information. Beyond a level of program complexity, however, the level of the programmer's abstractness decreases as program complexity increases. Although some of the reported research attempted to correlate programmers' performance in comprehension and modification tasks with program complexity (e.g. 17; 18; 53), no attempt has been made to test the interaction effect of cognitive complexity and program complexity on programming related tasks. This study is designed to accomplish that by evaluating the combined effect of cognitive complexity and program complexity on comprehending and modifying computer programs.

The final issue to be investigated, however, relates to experience. The differences between expert behavior and novice behavior are noticeable in complex cognitive processes, and programming is no exception. To explain such
differences in programming, Youngs (54) pointed out that the expert programmer not only performs the tasks better but also in a different way from the novice programmer. In support of Youngs' view, Shneiderman (43) points out that the real determinants of performance are the programmer's knowledge and skills accumulated through prolonged experience. Therefore, more skilled programmers should experience less difficulty and spend less time in comprehending and modifying a computer program.

Although some researchers (e.g., 5; 40; 41) sought to explain the aspects of and the reasons for such differences between novices and experts in programming, recent studies of these differences have met with mixed success. The data obtained from some of the investigations (e.g., 11; 36; 42; 43; 47) supported the intuition that expert programmers should perform better than novice programmers in programming-related tasks (e.g., coding, debugging, comprehension, modification). Some other studies (e.g., 17; 26; 29), however, have shown at best weak relationships between programmers' experience and a variety of program related variables. Perhaps these inconclusive results were the reasons for Sheil's (40) and Lawrence and Jeffery's (29) comments that there is a lack of evidence supporting today's wisdom in programming practice.

On the other hand, the generality of cognitive complexity is debatable. Some researchers (e.g., 16; 23, p.
have pointed out that experience, including past training in a specific domain, might influence the individual's cognitive complexity in that domain. Therefore, if the generality of cognitive complexity is questionable, the relationship of the programmer's cognitive complexity and his or her performance with respect to program comprehension and modification will possibly be affected by the programmer's experience. This possibility is recognized in this study whereas experience will be evaluated as a moderating factor in the relationship of cognitive complexity to program comprehension and modification.
1.5 SIGNIFICANCE OF THE STUDY

Software maintenance is an important activity in today's information systems, but unfortunately it has been neglected in the MIS literature. The cost of software maintenance was projected to contribute significantly to total expenditures of information systems (e.g., 15; 25; 32; 50). Diebold (1979), cited in (15), projected that by 1985 the cost of hardware will be one-tenth of the 1979 rate but labor costs would be twice as high as the 1979 rate. Therefore, improving programmer's productivity becomes a challenging objective for both researchers and practitioners in the Information Systems area.

The potential impact of the programmer's psychological factors on productivity is self-evident in the reported psychological studies of programming. The ultimate goal of these efforts has been the development of a psychological theory to explain the programmer's behavior and to identify the factors which influence that behavior while performing programming-related tasks. However, Sheil (40) comments, "Unfortunately, although some psychological theory is very suggestive, it usually lacks the robustness and precision required to yield exact predictions for behavior as complex as programming" (p. 102).

This study is designed to build upon prior psychological research on programming in an effort to add to the
body of knowledge in this area and provide some evidence about additional psychological factors that affect programming. Given the growing importance of programming in general, and of software maintenance in particular, along with the need for improving programmer productivity, this study is significant because it will accomplish the following:

(1) Determine the effect of programmers' cognitive complexity on their performance of program comprehension and modification tasks;

(2) Determine the effect of program complexity on programmers' performance of program comprehension and modification tasks;

(3) Determine the interaction effect of programmers' cognitive complexity and program complexity on programmers' performance of program comprehension and modification tasks; and

(4) Determine the importance of programming experience on the relationship of cognitive complexity and program comprehension and modification.

The need to conduct vigorous research to provide empirical evidence on the psychological factors that may explain the individual differences in programming is the impetus for this study. In particular, The study will provide some empirical evidence on the significance of the cognitive complexity factor in predicting individual
differences in programming. The evidence educed may contribute to continuing efforts to build a concise psychological theory of programming and the formulation of policies underlying education and training for programmers.
1.6 THE PLAN OF RESEARCH

Cognitive complexity is discussed in Chapter II. This chapter will encompass the origin and scope of cognitive complexity and its measurement, as well as the results of previous research in relevant decision-making applications. In Chapter III, an overview of the relevant programming issues will be discussed. The review includes program complexity and its measuring metrics, programmer's experience, and program comprehension and modification as programming relevant tasks. Chapter IV is devoted to the description of the research hypotheses and the experimental study in which the information to evaluate these hypotheses was obtained. In Chapter V, the statistical analysis of the data and the hypotheses testing results are presented. In the final chapter, Chapter VI, conclusions are drawn from the research results. It also includes a discussion of the implication of the research results and recommends the feasibility of future research.
CHAPTER BIBLIOGRAPHY


CHAPTER II

COGNITIVE COMPLEXITY: ITS ORIGIN AND SCOPE, MEASUREMENT, AND RESEARCH RESULTS

2.1 THE ORIGIN AND SCOPE OF COGNITIVE COMPLEXITY

The original work in cognitive complexity was formalized over thirty years ago in a theory reported by George A. Kelly in his book The Psychology of Personal Constructs. According to Kelly's theory of cognitive complexity, man is capable of representing his environment as well as merely responding to it, and different presentations lead to different behaviors. Kelly further states that man views his environment through transparent patterns, or templates, that he creates and then attempts to superimpose the realities of the world into these templates. "These patterns are Kelly's constructs, and it is the constructs that bring organization to behavior" (18, p. 104).

The personal construct system is a structural representation of cognition. It consists of a set of attributes (personal constructs) in terms of which reality is interpreted. Scott et al. (34, p. 37) further explain that behavior is guided by one's personal construct system. This system provides a means of understanding, anticipating, and predicting events.
Bieri (1966), cited in Bell and Keen (7), loosely defines cognitive complexity as "the tendency to construe social behavior in a multidimensional way, such that a more cognitively complex individual has available a more versatile system for perceiving the behavior of others than does a less cognitively complex person" (p. 143). The theoretical rationale for Bieri's concept of cognitive complexity is Kelly's (21, p. 155) distinction between propositional and constellatory constructs. Leitner et al. (25) explain, "In theory, an individual who uses propositional constructs will tend to construe events in a more complex manner than an individual who uses constellatory constructs, due to his greater freedom to place alternate constructions on events" (p. 4).

According to Kelly (21, p. 156), the personal construct system is organized to embrace ordinal relationships between constructs in such a way that the superdinate construct treats its subordinate constructs as if they were constellatory. Leitner et al. (25), therefore, argue that "although it may be more adaptive to be complex when one is using subordinate constructs, one also needs hierarchical integration if one is to make decisions about how and when to apply certain constructs" (p. 4). In other words, high complexity, in the absence of higher order integrations, may be correlated with chaos and confusion.
Leitner and his associates regard integration as necessary to organize the differentiated constructs in their view of Kelly's theory, whereas early work in cognitive complexity (e.g., 9) was primarily concerned with differentiation (37). However, Schroder et al. (32) and their followers perceive differentiation and integration as possibly more independent information processing characteristics that are correlated but not necessarily preconditions for each other.

Harvey et al. (19) and Schroder et al. (32) view cognitive growth as a process of increasing differentiation. As differentiation proceeds, it is usually, but not always, accompanied by increasingly elaborate integrative processes. When integration does develop apace with differentiation, it serves to maintain flexible interactions among the various ideas; if it does not, the distinct ideas may be disconnected or become interrelated in rather stereotyped and inflexible ways.

In support of this view, Zimring (41) argues that differentiation and integration, however separated, are equally necessary processes. Langley (23) further explains that, although differentiation serves the specialization of the organism's subsystems, integration preserves the integrity of the entire organism. Goldstein and Blackman (19, p. 138) argue also that integration is related to differentiation in that the greater the number of dimensions, the greater the
potential for complex schemata. At any given level of differ-
entiation, individual differences in integration ability
are likely to be present. Adams-Webber (1967) cited in (25),
however, questions the relationship between differentiation
and integration by arguing that whether cognitive complexity
reflects high differentiation, as Bieri suggests, and/or low
integration, as implied by research based on personal con-
struct theory, is unknowable.

Although there is no general agreement on what di-
mensions constitute cognitive complexity (e.g., 32; 34;
38), cognitive complexity is certainly a many-faceted
construct defined and operationalized in different ways (7).
The term "cognitive complexity" is used in this study to
encompass two dimensions: differentiation and integration.
This understanding of the term is based on the original
work of Kelly (21) and the results of evaluative studies
of cognitive complexity measures (e.g., 24; 29; 30; 40).
Cognitive differentiation measures how much a person's
constructs distinguish between elements in a stimulus;
cognitive integration reflects the hierarchical arrangement
of the constructs (27).

The view of cognitive complexity of Streufert and
Driver (37) further supports the perspective utilized
in this study. These researchers pointed out that two
subsystems of human information processing (HIP) would each
handle differentiation and integration in characteristic
degrees. The perceptual subsystem is largely concerned with
data search and intake. The executive subsystem, on the
other hand, would utilize the more or less differentiated
and integrated concepts generated by the perceptual sub-
system to make decisions (produce behavioral output), which
would again be more or less differentiated and integrated.

The extension of cognitive complexity to the domain of
decision making and information processing is central to the
position presented first in Harvey et al. (19) and then ex-
panded by Schroder et al. (32). The view of these authors is
that people engage in two activities in processing sensory
input: differentiation and integration. Differentiation
refers to the ability of the individual to locate stimuli
along dimensions. Integration refers to the ability of the
individual to utilize complex rules, or programs, to combine
these dimensions.

Schroder et al. (32) defined integrative complexity
as "the complexity of the schemata that determines the
organization of several dimensions in a complex cognitive
structure" (p. 165). Therefore, high integration index
structures have more rules (schemata) for forming new hier-
archies, which are generated as alternate perceptions or
further rules for comparing outcomes. The integratively high
structures contain more degrees of freedom and are more sub-
ject to change as complex changes occur in the environment
(34, p. 50).

People can be ordered along a continuum depending on
their ability to differentiate and integrate information.
The individual who is low in differentiation and integration
ability is said to be concrete; the individual who is high
in differentiation and integration is said to be abstract.

To explain, individuals varying in their levels of
cognitive complexity vary in the weightings they assign to
the dimensions they differentiate and the ways they combine
the information generated by different dimensions (18, p.
138; 25). The rules (schemata) that individuals use for
integrating information may be simple or complex along
the concrete-abstract dimension. Four levels of cognitive
complexity along the concrete-abstract dimension were
identified in the early work of Harvey et al. (19). Theseour levels are dependence, negativism, independence, and
interdependence.

The early work on integrative complexity centers around
identifying and studying differences in behavior among rep-
resentatives of these four levels of cognitive complexity.
In later research, however, this interest in the per-
formance of system-specific subjects was replaced by an
interest in the dimension of cognitive complexity as it
varies along the concrete-abstract continuum (18, p. 140).
The view of behavior as a function of the person and the environment is central to the approach of Schroder et al. (32). The behavior of an individual is best understood as an interaction of his or her differentiation and integration ability and the informational complexity of the environment. Figure 2 depicts these factors and their relationships.

![Diagram of person-environment impact on individual's behavior](image)

Fig. 2—Person-environment impact on individual's behavior

An individual's ability to differentiate and integrate mediates between the stimulus input and the behavioral output. The input consists of a range of stimuli that are filtered, or processed, by the mediating structure (cognitive complexity). This mediating structure allows the individual to differentiate environmental elements and to integrate those elements to produce behavioral output (18, p. 137).

In view of the effect of environmental complexity on the behavioral output, Schroder et al. (32) argue that each person or group information processing system responds in a typical curvilinear manner to current variations in
environmental complexity. Environmental complexity is defined as the aggregate effect of information complexity, noxity, and eucity. Information complexity is the most general component of environmental complexity and comprises information load, information diversity, and rate of information change. Noxity represents the "severity of the adverse consequences of behavior" (e.g., threats in a specific situation). Eucity is "the amount of reward or promise given by the environment" (24, p. 21).

The relationship between cognitive complexity, environmental complexity, and behavioral output is hypothesized in an inverted U-shaped curve (32). Figure 3 illustrates the hypothesized difference in the functioning of concrete and abstract individuals. The more abstract the individual, the higher the level of performance of which he or she should be capable. Abstract individuals and concrete individuals are both hypothesized to perform at best in environments of low informational complexity, but abstract individuals should perform better in environments of high informational complexity. The level of optimal performance attained by abstract individuals is hypothesized to occur at a higher level of informational complexity than the level of optimal performance for concrete individuals (18, p. 141).

What is valuable about the viewpoint of Schroder and his associates is that the personality variable is related to the environmental variable in an articulated manner.
Individuals vary in the complexity of their ability to process information and the environment varies in the complexity of the information it contains.

For the purpose of this study, however, a distinction should be made relevant to the particular environment which produces the input and upon which the behavioral output will take effect. Streufert and Driver (37) distinguish

Fig. 3—Functioning of concrete and abstract individual in relation to environmental complexity (adapted from Schroder et al., 1967).
between at least two different forms of complexity: social and non-social. The former is concerned with interpersonal perception and interaction. The latter is concerned with behavior in environments where perceptions and decisions have no interpersonal relevance. The concern of this study is the application of cognitive complexity theory in a task performance domain. Therefore, the measuring instruments of cognitive complexity and its relevant research in non-social applications only are considered below.
2.2 THE MEASURING INSTRUMENTS OF COGNITIVE COMPLEXITY

"Although most researchers discuss the concept of complexity in similar ways, they operationalize it very differently" (25, p. 6). Kelly (21) operationalized cognitive complexity by developing the Role Construct Repertory Test (REP Test). Originally, the REP Test was designed to be used in a clinical setting, in which a client rates people in terms of concepts or constructs elicited by the client. By comparing the pattern of responses from one to another, it is possible to determine the simplicity or complexity of the pattern. In addition to the work of Kelly, a number of other instruments also have been devised and used to measure cognitive complexity. These instruments can be arbitrarily classified into three groups based on the emphasis given to each of the two dimensions of differentiation and integration. These groups are measures emphasizing the differentiation dimension, measures emphasizing the integration dimension, and measures emphasizing both differentiation and integration dimensions.

2.2.1 Measures emphasizing the differentiation dimension:

Leitner et al. (25) acknowledge that the original work in operationalizing cognitive complexity was done by Bieri (9), who reasoned that people vary in the number of different constructs they have at their disposal. Some personal construct systems are highly differentiated (cognitively
complex) while others are relatively undifferentiated (cognitively simple). Furthermore, the difference in the number of constructs used by a person should be related to individual differences in behavior.

Since Kelly's (21) original Rep Test was reported, different instruments to measure cognitive complexity (differentiation) have been developed and used in cognitive complexity research. Among these measures are Bieri's (9) modified Rep Test, Scott's (33) information theory measure of cognitive complexity, the explanatory power of the first factor (EPFF) of the grid, Crockett's (15) method, average match between rows (AMR), explanatory power of the self-concept (EPSC), Vannoy's (40) methods, and Landfield's functionally independent construction (FIC) method. Leitner et al. (25) provide a critical review of several cognitive complexity measures.

Much of the research in cognitive complexity, however, has involved the use of different forms of Kelly's Rep Test technique (6). Extensive discussions of REP and its related methodology appear in several sources (e.g. 4; 21). The Rep Test is a "sorting task which allows for the assessment of relationships between constructs and which yields these primary data in matrix form" (4, p. 136). According to Goldstein and Blackman (18, pp. 106-107) and Metcalf (27), perhaps the most commonly used variation of the Rep Test
to measure differentiation is the version developed by Bieri (10) as later modified by Bieri et al. (1966) cited in (25).

In Bieri's version of the Rep Test 10 role types are identified, and the experimenter provides constructs for the subjects to use in rating each role type. The score for cognitive complexity is derived by comparing the rating given to one individual on a particular construct to ratings given to that individual on the other constructs. The higher the score, the lower the cognitive complexity. Numerous scoring methods—manual and computerized—are available for the users of Rep Tests (e.g., special issue of Int. J. of Man-Machine Studies, vol. XIII, 1980).

The main problem with the above technique is described by Goldstein and Blackman (18):

It should be noted that the measures of cognitive complexity derived from the Rep Test provide a measure of differentiation, the number of constructs used by the subject. The Rep Test does not provide a measure of integration, the ways the constructs interrelate (p. 135).

As a solution to the above problem, Leitner et al. (25) and Landfield and Barr (22) reported a technique to derive a measure of integration using a Rep Test approach. This technique will be discussed in section 2.2.3.

2.2.2 Measures emphasizing the integration dimension:

Several measures have been developed and used to measure integrative complexity as defined by Schroder et al. (32). Among these measures are the Paragraph Completion
Test (PCT), the Sentence Completion Test (SCT), the Systems
Types of Harvey et al. (19), Smith and Leach's (35) "This
I Believe" Test (TIB), the Conceptual Systems Tests (CST),
the Impression Formulation Test (IFT), and the Inter-
personal Topical Inventory (ITI).

The Paragraph Completion Test (PCT), first reported
as the Sentence Completion Test (SCT), is the most popular
measure of integrative complexity. It is a projective
method developed by Schroder et al. (32) to measure inte-
grative complexity in the general area of interpersonal
relations. When taking this test, the subject is presented
with five sentence stems (e.g., "When I am in doubt...,
"Rules.....") and allowed ninety seconds to complete these
sentences by writing at least two additional sentences in
response to each sentence stem. A score between one and
seven is assigned to the response according to the level of
integration that it reflects. Schroder et al. (32) suggested
using the mean of the two most abstract responses to provide
the integrative complexity score.

The main problem with PCT is that it does not provide a
measure of the individual's differentiation ability. Another
problem is that scoring the subjects' responses to a PCT
is not easy. Well-trained judges are required to score the
test. These judges have a great deal of opportunity for
exercising personal judgment (24, p. 19).
2.2.3 Measures emphasizing both differentiation and integration dimensions:

The third group of measures was used to assess the two dimensions of cognitive complexity (differentiation and integration). Among these measures Carr's (12) method, the multidimensional scaling (MDS) method, and Landfield and Barr's (22) integration (ordination) measure, which is used in conjunction with their older measure of differentiation (Functionally Independent Construction, FIC).

Carr's method attempts to combine both a measure of differentiation and a measure of integration called "discrimination." It measures differentiation in terms of the number of different perceptions (NDP) used by the subject. Overall integration (discrimination) is defined in terms of the mean number of interpersonal discriminations made on the three most frequently used, functionally-independent constructs. With regard to Carr's method, Leitner et al. (25) commented that, although this method is provocative and worthy of further investigation, there is a possibility that it does not detect the most meaningful constructs, resulting in the loss of interesting and important data.

In the multidimensional scaling (MDS) format, the subject is required to make judgments of similarity between pairs of complex objects. The resulting matrix of similarity judgments is analyzed to determine the minimal number of
dimensions that the subject might have used to generate the pattern of similarity ratings. MDS also generates weightings to indicate the importance of each dimension. It has been hypothesized that the number of dimensions specified in an MDS analysis relates to the differentiation dimension and that the weightings of the dimensions generated in the MDS analysis relate to the integration dimension (18, p. 148). Goldstein and Blackman (18) said, "it is unclear whether MDS will solve the measurement problem for integrative complexity theory. The parallel between dimensionality and differentiation is promising, but the relationship between the weightings of the dimensions and integration is unclear" (p. 148).

Landfield and Barr's (22) measure of cognitive complexity was developed to measure differentiation in terms of number of functionally independent constructs (FIC) and to measure integration (called "ordination") in terms of the number of levels the subject uses to apply the constructs. This method employs 15 by 15 Rep Test with elicited constructs. In rating the rule figures, the subject has the option of placing a 1 (indicating that the left pole of the construct applies), a 2 (indicating that the right pole of the construct applies), or an N (indicating that the construct does not apply to that figure). The FIC is computed by comparing similarities and differences in the
application of constructs across individuals. FIC is then defined as "the total number of separate construct units employed by a subject on a particular REP Test" (25, p. 20).

The ordination score (O) is obtained by first determining the number of levels a person uses to apply a given construct and the high-low scores; then these two numbers are to be multiplied. Given that the highest score for levels is 7 (in a 13-point scale) and the highest score (difference) for high-low is 6, by multiplication the range is zero through 42. These 15 individual construct ordination (O) scores are then averaged to yield an ordination score for constructs (Oc). The process is repeated to yield ordination scores for each person (element) being rated across the 15 constructs. These individual ordination scores for the people (elements) are then averaged to yield an ordination score for people (Op). Finally, the Oc and Op are added together to arrive at the overall ordination score. These calculations may be performed using a computer program.

The basic assumption behind the use of the Landfield and Barr's (22) measure is that the person is utilizing certain integrative, higher order conceptions, even though these conceptions do not appear on the conventional Rep Test. This issue was recognized by Leitner et al. (25) when they pointed out that "The idea of making inferences about
conceptual structures which cannot be directly assessed may be discomforting, but nonetheless scientific" (p. 47).

The cognitive complexity relevant studies have implemented a variety of the measuring instruments discussed above. The results of these studies are presented in the next section.
2.3 EMPIRICAL RESULTS OF COGNITIVE COMPLEXITY

The theory of cognitive complexity has been used in a number of studies in different domains to validate its hypotheses and its measuring instruments. Only those studies that attempted to apply cognitive complexity to decision-making and task performance are discussed below.

Schroder and his associates (32) reviewed the results of several experiments which were conducted in the form of games concerned with internation conflicts under varying environmental complexity. Using the PCT to measure cognitive complexity, they obtained results supporting their hypotheses concerning the curvilinear relationship between environmental complexity and level of information processing.

Claunch (1964), cited in (18, p. 162), hypothesized that abstract subjects would perform better than concrete subjects on an essay examination but that the two groups would perform similarly on an objective test. Using the SCT, Claunch selected 40 concrete and 40 abstract undergraduate students. The results showed that concrete students and abstract students performed similarly on an objective tests, but the abstract students performed better than the concrete students on the essay test—as hypothesized.

Amernic and Beechy (2) replicated Claunch's study using introductory accounting students with low and high levels of cognitive complexity as measured by the PCT. The researchers
analyzed the relative performance of the students on structured versus unstructured accounting examinations. The results showed that students at all levels of cognitive complexity performed equally well on the highly structured examination, but the students with high levels of cognitive complexity performed significantly better on the unstructured examination.

Suedfeld and Hogen (1966), cited in (18, p. 162), used a word association task to compare the information-processing ability of concrete and abstract subjects. The results suggested that the concrete subjects performed worse than abstract subjects in complex situations.

Using undergraduate students in a laboratory setting, Menasco (26) provided some evidence that high cognitive complexity was related to greater difficulty and discomfort in making difficult decisions. In a field study involving housewives, high cognitive complexity (differentiation) was related to greater difficulty in making decisions regarding the purchase of major appliances. The measure of decision-making difficulty, however, was based on the number of brands considered. Therefore, cognitive complexity was shown to relate to the complexity rather than the difficulty of the decision (18, p. 124).

Standing (1973), cited in (18, p. 131), conducted a study to investigate the relationship between cognitive complexity and the complexity of the work environment. He
hypothesized an inverted U-shaped relationship between the two variables. Using 40 steel mill inspectors, and a domain-specific modified REP Test, the research results supported Standing's hypotheses.

In the Information Systems area, there have been few reported studies (e.g., 11; 24; 36) that investigated the relationship of the decision maker's cognitive complexity to the use of information and decision outcomes in a decision support systems environment. These studies draw on the view of Schroder et al. (32) of cognitive complexity.

Carlisle (11) analyzed the interaction of three user characteristics (intelligence, cognitive complexity, and experience) with language interface complexity to measure their effects on problem-solving behavior. The language used consisted of a set of 26 commands organized into high, medium, and low hierarchical complexity. The general hypothesis tested was that for subjects who scored low on tests involving cognitive complexity, intelligence, and relevant experience, the optimal interface complexity would be lower than for subjects who scored higher on tests involving these characteristics. The results were in partial agreement with this hypothesis. The low complexity condition led to higher effectiveness for low user characteristics groups, and the medium complexity condition led to high user
effectiveness for the high user characteristics groups. The hypothesis was not supported for the high complexity condition.

The primary purpose of another study, Larreche's (24), was to investigate the differences in the information search behavior of individuals of differing integrative complexity when using computerized marketing models. To test his hypotheses, Larreche used undergraduate students in a marketing class and three computerized marketing models. The integrative complexity of the subjects was assessed using a domain-specific PCT. The results of the study support the applicability of the approach of Schroder et al. (32) of cognitive complexity to the implementation of computerized marketing models.

While Carlisle (11) and Larreche used artificial decision making situations, Stabell (36) investigated the relationship between cognitive complexity (integrative domain) of 30 portfolio managers' perceptions of their information environment and the volume, breadth, and balance of their use of information sources in actual decision-making situations. Cognitive complexity scores were obtained using a measure based on hierarchical cluster analysis of the output from a Rep Test. The results suggested that the volume and breadth of information were positively related to the integrative complexity of information environment perception.
More recently, Clark (13) conducted a study using graduate students in a systems analysis and design class to test the hypothesis that systems designers' psychological types as measured by a domain-specific REP Test affect the resulting system designs. Among other results, cognitive complexity was found to be inversely related to the number of modules used in the designer's solutions to a single problem. Cognitive complexity was also inversely related to the level of cohesion (functional relatedness) of the contents of the modules. That is, the modules of the relatively high complex subjects were designed with low degree of functional association among the elements included in each module. The program segment of the REP Test, however, did not significantly correlate with either the number of modules or the level of cohesion.

In another study using graduate students in a system analysis and design class, Clark (14) conducted an experiment to test the impact of various psychological characteristics on the task of systems analysis and design through the use of data flow diagrams. Cognitive complexity was measured using a domain-specific REP Test. The results showed that, as the flow complexity of the designer increases, so does his or her ability to employ maximum complexity in the design. The program complexity scores
did not show any significant relationships with any of the dependent variables.

From the studies reviewed above, it should be noted that different researchers operationalized cognitive complexity differently. In addition, different measures were used in these studies. Consequently, four issues should be considered before ending this review of cognitive complexity:

(1) the comparability of cognitive complexity research,
(2) elicited versus provided constructs in the design of Rep Tests,
(3) the generality of cognitive complexity, and
(4) the validity and reliability of cognitive complexity measures.

The following provides a brief discussion of each of these issues.

Comparing the results of research on cognitive complexity is not an easy task. The difficulty is due to several factors. First, much of the cognitive complexity research deals with interpersonal perception and has limited value in providing guidelines for modeling the activities of managers in processing information and making decisions (8). This problem may be a result of the lack of cognitive complexity research in Information Systems area.

A second factor making results comparison difficult is the failure of the researchers to report the absolute level
of cognitive complexity. A score sufficient to categorize a subject as high in cognitive complexity in one study may not be sufficiently high to categorize him or her as high in cognitive complexity in another study (18, p. 134).

A third problem making results comparison difficult is the failure of the researchers to report the proportion of the variance that is being accounted for in the statistical analysis of the results. This makes it almost impossible to assess the significance of cognitive complexity research in the various applicable areas.

A fourth problem making results comparison difficult is that much of the research (18, p. 134) was based on 10 by 10 REP Tests. This convention lends strength to the criticism of Schroder et al. (32) that a grid of 10-rows is not sufficient to assess the cognitive complexity of subjects.

Moreover, the original work on cognitive complexity as previously discussed (e.g., 9; 21) utilized a technique that required the subjects to elicit their own constructs. The modified version of the Rep Test (e.g., Bieri et al. 1966 cited in (18, p. 109)) required the subjects to use constructs that were provided to rate each role type. The comparability of the data obtained by the two different techniques is debatable.

Reviewing a variety of cognitive complexity measuring instruments, Leitner et al. (25) concluded that measures
using provided constructs do not correlate highly with measures using elicited constructs. In addition, the data provided by Metcalf (27) showed only partial support for the similarity of the scores derived from elicited and constructs that were provided.

On the other hand, in a review of the relevant literature Adams-Webber (1) concluded that the two techniques (elicited versus provided constructs) provide equivalent measures of cognitive complexity. Bieri et al. (1966), cited in (18, p. 109), reported the results of three studies that also indicated the similarity of cognitive complexity scores based on elicited and provided constructs.

This study will use a Rep Test with the elicited constructs approach. The elicited constructs approach is selected based on the position of Tripodi and Bieri (39) that for research purposes, the measurement of cognitive complexity using elicited constructs are comparable to the measurement using constructs that are provided. This viewpoint is also supported by Carr (12) and the conclusion of Goldstein and Blackman (18, p. 110) based on their review of the relevant literature.

Many of the studies (e.g.,24; 25; 29; 30; 36; 40) that compared different measures of cognitive complexity have been interpreted as arguing against the generality of the concept. This was due to the lack of significant relationships between the methods involved. However, it is not quite
clear whether this lack of relationship is due to the limitations of the measures or to the lack of generality of the concept.

Leitner et al. (25) view generality as "an underlying organizational tendency such that an individual will construe differing elements in a similar organizational manner" (p. 39). Thus, they continue,

Those studies which find that individuals who construe one set of elements (e.g., people) complexly also construe other sets of elements (e.g., nations, geometric designs, etc.) complexly will be interpreted as supporting the concept of generality. Those studies which do not find this will be interpreted as arguing for the specificity of the concept (p. 39).

The issue of the generality of cognitive complexity may be restated as questioning whether cognitive complexity is a trait. Goldstein and Blackman (18, p. 127) question whether cognitive complexity is a relatively enduring attribute of an individual or just a characteristic of an individual at a particular point in time that may vary from one situation to another. This position is also supported by Bannister and Mair (4) when they pointed out that, according to personal construct theory, the individual is in a constant state of change.

Crockett (15) argues for the specificity of cognitive complexity based on the findings that an increase in experience with a specific domain will increase complexity
in that domain. In addition, Goldstein and Blackman (18, p. 137) concluded, based on their review of studies in psychological development, that there is some evidence that the level of cognitive complexity of an individual is determined by his or her past training. The more familiar one is with objects, the more likely one is capable of differentiating between them.

Goldstein and Blackman (18, p. 173) take the position, based on their review of the studies using the integrative complexity dimension, that the level of integrative complexity may vary from one domain to another. For example, an individual who is complex in the interpersonal domain may be simple in the mathematical domain.

On the other hand, the findings of Miller (1969), cited in (25), support the generality of cognitive complexity—that is, the subjects' experience with the stimulus object has no effect on their cognitive complexity. This view is also supported by other researchers (e.g., 10; 12; 31; 39) who argue for the generality of cognitive complexity. Small but significant correlations were found in most of these studies.

Thus far, most of the work in cognitive complexity has been concerned with complexity in the interpersonal area. To
further improve the evidence for the generality of cognitive complexity, more studies across various domains are required.

Leitner et al. (25) explain:

While more research is needed in this area, the evidence for generality is stronger than the evidence for specificity. This leads to the tentative conclusion that the concept of cognitive complexity shows some generality, but that it is not nearly the general trait some investigators speak of it as being (p. 41).

Keen and Bronsema (20) comment in their evaluative paper on the studies that used the PCT to measure cognitive complexity. They stated that "Their work has not been followed up, mainly, we deduce, because of the gap between paradigm and measure. Schroder et al.'s Paragraph Completion test lacks psychometric validity" (p. 28).

In Vannoy's (40) study of the similarity of cognitive style measures, factor analysis indicated that the Sentence Completion Test (SCT) is independent of other measures. The SCT especially did not correlate significantly with a modified REP Test or a modified F scale. However, Vannoy's conclusion was criticized on the basis that he did not include many of the cognitive complexity measures in use at that time and that many of the methods he used were modified for his study (25).

Relating to the validity and reliability of the measuring instruments of cognitive complexity, Harvey (1966b), cited in (18), summarized construct validity for a number of measures of cognitive style, including a modified REP Test
and the TIB Test. Harvey reported that, in general, the "expected" relationships between integrative complexity and other measures of cognitive complexity were obtained.

Bannister and Fransella, 1971), cited in (18, p. 133), also questioned the validity of measures using Rep Tests based on constructs that were provided instead of elicited constructs. However, the above discussion of this issue shows the comparability of both techniques in measuring cognitive complexity at least for the research purposes.

On the other hand, various stimuli have been used to develop alternate forms of the REP Test that yield scores that are moderately intercorrelated. Goldstein and Blackman (18, p. 220) reported that Hunt (1951) found a test-retest correlation of 0.70 and Tripodi and Bieri (39) reported a test-retest reliability of 0.86 on their modified REP Test. Fjeld and Landfield (16) reported high degrees of reliability for different Rep Tests under different conditions of testing. Mueller (1974), cited in (18, p. 111), reported 0.82 test-retest correlation. Ohbuchi and Harike (28) also reported a moderate test-retest reliability results (.60 to .70) for REP Test administered under two different conditions.

Goldstein and Blackman (18, p. 111) stated, "Although the test-retest reliabilities reported in these studies are statistically significant, they are somewhat below the level
that is generally acceptable." However, Schroder et al. (32) reported in an evaluative study of different forms of Rep Tests satisfactory reliability and validity levels. Perhaps the consistent reliability results of these instruments were the reasons for Fjeld and Landfield (16) to comment that the use of REP Test permits to determine not only the validity of derivations from Kelly's theory of personal constructs but also to support Kelly's contention relating to the relative stability of person's construct system.

To summarize, the term "cognitive complexity," as used in this study, encompasses the differentiation and integration dimensions. Based on this view of cognitive complexity, the measuring technique of cognitive complexity of Landfield and Barr (22) is adopted for the present study to measure both dimensions of cognitive complexity. The technique of Landfield and Barr (22) provides a broader view of cognitive complexity theory as originally developed by Kelly (21). In addition, a domain-specific REP Test will be used to measure cognitive complexity in the domain of computer programming and data processing. This approach is further supported by Bannister when he made the following argument:

Repertory grid testing is essentially a highly flexible technique and not a single test. Thus although so far as it has been used to investigate constructs about people, there is no reason why the objects sorted by the subject should not be motor cars, political parties, sexual practices or domestic utensils, thereby allowing a variety of construct subsystems to be investigated (p. 113).
CHAPTER BIBLIOGRAPHY


3.1 PROGRAMMER'S EXPERIENCE

In programming, as in other complex cognitive processes, there are important differences between novice behavior and expert behavior. These differences have been noted in various domains such as chess, bridge, and music (49).

With regard to problem-solving in physics, Larkin (1980), cited in (55, p. 24), asked some beginning physics students and some experienced physicists to solve five problems in mechanics. The analysis of the different time intervals between parts of the problem solutions indicated that the experts used different patterns of processing from those the novice used. Specifically, Larkin concluded that experts use large-scale functional units and a quantitative theoretical representation, whereas novices translate a problem immediately into quantitative equations. Weiser and Shertz (70) replicated Larkin's study using programming students. The results of the experimental sorting tasks showed that expert programmers initially abstracted an
algorithm to solve a problem, whereas novice programmers
based their approach to solve a problem on the problem's
literal features.

A common explanation for this difference is that experts
have not only more information, but also they have that in-
formation better organized into useful chunks. Rather than
perceiving and remembering individual pieces of information,
the experts process meaningful groups of information, which
makes their perception more efficient and their recall per-
formance much higher (49).

Another distinction has been made between experts and
novices in the elaborateness of knowledge structures pos-
sessed by the problem-solver. When a high level of skill
is needed to solve a problem, these differences in knowl-
edge structures are likely to account for a large proportion
of the variance in performance among subjects (69).

In the programming domain, Youngs (72) points out the
possibility that the differences between novice programmers
and advanced programmers may be qualitative as well as
quantitative, in the sense that experienced programmers not
only perform the tasks better but also in a different way
from novices. Even more basic, according to Moher and
Schneider (51), is the problem of identifying subclassi-
fications of programmers based on their programming ability.
How is an expert programmer different from a novice, and how
do we recognize an expert when we see one?
Brooks (13) views programming as a process of applying knowledge structures (domains) to a problem to obtain a solution, ultimately expressed in a programming language. This knowledge structure is defined loosely by Vessey and Weber (69) as a general solution method. Floyd, cited in (69), also views the knowledge structure as a "paradigm" of programming; "branch-and-bound" and "divide-and-conquer" techniques are examples of such paradigms. An expert programmer who knows these techniques recognizes specific problems as being particular examples that are amenable to solution by using them. The novice programmer, however, may confront the problem and be left floundering (69).

Focusing on gross external performance measures, the recent studies of the differences between expert and novice programmers have met with mixed success. Experience has been shown to be related to the time necessary to get a particular program written and running, the number of times a program is submitted before completion, the number of languages known, and the rated familiarity of certain programming concepts. However, experience has not been shown to be related to the time required to debug, modify, or comprehend a new program (49; 59). Intuitively, one would expect that surely experts view these tasks differently than do novices although some of these external measures may not differ.
Jeffery and Lawrence (1979), cited in Lawrence and Jeffery (41), reported the results of the data they collected on 93 programs developed by three different organizations. The organizations were selected to represent different programming styles and environments. The analysis of data showed that "Variables which are normally held to substantially impact program development time such as programmer experience and testing turn-around time did not contribute to the model, even when greatly enlarged" (41, p. 29). In addition, programmer's experience did not impact program length.

Lawrence (1981), cited in (41), collected data on 248 commercial programs obtained from 22 organizations to evaluate the relationship of some of the programmer's characteristics and programming techniques to programming productivity. The results showed that productivity peaks after one year of experience but revealed no difference in productivity between intermediate (two to three years of experience) and experienced (more than three years of experience) programmers. In general, the results give no support to the hypothesis of experience greatly affecting productivity.

Lawrence and Jeffery (41), commenting on the above results, state, "While it is not yet clear what the explanation is for these counter-intuitive findings, other research in this area has yielded similar results" (p. 31).
This comment supports Shell's (57) conclusion, based upon a wide body of empirical research in programming, that there is a lack of evidence supporting much of today's conventional wisdom.

Chrysler (17) attempted to relate program development time to program and programmer variables. He found that the programmer's experience with the facility he or she works with is one of the variables that influenced programming time. Chrysler (18) later explored the variables which contribute to the variability in program size and found that programmer identity and experience did not appear to affect program size.

In a field study, Thadhani (68) compared the productivity of six programmers during a project and analyzed the productivity differences due to skill level. Experienced programmers were found to be two to four times more productive than the less experienced. Also, to create equivalent amounts of code, the less experienced programmers spent twice as much time at the terminal and submitted three times more compile and print jobs than the more experienced programmers.

While the researchers in the studies discussed above tested the relationship of programmer's experience to programming performance variables in field studies, others tested the same relationship in experimental settings.
Youngs (72) collected protocol data from 42 programmers (12 professionals and 30 novices) to analyze various programming errors made in performing programming tasks. In terms of programming experience, both beginners and advanced programmers had the same average number of errors in their programs. It is surprising that experience was not evident in the number of first pass bugs. In terms of the distribution of errors, experienced programmers distributed errors equally among syntax, semantic, and logical types for the first run; the syntax and semantic errors were eliminated rather quickly, leaving the bulk of errors as logic problems. Beginners, on the other hand, tended to be less able to correct the semantic errors.

Shneiderman (61) conducted an experiment to investigate the relationship between performance on a comprehension quiz and a program recall task. Forty-two students who were enrolled in an introductory programming course using COBOL were randomly divided into two groups based on past experience as measured by the number of months of programming and the number of lines of the longest program ever written by a programmer. The subjects were asked to answer 15 Multiple-choice comprehension questions and a recall task of 67 lines of COBOL program. The results showed that the main effect of experience was significant at the 0.001 level in all cases.

Sheppard et al. (1981), cited in (46), conducted an experiment to study the influence of specification format on
comprehending a program. Using nine specification-program pairs, 72 professional programmers were asked questions to assess their comprehension of a program. The results showed significant performance with greater number of programming languages.

Moher and Schneider (51) conducted an experiment to correlate a large number of background characteristics of programmers with their performance. They asked their subjects (100 students and 60 professional programmers) to perform three tasks: (1) to comprehend a short program (51 lines) and to measure the time required to correctly answer questions about the program, (2) to perform the same comprehension task using a longer program (221 lines), and (3) to code a short program (38-94 statements).

The results of Moher and Schneider's (51) experiment showed that student programmer performance correlated well with programming experience and programmer's aptitude as measured by grade point average (GPA). Professional programmer performance correlated well with programming experience but not with any measure of aptitude (not defined). Professional programmers with zero to two years of experience had similar relative performance on the three tasks; but as experience increased to more than six years, performances on all three tasks converged.
Magel (46) comments on Moher and Shneider's results by specifying three major factors in individual differences in understanding a program. The first is the knowledge of the constructs used in the program and its documentation; the second is the ability to discern patterns of similarity between the functional hypothesis and the operation hypothesis; and the third is the ability to abstract in meaningful ways from the program and from the documentation. Magel (46) then observes that the second and third factors can be taught and that they depend on previous non-programming experiences with abstract forming and pattern recognition.

Sheil (57) further explains Magel's (46) ideas: "As novices do not have the specialized knowledge and skills of experts, one might expect their performance to be largely a function of how well they can bring their skills from other areas to bear" (p. 119). Therefore, as programmers gain more experience with programming, differences in ability should become less important and performance should be improved due to the increase in programming experience to overwhelm the aptitude factor. Thus, performance should correlate better with experience than with aptitude for more experienced programmers.

In one of two studies, Mckeithen and Reitman (49) tried to infer the details of subjects' organizations of
programming concepts by constructing hierarchical representations of the relations among computer language keywords. The similarities and dissimilarities of these inferred organizations are related to the subjects' known skill level differences. Subjects of three skill levels (the manner in which skill levels were measured was not described) were shown a 31-line ALGOL W computer program in either normal or scrambled version for five two-minute study trials and then three minutes to recall. The results of the experiment showed that experienced programmers exhibited superior recall only in the normal version of the program. One explanation for this observation is that experienced programmers not only have more knowledge but have it better organized into meaningful chunks.

Sheil (57) also explains the difference between novice and expert computer programmers in terms of immediacy of response to a programming question as follows:

The immediacy with which the expert programmer "solves" problems of this sort indicates that the programmer's expertise is made up of an enormous number of interrelated pieces of knowledge. The primary piece of direct behavioral evidence for this position is Shneiderman's [1976] replication for programming of Chase and Simon's classic study on memory for chess position [1973]. In both these studies it was found that experts in a particular domain could memorize information from that domain (i.e., a program or a chess position) far better than novices, provided that the information was appropriately structured (p. 118).

The above discussion suggests that currently there is little evidence concerning how programming knowledge
structures are organized. The work on automatic programming and programming methodology, however, is suggestive. Sheil (57) points out that

This work shares the notion that programming knowledge can be thought of as a collection of units ("frames," "paradigms," "schemata"), each of which is organized as a program fragment, abstracted to some degree, together with a set of propositions about its behavior and rules for combining it with others, and indexed in terms of the problem classes for which it is appropriate (p. 118).

The structuring power of Sheil's (57) basic idea makes it attractive to programming researchers. However, one might add that, as cognitive complexity theory suggests, it is the programmer's ability to differentiate and integrate basic programming constructs under different levels of environmental complexity that affects his or her performance on programming related tasks. The present study will attempt to provide some empirical evidence on these suggestive relationships. In doing so, programmer's experience measures will be considered.

The reason behind the selection of different measures for experience is that the relevant literature suggests that "It is by no means clear which experience should be considered" (51, p. 74). For instance among the experience measures reported in the literature for distinguishing among experienced programmers are number of years of experience, job titles, longest program, and educational background.
Consequently, this study, as suggested in the above discussion, will employ a variety of measures to operationalize programmer's experience.
3.2 PROGRAM COMPLEXITY

Generally, the complexity of a problem is determined, in part, by two factors: first, the number of components in the problem; and, second, the problem solver's familiarity with the context of the problem. A person is more likely to solve a problem if it involves a small number of components. Performance is also better when the person is familiar with the context of the problem (65).

Thadhani (68) further explains the importance of these factors when he states, "There are at least two aspects of complexity. The first is intrinsic complexity, which may be defined in terms of the number of parts and the variables and connections between parts. A problem may be more complex if it has more parts, more relations, more interconnections" (p. 32). The second aspect of complexity is perceived complexity—that is, "irrespective of the intrinsic complexity of a problem, different people will perceive the same problem to be either less complex or more complex, depending on their expertise and past experience" (p. 32). Thus, the same problem requires different amounts of effort and time depending on its perceived complexity.

In his theory of program comprehension, Brooks (12) pointed out that the complexity of a program is based on how many different knowledge domains contribute information necessary to explain the operation of the program. He
believes that the programmer travels conceptually through a series of knowledge domains from the problem being solved to the program. The number of these domains, and the size of each, determine the relative complexity of the program.

Magel (46) also views program complexity as an indicator of how difficult it is to perform some particular task on that program. He adds that the difficulty of a task can be measured by either how long the task will take, how many resources it will require, or how successfully it will be performed.

Basically, program maintenance is most affected by program complexity. The degree to which the characteristics of a program impede its maintenance is called program maintainability and is driven primarily by program complexity, "the measure of how difficult the program is to comprehend and work with" (37, p. 65). Program comprehension and modification are among program maintenance tasks that are affected by program complexity.

Ejiogu (27) enumerated five categories of program complexity: structural complexity, computational complexity, logical complexity, conceptual complexity, and textual complexity. Structural complexity involves the natural expression of the topological relationships of the components of the system. Thus, structural complexity is more than merely the effects of control path; it is also the global attribute of every software module.
Computational complexity concerns the relative difficulty of accomplishing logical computational algorithms on data. This complexity is an immediate attribute of algorithms, not software. Logical complexity has to do with the relative difficulty of logical decisions or flows and branches within a system. This is a diminutive or localized structural complexity.

Conceptual complexity concerns the psychological perception of the relative difficulty of undertaking or completing a system. According to Shneiderman's (60, p. 113), psychological complexity deals with "characteristics which make it difficult for humans to understand software." The factors to be considered may also include diversity of human and material resources. Finally, textual complexity involves the static analysis of (program) source text which influences readability. The number of potential programmer's errors varies as the number of mental discriminations increases.

While Ejiogu (27) investigated the various categories of program complexity, Weissman (71, p. 57-63) concerned himself with specifying the factors that he thought might contribute to the complexity of programs. The list of Weissman includes (1) program form (e.g., use of comments, placement of declarations, choice of variable names, and paragraphing); (2) control flow (e.g., complexity of control
flow graph of a program, choice of control constructs, length and number of program segments, recursion, and level of nesting); (3) data flow (e.g., scope of variables, clustering of data references, declaration and use of data structures; and locality of operations performed or data structure), and (4) the interaction between control and data flow. Basili and Turner (1975), cited in (37), suggested that program size, data structure, data flow, and flow of control can also affect program maintenance.

There is a body of research that has to do with program complexity (52). Much of the work on measuring program complexity (e.g., 30; 35; 47) has been done in the last ten years. A number of measures have been developed to evaluate each of the factors affecting program complexity. A number of hybrid metrics have been suggested to consider more than one factor simultaneously. These measures have been the subject of several evaluative studies (e.g., 6; 7; 8; 25; 27; 34; 37; 48).

Basili and Hutchens (6) roughly classify current program complexity measures into two basic groups: (1) static metrics that are measures of the product at one particular point in time, and (2) historic metrics that are measures of the product and process taken over time. Static metrics are the most widely used measures of program complexity. Based on the physical attributes of a software product, static metrics fall into three basic categories: volume, control
organization, and data organization (6). Before discussing the measures in each category, it should be noted that this classification is arbitrary. Basili and Hutchens (6) stated, "it is not always clear to which category a particular metric belongs. For example, we may view cyclomatic complexity as a volume or a control metric depending upon the desired emphasis" (p. 667).

3.2.1 Volume metrics:

The oldest and probably the most widely applied complexity metric is program size. As Elshoff (28) pointed out, very large programs incur problems simply by virtue of the volume of the information that must be absorbed to understand the problem. Program size is easy to calculate and has definable measures. The number of lines of code (LOC), the number of procedures, Halstead's (35) volume (V) metric, the average length of procedures and the number of variables are examples of volume metrics. The number of input/output formats and other abstraction metrics are volume metrics as well. The latter, however, are measures of the logical size, rather than merely the physical size, of a program (6).

Perhaps the number of lines of code (LOC) and Halstead's (35) software science are the most widely used metrics to operationalize program size as a measure of program complexity (6; 46). Traditionally, different studies have
been reported with varying results to correlate programmer productivity and program characteristics with lines of code (LOC). For instance, Lientz and Swanson (43) reported that in their survey they found that larger systems (as measured by LOC) seemed to require more maintenance efforts, including debugging. Solivan (1974), cited in (34), found a strong relationship between an algorithm's length and the occurrence of errors.

Bowen (no date), cited in (34), examined the correlations between errors and program length for 75 modules in three projects for the U. S. Department of Defense. He found correlation coefficients ranging from 0.51 to 0.98 and obtained similar correlations when using McCabe's metric of cyclomatic complexity as a predictor variable.

Gremillion (34) analyzed the data for 346 programs making up a manufacturing support system used by a large electronics equipment manufacturer. The number of lines of code (LOC) was more highly correlated with Halstead's metric of program volume than with difficulty metric, lending support to the notion that a larger program is not necessarily a more difficult one. Gremillion adds that LOC would be the best measure of complexity to use for predicting repair requests.

Yet, Jones (39) and Jeffery and Lawrence (38) suggest that LOC has many drawbacks. In addition several researchers
(e.g., 38; 39; 47) reported that problems arise when different results are compared due to the differing definitions used and the dependence of the LOC counts on the implemented programming languages.

In an attempt to overcome the problems inherent in the LOC metric, Halstead (35, 36) introduced his theory of software science and its metrics. Halstead views a program as an ordered string of operators and operands—and nothing more than that. Consequently, a program can be characterized by four basic measures: (1) the number of unique operators \( n_1 \), (2) the number of unique operands \( n_2 \), (3) the total number of operators \( N_1 \), and (4) the total number of operands \( N_2 \). It follows that the length \( N \) of a program is merely the sum of \( N_1 \) and \( N_2 \); and similarly, its vocabulary \( n \) is simply the sum of \( n_1 \) and \( n_2 \).

Baker and Zweben (4) view the software science effort metric \( E \) as a measure which "is purported to capture an overall notion of program complexity" (p. 508). Basili and Hutchens (6) view it as a measure of only a single aspect (e.g., volume) of program complexity. The metric \( E \) is defined using the ratio of program volume \( V \) and program level \( L \) as follows:

\[
E = \frac{V}{L},
\]

where \( V \) (program volume) is computed as \( V = N \log_2 n \) and \( L \) (program level) is computed as \( L = \frac{2}{n_1} \frac{n_2}{N_2} \).
The E metric can also be derived as $E = VD$, where $D$ (program difficulty) is the inverse of $L$. The metric $D$ is used as a measure of "error-proneness." A program with high $D$ is likely to be more difficult to construct, and this may lead to more errors. As Christensen et al. (16) pointed out, this metric appears to be a measure of both the "ease of writing" and "ease of reading" a program.

The effort metric ($E$) was hypothesized to be the amount of mental effort made to complete a program in terms of the number of elementary mental discriminations required. Thus, the difficulty of programming increases as the volume of the program increases. A more thorough development of software science measures and a succinct survey of the major software science results are reported by Fitzsimmons and Love (29) and Shen et al. (58).

Most of the early work on the metric $E$ as a measure of program complexity was empirical in nature and provided evidence that $E$ is related to program design characteristics (e.g., 67), programming time (e.g., 19; 29; 35; 36), number of bugs encountered during program development (e.g., 36; 53; 54), program clarity (e.g., 31; 32), program comprehension and modification efforts (e.g., 22; 23; 29), and program modularity (e.g., 4).

The literature reviewed indicated generally that the metrics based on measures of program size have been the most successful to date, with experimental evidence indicating
that larger programs have greater maintenance costs than smaller ones. However, other characteristics such as data structure, data flow, and flow of control become vitally important as the size difference decreases. In other words, program size metrics can be a good nominal role to use in putting programs into one "complexity category," but these metrics may not be able to distinguish between different programs in the same category (37).

3.2.2 Control organization metrics:

Control organization metrics are measures of the comprehensibility of control structures. The majority of the work in software complexity over the past ten years has dealt with the effects of control flow on program complexity (37).

The complexity of control flow in a program is commonly measured based on the density of control transfers within the program or interactions of control transfers. Either approach to measuring control flow complexity normally represents a program as a flow graph to expose the control flow topology of the program. The flow graph of a program is simply a directed graph that corresponds to the program's flow of control.

Different metrics have been suggested to measure program complexity based on its control organization. The earliest metric is the logical complexity metric proposed by Gilb (30) of the amount of decision-making logic in the
program. The logical complexity measure depends on determining the number of absolute binary decisions in the program as well as the ratio of absolute logical complexity to the total number of statements in the program.

McCabe's (47) cyclomatic complexity metric, when viewed as the number of control paths (6), is also a control flow complexity metric. McCabe's metric is based on the cyclomatic number $V(G)$ of a program flow graph. For a flow graph with $e$ edges, $n$ nodes, and $p$ connected components, the cyclomatic complexity is calculated using the following equation:

$$V(G) = e - n + 2p$$

where the cyclomatic number $V(G)$ is viewed as the number of linearly independent circuits (one which is not a linear combination of two or more other circuits) in a strongly connected graph. The greater the cyclomatic number $V(G)$, the higher the complexity. This metric, however, can be reduced to counting the number of predicates (comparisons) in a module (i.e., module complexity = number of predicates + 1) (48).

McCabe's cyclomatic number has been widely accepted as a measure of control flow complexity, perhaps because it is easy to calculate and is intuitively satisfying (37). Or, as Baker and Zweben (4) concluded, "it seems that McCabe's cyclomatic complexity is on firm analytical ground and
adequately quantifies control flow complexity with the exception of the linearization problem" (p. 511). In addition, McCabe suggests that cyclomatic complexity is applicable in determining how difficult nesting will be, and empirical studies have been carried out on the effectiveness of this cyclomatic measure with favorable results (e.g., 22; 23; 56).

Although McCabe's metric does consider the control structures used in a module and the number of execution paths, it treats all predicates as contributing the same amount of complexity (48). Two modules can produce the same complexity according to McCabe's metric, but one is more complex than the other. Mayers (1977), cited in (37), recognized this problem and tried to extend McCabe's metric by noting that predicates with compound conditions are more complex than predicates with a single condition. Unfortunately, no results comparing Mayers' measure to maintenance difficulties have been reported, so its applicability is uncertain.

In general, control flow metrics fail to be comprehensive. They do not take into consideration the contribution of any factor except control flow complexity (37). However, control flow metrics do a fairly good job of differentiating between two programs that are otherwise equivalent in other characteristics such as size. Therefore, a useful approach may be to use control flow metrics to
differentiate among programs that have already been placed in the same size categories using size metrics.

3.2.3 Data organization metrics:

Data organization metrics are measures of data use and visibility as well as the interactions between data within a program. Different data organization measures reported in the literature (e.g., 6; 37) can be used to measure complexity based on the way program data are used, organized, or allocated. Among these measures the span between data references (based on the locality of the data references in the program), segment-global usage pair (based on the usage of global data within the program) and Chapin's Q measure (data items) are viewed differently, depending on how they are used.

Harrison et al. (37), however, criticize data organization metrics based on the failure of that approach to be comprehensive because the span between data references indirectly measures program length in some cases but not consistently enough to qualify as a true hybrid. Furthermore, most of these techniques are not widely applicable and have not been used in studies of their predictive power for software maintenance efforts.

Finally, the above discussion of program complexity measures suggests that what is available in the literature is a variety of metrics to measure different aspects of
program complexity. However, as McClure (48) suggests, a more complete (hybrid) technique for measuring complexity must include an extension of the number of possible execution paths and the control structures and the variables used to direct path selection. To date, suggested general (hybrid) techniques such as Hansen's measure and Oviedo's measure (37), Albrecht's function points (1), and Behrens (9) lack empirical support for their validity.

In the absence of a generally accepted measure of program complexity, one must choose one or more of the complexity measures available in the literature. Evaluative studies of cognitive complexity measures (e.g., 7; 10; 21; 37; 40; 58) provide some evidence demonstrating that software science metrics and McCabe's cyclomatic number are ranked high in terms of their empirical evidence and their wide applicability. Therefore, these two measures, as well as the number of lines of codes (LOC), will be used in this study to rank the experimental programs in terms of their complexity.
3.3 PROGRAM COMPREHENSION AND MODIFICATION

As mentioned earlier, in such a complex activity as programming, it would be appropriate, especially for research purposes, to isolate certain aspects of the activity for more focused analysis (3; 57). Sheil (57) explains that the difficulty of reliable experimentation using complete programming tasks suggests experiments which focus on either isolated aspects of the programming task or on the psychological claims implied in current programming techniques.

Programming activity can be decomposed into different subtasks. Moher and Shneider (51) decompose programming activity into the following steps: defining the problem, outlining the solution, selecting and representing algorithms, coding the problem, debugging, testing and validation, documentation, program maintenance, and understanding. Shneiderman (62) distinguishes among four tasks relating to programming: comprehension, composition, debugging, and modification.

The most basic task—and in some ways the hardest one to measure—is program comprehension (e.g., 11; 51; 61). Shneiderman (61) stresses the importance of the program comprehension task: "Although much attention has been focused on program composition, comprehension is becoming
recognized in its own right and as a key component of the vital tasks of debugging and modification" (p. 465).

Magel (46), Di Persio et al. (26), Mynatt (52), and Shneiderman and Mayer (63) further emphasize the importance of comprehension as it is necessary to perform the tasks of composition, debugging, and modification. Thus, factors relating to comprehension must be explored. The central contention of their views is that programmers develop an internal semantic structure to represent the syntax of the program.

As described in Shneiderman and Mayer's (63) syntactic/semantic model of programmer behavior, "encoding" is the process by which programmers convert the program to internal semantics. This process is analogous to the "chunking" process first described by Miller (50) in his classic paper "The Magical Number Seven Plus or Minus Two." Instead of absorbing the program on a line-by-line basis, programmers recognize the function of groups of statements and then piece together these chunks to form even larger chunks until the entire program is comprehended.

Chaudhary and Sahasrabuddhe (15) view program comprehension as a two-stage process: (1) the interpretation stage and (2) the learning stage. In the first stage, explicit facts about the program are abstracted for mental representation. In the second stage, various meta-inferential techniques are applied to search for rules (implicit facts)
hidden under the gathered explicit facts by suggesting hypotheses for testing and eventual confirmation into such rules. The psychological complexity of programs will be a function of all the factors, discussed in section 3.2, which either aid or hinder the interpretation and learning of the program.

Comprehending a program has been reported to be a measure of how well a subject understands a program (62), but it has been operationalized in different ways. According to Lukey (45), understanding a program may be operationalized by the construction of descriptions of the program. These descriptions should indicate what the program does and how it does it.

Shneiderman and his associates (60; 61; 62; 63) are responsible for most of the published work in program comprehension and the use of program memorization (recall) as a measure of comprehension. Other comprehension measures such as subjective examinations, program-flow tracing, multiple choice questions, and self-evaluation have also been used (e.g., 61; 71).

Program memorization is the most widely used measure of program comprehension in programming studies. Criak and Lockhart (1972), cited in (44), suggested that one remembers best those things he understands most thoroughly. The basic notion is that the human processor which handles information
chunks is limited. The human processor chunks and processes information in two different ways—one that serves to maintain the information at a given level and another that analyzes the information to form a chunk at a deeper level. The speed at which information will decay from memory is a function of the level to which the information has been processed; and the depth to which information is processed is a function of available processing time, the person's attention, and the compatibility of the new information with the existing information in memory. Therefore, if a person is asked to memorize programs, those programs that were most easily understood would be recalled best. Therefore, program memorization tasks may be viewed as a useful test of programmer performance since they are easy to prepare, relatively easy to evaluate, and appear to provide a good metric of overall subject comprehension of programming concepts (26).

The validity of the memorization measure is based on the syntactic/semantic model of programmer behavior introduced by Shneiderman and Mayer (63). Based on this model, Shneiderman has suggested the performance on memorization/reconstruction tasks as a measure of programmer ability. He hypothesized that, when people gain experience in programming, their capability for recognizing program structures increases. The results of memorization experiments (e.g., 15; 22; 26; 44; 59; 63; 64; 71) support these
hypotheses and indicate that memorization is a strong correlate of program comprehension and programming ability.

It should be noted that the above discussion emphasizes program comprehension rather than program writing. There are two reasons for doing so. First, comprehending a program is an essential step in the performance of any program maintenance task, which is the concern of this study. Second, experimentation with program writing has shown that the ability to write a program and the ability to understand it are intimately related (2).

On the other hand, there are multiple levels of program comprehension. It is possible to follow each line of code without understanding the overall program function, and also it may be possible to understand the program function without understanding each of the steps. There is also an intermediate level of understanding concerning control structures, module design, and data structures (24; 25; 61).

While memorization tests the programmer's comprehension of the "global" perspective of a program, making a modification of that program requires the programmer to comprehend its local and detailed prospective. However, the ability to modify a program is affected by its complexity; consequently, it is important to measure performance in terms of program modification as well. Although the performance of modification may correlate with performance in comprehension
(e.g., 61), program modification is treated as one of the dependent variables in this study for two reasons: (1) the relative difficulty of the modification task and (2) the relative importance of program modification in the software maintenance environment.

Modifying computer programs is difficult for several reasons. First, the programmer must simultaneously keep track of several aspects of the program's detailed specification, but the ability to do this is severely restricted (3). Second, the variety within all computer programs that must be diagnosed and modified is probably greater than the variety within the examples learned or demonstrated in training or learning sessions. Third, modifying as well as writing a program requires a high degree of precision.

The importance of software maintenance has been stressed recently in Information Systems and in Computer Science literature (e.g., 20; 37; 42; 43; 66). This importance, however, was predicted thirteen years ago by Canning (14), when he suggested that software maintenance is similar to an "iceberg" with low visibility but high impact on the success of information systems organizations.

Research on software maintenance (e.g., Lientz, 43) shows that software maintenance and operational support consume substantial hardware and software resources in the information systems environment. In this regard, Lientz
(43) reported that maintenance and enhancement were found to consume approximately half of system and programming personnel hours and that approximately 60 percent of maintenance/enhancement effort was for perfective maintenance.

The term "software maintenance" is usually used in the literature (e.g., 42; 66) to include three types of maintenance: (1) corrective (repair)—dealing with failure in performance, processing, or implementing, (2) adaptive—responding to a changing environment, and (3) perfective—enhancing processing efficiency, performance, or system maintainability. Enhancement is further decomposed into new reports, data addition, file reformatting, consolidation, file expansion, and condensation.

In a program maintenance task, the programmer first develops internal semantics representing the current program. The statement(s) of the modification must be reflected in an alteration of these internal semantics, followed by an alteration of the programming statements. The modification task requires skills gained in composition, comprehension, and debugging (63).

Program comprehension is the first dependent variable to be investigated in this study, and program modification is the second. The experimental modification task will be designed to represent the type of tasks included within
adaptive maintenance—the third category of software maintenance. The adaptive modification task is selected because of the relative importance of adaptive maintenance activities in the software maintenance environment (42).
3.4 **DEFINITION OF TERMS**

Specific terms used in this study have the following definitions:

1. **Cognitive complexity**: the complexity of the programmer's cognitive structure as measured by the differentiation and integration scores on a domain-specific Rep Test.

2. **Programmer's experience**: the computer and programming-related experience as operationalized by a number of measures such as the number of programming courses, number of computer related courses, number of months of programming in COBOL, number of months of programming in other programming languages, number of programming languages known, the length of longest program ever written, number of months spent in developing new programs, number of months spent in maintaining old programs, and the familiarity with the common programming methodologies and techniques.

3. **Program complexity**: the characteristics of a program as measured by the number of executable lines of code (LOC), Halstead's effort (E) metric, and McCabe's cyclomatic number, which contribute to the difficulty of comprehending and modifying that program.
(4) Program comprehension: the programmer's ability to recall a functionally equivalent program after reading it, as measured on a scale from 0 to 100. Zero indicates that the program is incorrectly recalled, and 100 indicates that the program is perfectly recalled).

(5) Program modification: the programmer's ability to modify a program to satisfy specific requirements as measured by a scale from 0 to 100. Zero indicates that the modifications are totally incorrect, and 100 indicates that the modifications are perfectly correct.
CHAPTER BIBLIOGRAPHY


CHAPTER IV

RESEARCH METHODOLOGY

4.1 RESEARCH FRAMEWORK

The variables being investigated in this study are shown below in Table I. The dependent, independent, and covariate variables are a subset of the variables presented in the programming framework (figure 1).

TABLE I

RESEARCH VARIABLE SET

<table>
<thead>
<tr>
<th>INDEPENDENT VARIABLES</th>
<th>DEPENDENT VARIABLES</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cognitive Complexity:</td>
<td>Programming Performance:</td>
</tr>
<tr>
<td>- Differentiation</td>
<td>- Comprehension</td>
</tr>
<tr>
<td>- Integration</td>
<td>- Modification</td>
</tr>
</tbody>
</table>

COVARIATE VARIABLES

- Programmer's Experience

The subsequent sections describe the hypothesized relationships among the variables presented in the table above and the procedure required to evaluate these hypotheses.
4.2 RESEARCH HYPOTHESES

To carry out the purpose of this study eight hypotheses are formally stated below. They are divided into three groups. Each hypothesis is stated in the null form.

Group I hypotheses: The relationship of cognitive complexity to program comprehension and modification.

Although the possible impact of the programmer's psychological factors on programming is well recognized in the literature (e.g., 24; 29; 33), little has been determined concerning the nature of these factors and their significance in predicting and explaining individual differences in programming.

One of the factors that may influence the performance of programmers is the ability to withstand the stress of meeting project deadlines and the ability to adapt to changes in work environment (29). Withstanding stress and handling complex and unstructured input from the outside world are central to cognitive complexity theory.

The results of the studies which followed the view of Schroder et al. (21) of cognitive complexity in general (e.g., 4; 17; 30) support the tested hypotheses relating information processing behavior to level of cognitive complexity. The outcomes of the tasks themselves, however, were not formally investigated in these studies. Moreover,
the results of Clark's (6, 7) studies showed tentative relationships between the subjects' cognitive complexity and the characteristics of the assigned system analysis and design tasks outcomes.

Thus far, there are no reported studies on testing hypotheses derived from cognitive complexity theory in the programming domain. Therefore, Group I hypotheses are designed to explore the possible relationship between programmers' cognitive complexity and the characteristics of their performance in comprehending and modifying programs. These hypotheses are formalized to stress the gross outcomes of task performance rather than the information processing behavior in such performance.

H1a- The performance of programmers in program comprehension tasks does not correlate with their cognitive differentiation scores on a domain-specific Rep Test.

H1b- The performance of programmers in program comprehension tasks does not correlate with their cognitive integration scores on a domain-specific Rep Test.

H2a- The performance of programmers in program modification tasks does not correlate with their cognitive differentiation scores on a domain-specific Rep Test.

H2b- The performance of programmers in program modification tasks does not correlate with their cognitive integration scores on a domain-specific Rep Test.

Group II hypotheses: The interaction effect of cognitive complexity and program complexity on program comprehension and modification.

As discussed in Chapter II, Schroder et al. (21) and their associates hypothesized and empirically tested the
interaction effect of the individual's cognitive complexity and environmental complexity on that individual's performance. The results of reported studies (e.g., 4; 10; 17; 30) concerning the decision-making environment support of the hypotheses of Schroder et al. (21) regarding the behavior of individuals in information processing. In addition, the data reported by Amernic and Beechy (2) and Claunch (1964), cited in (13, p. 162), provide some evidence that high and low cognitive individuals perform equally well in highly structured tasks whereas highly complex individuals perform significantly better than less complex individuals in less structured tasks. In a systems analysis education environment, however, Clark (6, 7) reported that the program complexity segment of a REP test did not correlate significantly with the number of modules used or the number of primitive constructs used in problem solution.

Drawing on the hypotheses derived from the cognitive complexity theory in comprehending and modifying a program, an increase in environmental complexity, measured by program complexity, is expected to lead to an increase in the information load required to perform these tasks. This increase in information load increases the level of the programmer's abstractness—the ability to differentiate and integrate information. Beyond a level of program complexity, however,
the level of the programmer's abstractness decreases as program complexity increases.

Therefore, if the programming activity involves the decomposition into pieces and recoding the semantic chunks, cognitively complex programmers will be more able to comprehend a program that requires a higher level of recoding and will be able to modify it to satisfy new requirements. However, when the program is relatively simple, both cognitively simple and complex programmers would perform equally well.

Although some of the reported research attempted to correlate programmers' performance in comprehension and modification tasks with program complexity (e.g. 8; 9; 23; 33), no attempt has been made to test the interaction effect of cognitive complexity and program complexity on programming related tasks. Therefore Group II hypotheses are formalized to test that interaction effect.

H3a- There is no difference between the performance of programmers who are ranked high in cognitive differentiation and the performance of those ranked low in comprehending a program of a relatively low complexity.

H3b- There is no difference between the performance of programmers who are ranked high in cognitive differentiation and the performance of those ranked low in comprehending a program of a relatively high complexity.

H4a- There is no difference between the performance of programmers who are ranked high in cognitive differentiation and the performance of those ranked low in modifying a program of a relatively low complexity.
H4b- There is no difference between the performance of programmers who are ranked high in cognitive differentiation and the performance of those ranked low in modifying a program of a relatively high complexity.

H5a- There is no difference between the performance of programmers who are ranked high in cognitive integration and the performance of those ranked low in comprehending a program of a relatively low complexity.

H5b- There is no difference between the performance of programmers who are ranked high in cognitive integration and the performance of those ranked low in comprehending a program of a relatively high complexity.

H6a- There is no difference between the performance of programmers who are ranked high in cognitive integration and the performance of those ranked low in modifying a program of a relatively low complexity.

H6b- There is no difference between the performance of programmers who are ranked high in cognitive integration and the performance of those ranked low in modifying a program of a relatively high complexity.

Group III hypotheses: The influence of programmer's experience on the relationship of cognitive complexity to program comprehension and modification.

Although some researchers (e.g., 3; 22; 24) attempted to explain the aspects of and the reasons for such differences between novices and experts in programming, more recent studies of these differences have met with mixed success. The data obtained from some of the investigations (e.g., 5; 19; 25; 26; 32) supported the intuition that expert programmers should perform better than novice programmers in programming-related tasks. Other studies (e.g., 9; 15; 18), however, have shown weak relationships at best between programmers' experience and a variety of
program-related variables. Perhaps these inconclusive results were the reasons for Sheil's (22) and Lawrence and Jeffery's (18) comments that there is a lack of evidence supporting today's wisdom in programming practice.

On the other hand, the generality of cognitive complexity continues to be debatable. Certain researchers (e.g., 13, p. 137) point out that experience and past training in a specific domain might influence the individual's cognitive complexity in that domain. Therefore, if the generality of cognitive complexity is debatable, the relationship of the programmer's cognitive complexity and his or her performance with respect to program comprehension and modification will possibly be affected by the programmer's experience. This possibility is formally recognized in Group III hypotheses.

**H7a** - When adjusted for programmer's experience, there is no difference between the performance of programmers who are ranked high in cognitive differentiation and the performance of those ranked low in comprehending a program of a relatively high complexity.

**H7b** - When adjusted for programmer's experience, there is no difference between the performance of programmers who are ranked high in cognitive integration and the performance of those ranked low in comprehending a program of a relatively high complexity.

**H8a** - When adjusted for programmer's experience, there is no difference between the performance of programmers who are ranked high in cognitive differentiation and the performance of those ranked low in modifying a program of a relatively high complexity.
H3b- When adjusted for programmer's experience, there is no difference between the performance of programmers who are ranked high in cognitive integration and the performance of those ranked low in modifying a program of a relatively high complexity.
4.3 RESEARCH DESIGN

The research approach that this study will adopt is a laboratory experimental setting conducted in the classroom. Two groups will be used, one for each dependent variable—program comprehension and program modification. Each group will perform two tasks using two programs of significantly different levels of complexity.

The experiment will be presented to the students in several sections of different graduate and undergraduate Information Systems classes at North Texas State University (NTSU). Participation will be on a voluntary basis however. The subjects will be selected to represent different groups of students with different levels of computer and programming experience.

The study will examine the influence of cognitive complexity (differentiation and integration) upon the two dependent variable (comprehension and modification) using two programs with different levels of complexity. The study also examines the possible influence of programmer's experience on the relationship between the independent and dependent variables. Among the available tested instruments designed to measure cognitive differentiation and cognitive integration (see section 2.2), this study will use a domain-specific REP Test. This test was chosen for three reasons: (1) the REP Test is not a projective test—that is an expert
is not needed to grade the responses; (2) given the conflicting views of the generality of cognitive complexity traits, a domain-specific test seems more appropriate in analyzing the research results; and (3) the satisfactory reported reliability and validity tests of different forms of the REP Test (1; 11; 12; 20).

The domain-specific REP Test was developed by modifying the original REP Test in use in psychology. The modified domain-specific REP Test differs from the original in two aspects. First, the columns constructs were replaced by a selected sample of data processing and programming concepts. Second, the instructions were modified to restructure the response procedure to conform to the domain-specific REP Test.

The initial sample of data processing and programming constructs was selected based on their commonality in programming languages and data base applications. Using the notion of "operator" and "operand," the sample represented both a group of operations and the data type to be operated on. The data type represents both the logical data view and physical data view. The final set of the constructs, as presented in the modified version of the domain-specific REP Test, were selected as a result of (a) the feedback obtained from a number of Information Systems and Computer Science faculties at NTSU and (b) a pilot study of two versions of a 15 by 15 REP Tests administered to 25 graduate students in
Information Systems. Appendix A represents an example of the developed domain-specific REP Test for the purpose of this study.

The REP Test measures two dimensions of cognitive complexity: (1) differentiation, and (2) integration. These dimensions are viewed as continuum, in which individuals are classified as relatively complex or simple.

Differentiation scores are computed by comparing similarities and differences in the application of the elicited constructs (rows) across the programming and data processing concepts (columns). As discussed earlier in Chapter II, the functionally independent construction (FIC) is the total number of separate construct units employed by a subject on a particular REP Test. The total FIC is the summation of the FIC for columns and FIC for rows. The higher the FIC score, the more dimensions the person used to evaluate and sort the programming and data processing concepts.

Integration scores are computed by assessing the number of levels a person uses to apply a given construct. The integration scores are first computed for the columns and rows by multiplying the number of rating levels by the difference between the highest and lowest rating used for each construct. The final integration score is the summation of the columns and rows integration scores. The higher the
integration score the more complex the rules and programs used by the person to combine the differentiated dimensions.
4.4. PROCEDURES FOR COLLECTION OF DATA

The data collection process will be divided into two phases. Phase I is the administration of the background and experience questionnaire and the domain-specific REP Test. The background and experience questionnaire solicits information on (1) previous academic background and achievement, (2) previous exposure to programming and data processing in college education, (3) formal programming training and previous work experience, and (4) familiarity and use of programming methodologies and techniques. The background and experience questionnaire appears as Appendix B.

The data collected from the REP Tests as completed by the subjects will be used to obtain the scores for cognitive differentiation and integration. The subjects, according to their scores, will be classified into one of three groups within each dimension: (1) the relatively high group, (2) the medium group, and (3) the relatively low group.

Phase II of the data collection process is the administration of the program comprehension and program modification exercises. The exercises are a paper and pencil test designed to permit the evaluation of the subject's ability to understand and correctly recall a program or to modify the program to meet specific requirements. Section 4.4.1 includes a detailed description of the contents of the exercises.
The exercises to be used in Phase II use the same programs and they differ only in the type of response the subjects are asked for. One group will be asked to study a program and then reconstruct a program functionally equivalent to the original one, hereafter referred to as "the program comprehension group." The other group will be asked to add to and/or change the necessary statements to modify a program to satisfy given requirements, hereafter referred to as "the program modification group."

4.4.1 EXERCISES USED

The exercises to be used in Phase II of this laboratory experiment are self-contained instruments to be administered in about a one-hour sitting. There is a total of four booklets. These booklets are reproduced in appendices C, D, E, and F. The first two booklets, "PROGRAM 1" and "PROGRAM 2," are two COBOL programs to be used by both the program comprehension group and the program modification group.

The two programs were selected from among the general business application programs in COBOL programming textbooks. The structure of the programs is modularized so that different paragraphs perform different functions. The two programs were selected based on the following criteria:

a. Typical data structure and operations for the business problem domain should be included. Therefore, PROGRAM 1
performs the purchase orders listing function and PROGRAM 2 performs employee records editing function.

b. The two programs should represent two different levels of complexity. The complexity of the two selected programs is determined by using three measures: (1) the number of executable lines of code (LOC), (2) the Halstead's effort (E) metric, and (3) McCabe's cyclomatic number (V(G)). The resulting complexity metrics are presented in Table II.

**TABLE II**

<table>
<thead>
<tr>
<th>PROGRAM</th>
<th>LOC</th>
<th>Halstead's E</th>
<th>McCabe's V(G)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROGRAM 1</td>
<td>116</td>
<td>31,692</td>
<td>5</td>
</tr>
<tr>
<td>PROGRAM 2</td>
<td>219</td>
<td>138,049</td>
<td>36</td>
</tr>
</tbody>
</table>

The source of the two programs is the Instructor's Manual for COBOL for the 80's by J. Wayne Spence. PROGRAM 1 represents the relatively low complex program, and PROGRAM 2 represents the relatively complex one. The programs were cleaned up and reconstructed in order to keep constant the factors not being explicitly varied throughout the two programs. In reconstructing the programs, care was taken to ensure that the programs were broken at reasonable points over page boundaries (e.g., divisions and sections
boundaries). In addition, no comments were included in either of the programs listings.

The third and the fourth booklets, titled "PROGRAM COMPREHENSION EXERCISE" and "PROGRAM MODIFICATION EXERCISE," contain the exercises and the procedure which the subject must use to do the exercises. The two booklets have the same organization: (1) the INSTRUCTIONS part and (2) the exercises part. The exercises part was further divided into two separate sections: SECTION 1 and SECTION 2.

The INSTRUCTIONS part of the PROGRAM COMPREHENSION EXERCISE booklet explains the nature and the sequence of the exercises and the procedures to be followed to do these exercises. The instructions inform the subject that he/she will be asked to do a comprehension exercise in a fixed amount of time. The subject will be given a COBOL program to study for a short time, using any method he/she wish to understand the functions of the program. The subject, then, will be asked to reconstruct a COBOL program which he/she thinks is functionally equivalent to the original one. An illustrative example is included in the instructions to demonstrate the procedure.

SECTION 1 of the program comprehension exercises includes two parts. PART 1 is to inform the subject of the procedure and the time allowed to study PROGRAM 1. PART 2 is to inform the subject of the procedure and the time
allowed to reconstruct the PROCEDURE DIVISION of the program he/she studied in PART 1. The time allowed for PART 1 and PART 2 is 6 and 12 minutes, respectively. SECTION 2 is the same as SECTION 1 except that PROGRAM 2 is used instead of PROGRAM 1; and the time allowed for PART 1 and PART 2 is 12 and 25 minutes, respectively.

The INSTRUCTIONS part of the PROGRAM MODIFICATION EXERCISE booklet explains the nature and the sequence of the exercises and the procedures to be followed. The instructions inform the subject that he/she is asked to make a specific modification to a COBOL program within a fixed amount of time. The subject should first study the program and understand its functions before adding and/or modifying the necessary statements to satisfy specific requirements. An illustrative example is presented in the instructions to demonstrate the procedure.

SECTION 1 of the exercises includes one part to inform the subject of the procedure and the time allowed to modify PROGRAM 1 to satisfy three given requirements. The time allowed for SECTION 1 is 18 minutes. SECTION 2 is the same as SECTION 1 except that PROGRAM 2 is used instead of PROGRAM 1 and the time allowed for SECTION 2 is 28 minutes. It should be noted that the time prescribed for the exercises was based on the results of a pilot study using twelve students who were working as student consultants at NTSU computer labs in the summer of 1985.
4.4.2 GRADING THE EXERCISES

The technique to be used to quantify the subject's performance in the comprehension exercises was designed to determine the correctness of the reconstructed programs. The

TABLE III
POINT VALUES FOR GRADING PROGRAM 1 COMPREHENSION

<table>
<thead>
<tr>
<th>CONSTRUCT</th>
<th>POINTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>The main logic organization</td>
<td>20</td>
</tr>
<tr>
<td>Opening the files and reading the first input record</td>
<td>10</td>
</tr>
<tr>
<td>Writing report headings</td>
<td>10</td>
</tr>
<tr>
<td>Moving input fields to output fields</td>
<td>15</td>
</tr>
<tr>
<td>Checking and writing the contents of the detail line</td>
<td>30</td>
</tr>
<tr>
<td>Looping and reading the next input record</td>
<td>10</td>
</tr>
<tr>
<td>Closing files and end program</td>
<td>5</td>
</tr>
<tr>
<td>TOTAL</td>
<td>100</td>
</tr>
</tbody>
</table>

grading technique should capture the functional equivalence of the subject's recalled program. To do that, the logical constructs which appeared in each program will be used in grading the comprehension exercises. These logical constructs and their corresponding point values are presented in Tables III and IV.
TABLE IV
POINT VALUES FOR GRADING PROGRAM 2 COMPREHENSION

<table>
<thead>
<tr>
<th>CONSTRUCT</th>
<th>POINTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>The main logic organization</td>
<td>5</td>
</tr>
<tr>
<td>Open files and read the first record</td>
<td>5</td>
</tr>
<tr>
<td>The editing process sequence and looping</td>
<td>10</td>
</tr>
<tr>
<td>Validate the department number</td>
<td>3</td>
</tr>
<tr>
<td>Validate and match the employee number</td>
<td>6</td>
</tr>
<tr>
<td>Validate the social security number</td>
<td>3</td>
</tr>
<tr>
<td>Validate the employee name</td>
<td>3</td>
</tr>
<tr>
<td>Validate the pay type</td>
<td>3</td>
</tr>
<tr>
<td>Validate the pay frequencies</td>
<td>3</td>
</tr>
<tr>
<td>The logic for checking pay rates</td>
<td>7</td>
</tr>
<tr>
<td>Validate the hourly pay rate</td>
<td>3</td>
</tr>
<tr>
<td>Validate the salary pay rate</td>
<td>8</td>
</tr>
<tr>
<td>Validate the employment date</td>
<td>6</td>
</tr>
<tr>
<td>Validate the health information</td>
<td>3</td>
</tr>
<tr>
<td>Validate the retirement information</td>
<td>3</td>
</tr>
<tr>
<td>Validate the savings information</td>
<td>6</td>
</tr>
<tr>
<td>Write the error message</td>
<td>5</td>
</tr>
<tr>
<td>Write the report headings</td>
<td>10</td>
</tr>
<tr>
<td>Close files and end program</td>
<td>3</td>
</tr>
<tr>
<td>TOTAL</td>
<td>100</td>
</tr>
</tbody>
</table>

It should be noted that the point values were assigned to every logical construct based on the relative difficulty of the construct. The relative difficulty of a logical construct was measured in terms of the number of lines of code and IF statements required to implement the construct in COBOL. The higher the level of construct difficulty, the more points awarded for its recognition.

The comprehension scores are based on a scale from 1 to 100. Points are awarded on the basis of the subject's reconstruction efforts (see Table III and IV). Thus, a subject
will be graded based on the correctness of his/her logic and the syntax of his/her reconstructed programs. The syntax appearance of the reconstructed version, however, need not be the same as the original one. This approach will facilitate testing the subjects ability to turn the syntax structures of the original program into semantic structures and, in turn, translate the semantic structures back into syntactically correct programs. The validity of this approach is based on the syntactic/semantic model of programmer behavior (27).

### TABLE V

**POINT VALUES FOR GRADING PROGRAM 1 MODIFICATION**

<table>
<thead>
<tr>
<th>MODIFICATION REQUIREMENTS</th>
<th>DATA DIVISION (POINTS)</th>
<th>PROCEDURE DIVISION (POINTS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Check the DISCOUNT is zero when QUANTITY &lt; 2000</td>
<td>--</td>
<td>20</td>
</tr>
<tr>
<td>2. Calculate and print the COST, TAX, and DISCOUNT subtotals for each vendor</td>
<td>20</td>
<td>30</td>
</tr>
<tr>
<td>3. Calculate and print the COST, TAX and DISCOUNT grandtotals</td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td><strong>TOTAL</strong></td>
<td><strong>30</strong></td>
<td><strong>70</strong></td>
</tr>
</tbody>
</table>

The technique to be used in quantifying the subject's performance in the modification exercises was designed to
determine the correctness and the adequacy of the modified parts of the programs to satisfy the given requirements. To do that, the required new and/or modified statements in both DATA DIVISION and PROCEDURE DIVISION will be used for grading the modification exercises. The modification requirements and their corresponding point values are presented in Tables V and VI.

TABLE VI
POINT VALUES FOR GRADING PROGRAM 2 MODIFICATION

<table>
<thead>
<tr>
<th>Modification requirements</th>
<th>DATA DIVISION point values</th>
<th>PROCEDURE DIVISION point values</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Check the matching between EMPLOYEE NUMBER and DEPARTMENT NUMBER</td>
<td>-</td>
<td>25</td>
</tr>
<tr>
<td>2. Calculate and print the SAVING subtotals and the SAVING grandtotals</td>
<td>15</td>
<td>30</td>
</tr>
<tr>
<td>3. Determine and print the highest and lowest PAY RATE for every PAY TYPE</td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>TOTAL</td>
<td>25</td>
<td>75</td>
</tr>
</tbody>
</table>

From Tables V and VI, for every modification subtask, point values were assigned for the required modifications in both DATA DIVISION AND PROCEDURE DIVISION. The point values were assigned based on the relative difficulty of imple-
menting the modification requirements. The higher the level of difficulty, the more points awarded for the implementation.

The modification scores are based on a scale from 1 to 100. Points are awarded on the basis of the relative success of the subject's modification effort (see Table V and VI). This approach differs from the one used by Shneiderman (25), who used a self-evaluation subjective measure for a modification task. In this study, the adequacy and the correctness of both the logic and the syntax of the implementation will be taken into account in scoring the modification exercises. The final score will be the sum of the individual modifications scores.

To maintain consistency in grading the exercises, the grader will score both the first and the second exercises. Scores will be indicated for each subpart within each booklet. The final grades will be noted on the cover sheet of the booklet. This procedure will be repeated to ensure the consistency and the accuracy of the scoring process.
4.5 PROCEDURE FOR ANALYZING THE DATA

The general purpose of the data analysis is to look first for the expected relationships between the independent and dependent variables. Then, if the analysis shows the existence of such relationships, the second step is to look for differences in performance that might be affected by the factors being studied. Primarily, significant effects that are due to single factor variations are sought, although the analysis will also indicate when several factors interact. The steps and the statistical techniques required to perform this analysis are described as follows:

(1) Because of the exploratory nature of this study, descriptive statistics (e.g., mean, median, variance, standard deviation, etc.) will be computed to learn as much as is possible about the set of variables before using the data for testing the research hypotheses. Based on the statistics, a profile for each independent and dependent variable will be obtained to determine the characteristics of the data distributions.

(2) Basically, nonparametric tests will be applied in testing the research hypotheses. Nonparametric tests are justified because (1) the independent and dependent variables are measured on scales which are probably not interval measures, (2) the possible lack of homogeneity of variance in the data, and (3) the expected relatively
small sample size (28). However, should further analysis be needed due to the exploratory nature of the study and nonparametric tests are not available, parametric tests will be applied to further the analysis. The level of significance below which every null hypothesis will be rejected is 0.05.

(3) To test Group I hypotheses, the significance of the degree of the association between the independent variable (cognitive complexity scores) and the dependent variables (comprehension scores and modification scores), the Spearman rank correlation coefficient (rs) procedure will be used. The power efficiency for this test is 91 per cent (28, p. 213). The correlation coefficient (rs) measures the observed correlation between cognitive complexity variables (differentiation and integration) and each of the dependent variables (comprehension and modification).

(4) To test Group II hypotheses, first, subjects will be divided into three groups in terms of their cognitive differentiation and integration scores respectively as follows:

   High: \( X \geq (M + (.8S)) \)
   Medium: \((M + (.8S)) > X > (M - (.8S))\)
   Low: \( X \leq (M - (.8S))\)

where \( X \) is the subject scores in each dimension of cognitive complexity, \( M \) is the mean score for the group,
and \( S \) is the standard deviation of the scores distribution. Assuming the normality of cognitive complexity scores, the \((.85)\) should classify the sample as follows: 20 per cent (high), 60 per cent (medium), and 20 per cent (low). Second, the Kruskal-Wallis one-way analysis of variance will be applied to test for significant differences among the three groups' (high versus medium versus low) scores on the two dependent variables (comprehension and modification). This analysis will be done using the performance scores adjusted for the possible influence of the covariate factor (programmer's experience). The power efficiency of the Kruskal-Wallis test, comparing with the \( F \) test, is 95.5 per cent ([28, p. 192-193]).

(5) To test group III hypotheses, the significant of the moderating effect of programmer's experience on the performance differences among the three subgroups (high, medium, and low), this procedure will be followed. First, the Spearman rank correlation coefficient (\( rs \)) procedure will be used to evaluate the association between the different measures of experience (as covariates) and the dependent variables (comprehension and modification). The purpose of this step is to identify the experience variables which correlate with the two dependent variables at \( P < .10 \). Second, the portion of performance scores (comprehension and modification) that is due to
experience will be calculated. This portion equals the difference between the performance raw scores and the residuals after sorting out the influence of experience of those raw scores. Third, the Kruskal-Wallis test will be applied to test the significance of the differences of this portion among the three groups (high, medium, and low).

(6) The data and the results of the above analyses will be reported in tables designed to reflect the data distribution and the resulting statistics of testing the research hypotheses.
4.6 RESEARCH LIMITATIONS

According to Kerlinger (16), limitations and trade-offs are present in the selection of one of several potential research designs. This study is no exception. Moreover, being classified with the psychological research in MIS, the study may share the same criticism (e.g., Huber, 14; Taylor and Benbasat, 31) directed to the value of that line of research.

The general problem of psychological studies of programming is described by Sheil (22,p. 118) when he says,

Hypotheses which posit differences in either individual aptitude or task difficulty are, therefore, at best, extremely difficult to investigate, as the enormous size of the knowledge base being drawn on imply that different individuals approach the "same" task with vastly different resources (p. 118).

In this study, it was the intention of the researcher to use an experimental design which uses professional programmers with randomization to control for extraneous variables and, consequently, enhance the external and internal validity of the results. However, the decision was made to use the research design described in this chapter and the use of computer programming students because of the difficulty experienced in finding sufficient professional programmers who would voluntarily participate in the experiment.

Another limitation of this study is the restriction on the generalizations that can be drawn from the data
collected. This limitation is due to the exploratory nature of the study, its narrow scope (e.g., the relatively small sizes of the experimental programs), and the exclusion of other important variables (e.g., the other components of program complexity) included in the research framework (figure 1). It should also be recognized that the relative importance of those factors that are controlled versus those that are varied is an issue of real-world significance.
CHAPTER BIBLIOGRAPHY


CHAPTER V

THE RESULTS AND ANALYSIS OF DATA

5.1 ANALYSIS OF DATA

The information collected in Phase I from the subjects' responses to the background and experience questionnaire was summarized to provide a profile of the subjects who participated in the study. The profile considers the means, medians, standard deviations, and frequency distributions for the gathered data.

The subjects' responses to the REP Test collected in Phase I of this study were then scored, using a computer program which was available at North Texas State University (NTSU). This program produces different statistics for the two dimensions of cognitive complexity—differentiation and integration. The means, medians, and standard deviations for the differentiation and integration scores were then computed for the whole sample.

The subjects' responses to the comprehension and modification exercises collected in Phase II of the study were graded by the researcher using the grading scheme discussed earlier in section 4.4.2. The assigned scores represent the subjects' performance as to each exercise.

These data were then used to perform a series of statistical analyses to test the research null hypotheses. First, a series of parametric and nonparametric correlation
analyses was conducted to test the significance of the relationship between cognitive complexity variables and programming performance variables as stated in Group I hypotheses. Second, the data were further analyzed with the Kruskal-Wallis one-way analysis of variance test to evaluate the significance of the difference existing among the performance of the different subgroups as stated in Group II hypotheses. Third, the data were finally analyzed with the Kruskal-Wallis test to evaluate the significance of the moderating effect of subjects' experience as stated in Group III hypotheses. The significance below which each hypothesis was to be rejected was set at the .05 level.
5.2 SUBJECT'S PROFILE

Initially, one hundred and nineteen subjects participated in this study. They were students at NTSU, enrolled in seven different graduate and undergraduate Business Computer Information Systems (BCIS) courses for the 1985 Summer Semester. The BCIS courses included one section of a graduate course in Systems Analysis and Design, one section of a senior level course on Problems in Information Systems, one section of a course on Software Engineering, three sections of the advanced course in COBOL Programming, and two sections of the introductory course in COBOL Programming. The data were collected during the seventh and eighth week of the semester. Students were selected so as to obtain various levels of computer and programming experience.

Twenty-six subjects were excluded from the data analysis for one or more of the following reasons: (1) incomplete response to the REP Test in Phase I of the experiment; (2) the failure to comply with the instructions in Phase II of the experiment; (3) withdrawal of the subject from class during the course of the experiment. These exclusions left us with ninety-three subjects' responses to be used in data analysis phase.

The subjects' demographic information was collected in Phase I. This information included the following: (1) ages
of the subjects who ranged from twenty to forty-six with a mean of twenty-six and a standard deviation of 5.3; (2) the group was made up of fifty-six males and thirty-seven females; (3) fifty-six of the subjects spoke English as a native language while thirty-seven did not; (4) the group was made up of forty-eight undergraduate and forty-five graduate students; (5) for educational background, 46% reported no degrees, 42% reported a BA degree, and 12% reported an MBA degree.

Table VII summarizes the computer and programming background of the subjects. The subjects reported a relatively high overall grade point average (GPA) (mean = 3.1); they had completed an average of two COBOL courses, and the total number of COBOL courses ranged from one to five. They had finished an average of six computer-related courses. The subjects, however, reported a very low average in their formal programming training in COBOL (mean = 0.1) and the other programming languages (mean = 1.6).

Relating to work experience, the subjects reported on the average three months of experience in program development (mean = 3.0, minimum = 0, and maximum = 78) and on the average two months experience in program maintenance (mean = 2.2, minimum = 0, and maximum = 66). The group reported that on the average they knew and programmed with three different languages (mean = 3.3, minimum = 1, and maximum = 9). Among the common programming techniques and
methodologies reported in the background and experience questionnaire (Appendix B), group members were familiar with and used five (mean = 5, minimum = 0, and maximum = 12), familiar with but did not use 3 (mean = 3.5, minimum = 0, and maximum = 8), and unfamiliar with four (mean = 4.5, minimum = 0, and maximum = 13).

TABLE VII
A PROFILE OF THE SUBJECTS' COMPUTER AND PROGRAMMING BACKGROUND
(N = 93)

<table>
<thead>
<tr>
<th>VARIABLE</th>
<th>MEAN</th>
<th>MEDIAN</th>
<th>STANDARD DEVIATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Grade point average</td>
<td>3.1</td>
<td>3.1</td>
<td>.66</td>
</tr>
<tr>
<td>Number of COBOL courses</td>
<td>2.2</td>
<td>2.0</td>
<td>.76</td>
</tr>
<tr>
<td>Number of computer related courses taken at NTSU</td>
<td>6.0</td>
<td>6.0</td>
<td>2.50</td>
</tr>
<tr>
<td>other schools</td>
<td>1.4</td>
<td>1.0</td>
<td>1.69</td>
</tr>
<tr>
<td>Number of days training in:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>COBOL</td>
<td>0.1</td>
<td>0.0</td>
<td>1.04</td>
</tr>
<tr>
<td>other languages</td>
<td>1.6</td>
<td>0.0</td>
<td>5.16</td>
</tr>
<tr>
<td>Number of months in a full/part time job requiring:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>program development</td>
<td>3.0</td>
<td>0.0</td>
<td>10.33</td>
</tr>
<tr>
<td>program maintenance</td>
<td>2.2</td>
<td>0.0</td>
<td>8.98</td>
</tr>
<tr>
<td>Number of languages known</td>
<td>3.3</td>
<td>3.0</td>
<td>1.84</td>
</tr>
<tr>
<td>Frequencies of the programming methodologies/techniques:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>familiarity and use</td>
<td>5.0</td>
<td>5.0</td>
<td>2.49</td>
</tr>
<tr>
<td>familiarity but no use</td>
<td>3.5</td>
<td>3.0</td>
<td>2.19</td>
</tr>
<tr>
<td>no familiarity</td>
<td>4.5</td>
<td>4.0</td>
<td>2.89</td>
</tr>
</tbody>
</table>
Relating to programming practice, the following information was reported: (1) for the largest program ever written, 10% reported less than 500 lines, 48% reported 500 and less than 1000 lines, 24% reported 1000 to 2000 lines, and 18% reported more than 2000 lines; (2) for the largest program ever written in COBOL, 13% reported less than 500 lines, 52% reported 500 and less than 1000 lines, 19% reported 1000 to 2000 lines, and 16% reported more than 2000 lines; (3) for the use of structured-programming techniques, 86% reported they had to use them in their programming assignments, and 14% reported they did not have to; (4) for the actual use of structured-programming techniques, 1% never used them, 12% used them in some of their programs, 47% used them in most of their programs, and 40% used them in all of their programs.

These subjects, as a group, have greater computer and programming experience than those used in some of the previously cited classroom programming experimental studies (e.g., 2; 3; 4).
5.3 DESCRIPTIVE STATISTICS

The classification of the subjects according to their cognitive differentiation and integration levels was made using the resulting scores from the REP Test administered in Phase I of the experiment. The differentiation score ranges from two to twenty—the higher the score the higher the differentiation ability of the person. The integration score ranges from 0 to 60 and, again, the higher the score the higher the integration ability of the person. The frequency distributions of cognitive differentiation and integration scores are presented in Tables VIII and Table IX.

<table>
<thead>
<tr>
<th>TABLE VIII</th>
</tr>
</thead>
<tbody>
<tr>
<td>FREQUENCY DISTRIBUTION FOR COGNITIVE DIFFERENTIATION SCORES</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SCORE</th>
<th>FREQUENCIES</th>
<th>PERCENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-4</td>
<td>10</td>
<td>10.9</td>
</tr>
<tr>
<td>5-8</td>
<td>27</td>
<td>29.3</td>
</tr>
<tr>
<td>9-12</td>
<td>22</td>
<td>23.9</td>
</tr>
<tr>
<td>13-16</td>
<td>23</td>
<td>25.0</td>
</tr>
<tr>
<td>17-20</td>
<td>10</td>
<td>10.9</td>
</tr>
<tr>
<td></td>
<td>92</td>
<td>100.0</td>
</tr>
</tbody>
</table>

The method for grading the comprehension and modification exercises was based on a hundred-points scale (section 4.4.2). The lower the score, the more errors there
Table IX
FREQUENCY DISTRIBUTION FOR COGNITIVE INTEGRATION SCORES

<table>
<thead>
<tr>
<th>SCORE</th>
<th>FREQUENCIES</th>
<th>PERCENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-10</td>
<td>1</td>
<td>1.1</td>
</tr>
<tr>
<td>11-20</td>
<td>4</td>
<td>4.4</td>
</tr>
<tr>
<td>21-30</td>
<td>22</td>
<td>24.2</td>
</tr>
<tr>
<td>31-40</td>
<td>39</td>
<td>42.8</td>
</tr>
<tr>
<td>41-50</td>
<td>21</td>
<td>23.1</td>
</tr>
<tr>
<td>51-60</td>
<td>4</td>
<td>4.4</td>
</tr>
<tr>
<td></td>
<td>91</td>
<td>100.0</td>
</tr>
</tbody>
</table>

TABLE X
FREQUENCY DISTRIBUTION FOR PROGRAM 1 COMPREHENSION SCORES

<table>
<thead>
<tr>
<th>SCORE</th>
<th>FREQUENCIES</th>
<th>PERCENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>21-30</td>
<td>1</td>
<td>2.3</td>
</tr>
<tr>
<td>31-40</td>
<td>1</td>
<td>2.3</td>
</tr>
<tr>
<td>41-50</td>
<td>3</td>
<td>6.8</td>
</tr>
<tr>
<td>51-60</td>
<td>2</td>
<td>4.5</td>
</tr>
<tr>
<td>61-70</td>
<td>5</td>
<td>11.4</td>
</tr>
<tr>
<td>71-80</td>
<td>8</td>
<td>18.2</td>
</tr>
<tr>
<td>81-90</td>
<td>10</td>
<td>22.7</td>
</tr>
<tr>
<td>91-100</td>
<td>14</td>
<td>31.8</td>
</tr>
<tr>
<td></td>
<td>44</td>
<td>100.0</td>
</tr>
</tbody>
</table>
were in comprehension or modification of the experimental programs. Frequency distributions for the overall performance scores in program comprehension and program modification exercises are presented in Tables X, XI, XII, and XIII.

**TABLE XI**

**FREQUENCY DISTRIBUTION FOR PROGRAM 2**
**COMPREHENSION SCORES**

<table>
<thead>
<tr>
<th>SCORE</th>
<th>FREQUENCIES</th>
<th>PERCENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-10</td>
<td>2</td>
<td>4.5</td>
</tr>
<tr>
<td>11-20</td>
<td>2</td>
<td>4.5</td>
</tr>
<tr>
<td>21-30</td>
<td>12</td>
<td>27.4</td>
</tr>
<tr>
<td>31-40</td>
<td>18</td>
<td>40.9</td>
</tr>
<tr>
<td>41-50</td>
<td>7</td>
<td>15.9</td>
</tr>
<tr>
<td>51-60</td>
<td>2</td>
<td>4.5</td>
</tr>
<tr>
<td>61-70</td>
<td>1</td>
<td>2.3</td>
</tr>
<tr>
<td>44</td>
<td></td>
<td>100.0</td>
</tr>
</tbody>
</table>

**TABLE XII**

**FREQUENCY DISTRIBUTION FOR PROGRAM 1**
**MODIFICATION SCORES**

<table>
<thead>
<tr>
<th>SCORE</th>
<th>FREQUENCIES</th>
<th>PERCENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-10</td>
<td>3</td>
<td>6.3</td>
</tr>
<tr>
<td>11-20</td>
<td>4</td>
<td>8.3</td>
</tr>
<tr>
<td>21-30</td>
<td>7</td>
<td>14.6</td>
</tr>
<tr>
<td>31-40</td>
<td>10</td>
<td>20.7</td>
</tr>
<tr>
<td>41-50</td>
<td>14</td>
<td>29.2</td>
</tr>
<tr>
<td>51-60</td>
<td>7</td>
<td>14.6</td>
</tr>
<tr>
<td>61-70</td>
<td>3</td>
<td>6.3</td>
</tr>
<tr>
<td>48</td>
<td></td>
<td>100.0</td>
</tr>
</tbody>
</table>
In addition to frequency distributions, the mean scores and standard deviations for cognitive complexity and programming performance were calculated. The results are presented in Table XIV.

From the information presented in Table XIV, within the program comprehension group, the subjects were more successful in comprehending PROGRAM 1 than PROGRAM 2. Also, within the program modification group, the subjects were more successful in modifying PROGRAM 1 than PROGRAM 2. This is not surprising since PROGRAM 1 is less complex than PROGRAM 2. Moreover, the program comprehension group performed better in comprehending both programs than did the program modification group in modifying the same programs.
<table>
<thead>
<tr>
<th>VARIABLE</th>
<th>MEAN</th>
<th>MEDIAN</th>
<th>STANDARD DEVIATION</th>
<th>Sample SIZE (N)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. COGNITIVE COMPLEXITY</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Differentiation</td>
<td>10.90</td>
<td>11.0</td>
<td>5.17</td>
<td>92</td>
</tr>
<tr>
<td>Integration</td>
<td>34.95</td>
<td>35.5</td>
<td>9.76</td>
<td>91</td>
</tr>
<tr>
<td>2. PROGRAMMING PERFORMANCE</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Comprehension</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PROGRAM 1</td>
<td>76.7</td>
<td>82.0</td>
<td>18.68</td>
<td>44</td>
</tr>
<tr>
<td>PROGRAM 2</td>
<td>33.3</td>
<td>34.0</td>
<td>11.20</td>
<td>44</td>
</tr>
<tr>
<td>Modification</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PROGRAM 1</td>
<td>38.7</td>
<td>41.0</td>
<td>15.43</td>
<td>48</td>
</tr>
<tr>
<td>PROGRAM 2</td>
<td>34.4</td>
<td>32.0</td>
<td>19.51</td>
<td>48</td>
</tr>
</tbody>
</table>
5.4 TESTING THE HYPOTHESES

Using different parametric and nonparametric tests, the data from this experiment were analyzed to test all of the hypotheses. The two dependent variables (program comprehension and program modification) were analyzed separately. For each dependent variable, the subjects were classified into three groups (high, medium, and low) according to their differentiation scores and integration scores, respectively. More specifically, the subjects were classified first into high, medium, and low in cognitive differentiation, and second, into high, medium, and low in cognitive integration. Therefore, the two dimensions of cognitive complexity were analyzed separately.

In was thought that, before performing any hypotheses testing, evaluating the main effect of program complexity upon the performance of the program comprehension and program modification tasks should be done first. Following this sequence, the nonparametric Wilcoxon matched-pairs signed-rank test was used to test the significance of the difference between the performance scores using the relatively simple program (PROGRAM 1) and the performance scores using the relatively complex one (PROGRAM 2).

For the program comprehension group, the performance scores using PROGRAM 1 were significantly higher than the performance scores using PROGRAM 2 at P < .01 (one-tailed).
For the program modification group, the performance scores using PROGRAM 1 were significantly higher than the performance scores using PROGRAM 2 only at $P < .10$ (one-tailed). These results suggest that although program complexity strongly influenced the comprehension group's performance, it did not influence the modification group's performance with the same degree of strength. The results, in general, support the researcher's intention of selecting two experimental programs to represent two different levels of complexity to test the research hypotheses.

5.4.1 Testing Group I Hypotheses

Group I hypotheses (section 4.2) were designed to evaluate the direction and the strength of the relationships between cognitive complexity variables and programming performance variables. Hypotheses 1a and 2a state that the performance of programmers in program comprehension tasks and program modification tasks do not correlate with their cognitive differentiation scores on a domain-specific REP Test. Hypotheses 1b and 2b state that the performance of programmers in program comprehension tasks and program modification tasks do not correlate with their cognitive differentiation scores on a domain-specific REP Test.

In evaluating Group I hypotheses, the degree of association between cognitive complexity and programming performance was tested using the performance scores in
comprehending and modifying both PROGRAM 1 and PROGRAM 2. To measure these associations, the Spearman rank correlation coefficients (rs) were first computed using the raw scores before sorting out the effect the subjects' experience might have had upon the relationship between cognitive complexity and programming performance. The results, however, showed no significant correlation between cognitive complexity and programming performance at the preselected level of significance (P = .05). Some of the correlation coefficients were even negative (opposite to the predicted direction). However, none of these negative coefficients were significant at P < .05.

To analyze further the relationship between cognitive complexity and programming performance, Spearman rank correlation coefficients (rs) were computed for (1) the association between the subjects' background and experience variables and programming performance variables and (2) the association between the subjects' background and experience variables and cognitive complexity variables. The results of this analysis are presented in Tables XV and XVI.

From the information presented in Table XV, the following variables were found to be associated with cognitive complexity and/or program comprehension variables (P < .10): student classification, GPA, number of COBOL courses, number of computer related courses, formal COBOL training, work experience in developing and maintaining programs, the length
TABLE XV

SPEARMAN RANK CORRELATION COEFFICIENTS ($r_s$) FOR BACKGROUND AND EXPERIENCE VARIABLES WHICH SIGNIFICANTLY CORRELATED WITH COGNITIVE COMPLEXITY AND PROGRAM COMPREHENSION AT $P < .10$.

<table>
<thead>
<tr>
<th>EXPERIENCE VARIABLES</th>
<th>COGNITIVE COMPLEXITY</th>
<th>PROGRAM 1</th>
<th>PROGRAM 2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>DIFFER.</td>
<td>INTEG.</td>
<td></td>
</tr>
<tr>
<td>Student classification</td>
<td></td>
<td></td>
<td>.27</td>
</tr>
<tr>
<td>GPA</td>
<td></td>
<td></td>
<td>-.32</td>
</tr>
<tr>
<td>Number of COBOL courses</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number of computer related courses at NTSU other schools</td>
<td>-.29</td>
<td>.21</td>
<td>.29</td>
</tr>
<tr>
<td>Number of days in COBOL training</td>
<td></td>
<td>.21</td>
<td></td>
</tr>
<tr>
<td>Number of months worked in: developing maintenance</td>
<td>-.27</td>
<td>.23</td>
<td>.24</td>
</tr>
<tr>
<td>Largest COBOL program ever written</td>
<td></td>
<td></td>
<td>.20</td>
</tr>
<tr>
<td>Familiarity and use of programming techniques</td>
<td>-.23</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Familiarity but not use of programming techniques</td>
<td></td>
<td></td>
<td>.31</td>
</tr>
<tr>
<td>Unfamiliarity with programming techniques</td>
<td></td>
<td></td>
<td>-.34</td>
</tr>
</tbody>
</table>
TABLE XVI

SPEARMAN RANK CORRELATION COEFFICIENTS (rs) FOR BACKGROUND AND EXPERIENCE VARIABLES WHICH SIGNIFICANTLY CORRELATED WITH COGNITIVE COMPLEXITY AND PROGRAM MODIFICATION AT P < .10.

<table>
<thead>
<tr>
<th>EXPERIENCE VARIABLES</th>
<th>COGNITIVE COMPLEXITY</th>
<th>MODIFICATION</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>DIFFER.</td>
<td>INTEG.</td>
</tr>
<tr>
<td>Age</td>
<td>.</td>
<td>-.26</td>
</tr>
<tr>
<td>Nationality</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>Student classification</td>
<td>.</td>
<td>-.38</td>
</tr>
<tr>
<td>Degrees obtained</td>
<td>.</td>
<td>-.26</td>
</tr>
<tr>
<td>GPA</td>
<td>.34</td>
<td>.</td>
</tr>
<tr>
<td>Number of COBOL courses</td>
<td>.</td>
<td>-.34</td>
</tr>
<tr>
<td>Number of computer related courses at NTsu other schools</td>
<td>:</td>
<td>:</td>
</tr>
<tr>
<td>Number of months worked in maintenance</td>
<td>.</td>
<td>.19</td>
</tr>
<tr>
<td>Number of languages known</td>
<td>.28</td>
<td>.</td>
</tr>
</tbody>
</table>

of the longest COBOL program ever written, and the familiarity and unfamiliarity with programming methodologies and techniques.

From the information presented in Table XVI, the following variables were found to be associated with cognitive
complexity and/or program modification performance variables (P < .10): student classification, age, nationality, GPA, number of COBOL courses, number of computer related courses, work experience in maintaining programs, and the number of programming languages known.

The results of the association between background and experience variables on the one hand and cognitive complexity and programming performance on the other hand suggested that the influence of the subjects' background and experience should be sorted out from the raw scores before evaluating the significance of the association between these two groups of variables (cognitive complexity and performance). Therefore, multiple regression analysis was applied to the data. In the regression analysis, the effect of the experience variables presented in Tables XV and XVI was sorted out of both cognitive complexity and programming performance scores. Spearman rank correlation coefficients (rs) were, then, computed using the residuals of the two group of variables. The resulting rs before and after sorting out the effect of the subjects' background and experience are presented in Table XVII.

From the information included in Table XVII, there is no change in either the direction or in the strength of the correlation coefficients as a result of sorting out the effect of background and experience variables. The only apparent change is that of the correlation coefficient
TABLE XVII

SPEARMAN CORRELATION COEFFICIENTS (rs) FOR THE ASSOCIATION BETWEEN COGNITIVE COMPLEXITY AND PROGRAMMING PERFORMANCE BEFORE AND AFTER SORTING OUT THE EFFECT OF SUBJECTS' BACKGROUND AND EXPERIENCE

<table>
<thead>
<tr>
<th></th>
<th>DIFFERENTIATION</th>
<th></th>
<th>INTEGRATION</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>BEFORE</td>
<td>AFTER</td>
<td>BEFORE</td>
<td>AFTER</td>
</tr>
<tr>
<td>Comprehending</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PROGRAM 1</td>
<td>.03</td>
<td>-.09</td>
<td>-.10</td>
<td>-.06</td>
</tr>
<tr>
<td>PROGRAM 2</td>
<td>.06</td>
<td>.09</td>
<td>-.20</td>
<td>-.19</td>
</tr>
<tr>
<td>Modifying</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PROGRAM 1</td>
<td>-.10</td>
<td>-.12</td>
<td>-.13</td>
<td>-.03</td>
</tr>
<tr>
<td>PROGRAM 2</td>
<td>-.13</td>
<td>-.16</td>
<td>.16</td>
<td>.21*</td>
</tr>
</tbody>
</table>

* significant at p < .10

between cognitive integration and PROGRAM 2 modification performance. This coefficient (rs = .21) became significant at P < .10.

To further explore the relationship of cognitive complexity to performance, the residual plots of the scores (Appendix G) were investigated. The investigation suggested that the relationships between cognitive differentiation scores and comprehension and modification scores were rather weak and that there was no sign of any pattern in these relationships. The investigation, however, suggested that there was a sign of a nonlinear monotonic pattern of
relationship between cognitive integration scores and PROGRAM 2 modification scores.

Therefore, some form of data re-expression was required to detect the nonlinear relationship between cognitive integration and PROGRAM 2 modification scores. In the absence of a theoretical guidelines to choose the appropriate transformation function in this situation, the trial and error method was the only available choice (1). Using this option, different transformation functions (e.g., square roots, cube roots, quadratic roots, and common logarithms) were applied to the residuals before analyzing these residuals using Spearman rank correlation coefficient (rs) procedure. The correlation analysis using the logarithmic transformation of the data produced the best description of the relationship between cognitive integration and PROGRAM 2 modification. The results of such analysis are presented in Table XVIII.

The information presented in Table XVIII suggests that there was a positive correlation ($r = .33$) between cognitive integration and program modification and that this relationship is significant at $P < .01$. None of the other correlation coefficients is significant at $P < .05$. Therefore, the results of the correlation analysis do not support the rejection of hypotheses 1a, 1b, and 2a; the results, however, do support the rejection of hypothesis 2b.
TABLE XVIII
PERSON CORRELATION COEFFICIENTS FOR THE ASSOCIATION BETWEEN COGNITIVE COMPLEXITY AND PROGRAMMING PERFORMANCE USING A LOG TRANSFORMATION OF THE RESIDUALS.

<table>
<thead>
<tr>
<th></th>
<th>DIFFERENTIATION</th>
<th>INTEGRATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMPREHENDING</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PROGRAM 1</td>
<td>-.16</td>
<td>.09</td>
</tr>
<tr>
<td>PROGRAM 2</td>
<td>-.22</td>
<td>-.06</td>
</tr>
<tr>
<td>MODIFYING</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PROGRAM 1</td>
<td>.06</td>
<td>-.15</td>
</tr>
<tr>
<td>PROGRAM 2</td>
<td>.02</td>
<td>.33**</td>
</tr>
</tbody>
</table>

** significant at p < .01 (one-tailed)

In summary, the conclusion reached from testing hypotheses 1a, 1b, 2a, and 2b is that there was no relationship between the subjects' cognitive differentiation scores and their informance scores in either program comprehension or program modification. There was no relationship between the subjects' cognitive integration scores and their performance scores in the program comprehension tasks. The subjects' performance scores in PROGRAM 2 modification, however, was positively correlated with their cognitive integration scores. This means the increase in the subject's integration level was accompanied by an increase in his/her performance in modifying a relatively complex program; however, the performance scores were increasing at a decreasing rate.
5.4.2. Testing Group II hypotheses:

Group II hypotheses were designed to evaluate the combined influence of cognitive complexity and program complexity on programming performance. Hypotheses 3a, 3b, 4a, and 4b state that there is no significant difference in the performance scores between the relatively high differentiating group and the relatively low differentiating group in (1) comprehending the relatively simple program, (2) comprehending the relatively complex program, (3) modifying the relatively simple program, and (4) modifying the relatively complex program. Hypotheses 5a, 5b, 6a, and 6b state that there is no significant difference in the performance scores between the relatively high integrating group and the relatively low integrating group in (1) comprehending the relatively simple program, (2) comprehending the relatively complex program, (3) modifying the relatively simple program, and (4) modifying the relatively complex program.

In order to test Group II hypotheses, each experimental group was classified into three subgroups: high, medium, and low in terms of their differentiation and integration scores respectively. Each subject was situated in one of the three subgroups based on the classification scheme stated earlier in section 4.5 (Mean score ± .8 Std. Div.).

Initially, the nonparametric Mann–Whitney U test was conducted to test for any significant difference in the differentiation and integration scores between the program
comprehension group and program modification group. The results of such test showed no significant differences between the two groups. However, it was felt that it would be more appropriate to use each group's mean scores and standard deviations to classify the group into high, medium, and low in terms of their differentiation and integration scores.

The Kruskal-Wallis one-way analysis of variance test was, then, applied to the data (after sorting out the subjects' background and experience effect) to test this group of hypotheses. This nonparametric test analyzed the differences in program comprehension and program modification scores among the high, medium, and low groups. The test results for

TABLE XIX

THE RESULTS OF KRUSKAL-WALLIS TEST WHERE COGNITIVE DIFFERENTIATION IS THE CLASSIFICATION FACTOR

<table>
<thead>
<tr>
<th></th>
<th>HIGH</th>
<th>MEDIUM</th>
<th>LOW</th>
<th>CHI-SQUARE</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMPREHENSION:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mean ranks for</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>H3a PROGRAM 1</td>
<td>N=12</td>
<td>N=15</td>
<td>N=16</td>
<td></td>
</tr>
<tr>
<td></td>
<td>22.3</td>
<td>21.9</td>
<td>21.9</td>
<td>0.0068</td>
</tr>
<tr>
<td>H3b PROGRAM 2</td>
<td>26.2</td>
<td>18.3</td>
<td>22.3</td>
<td>2.6103</td>
</tr>
<tr>
<td>MODIFICATION:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mean ranks for</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>H4a PROGRAM 1</td>
<td>N=13</td>
<td>N=23</td>
<td>N=11</td>
<td></td>
</tr>
<tr>
<td></td>
<td>23.2</td>
<td>23.8</td>
<td>25.5</td>
<td>0.1791</td>
</tr>
<tr>
<td>H4b PROGRAM 2</td>
<td>23.1</td>
<td>24.0</td>
<td>25.0</td>
<td>0.1177</td>
</tr>
</tbody>
</table>
hypotheses 3a, 3b, 4a, and 4b are summarized in Table XIX and the test results for hypotheses 5a, 5b, 6a, and 6b are summarized in Table XX.

From the information reported in Table XIX, classified by their differentiation scores, there is no significant difference among the three subgroups (high versus medium versus low) in their performance in comprehending either the relatively simple program (PROGRAM 1) or the relatively complex one (PROGRAM 2). Also, there is no significant difference among the three subgroups in their performance in modifying either the relatively simple program or the relatively complex one. These findings do not support the rejection of hypotheses 3a, 3b, 4a, and 4b. Therefore, it

**TABLE XX**

THE RESULTS OF KRUSKAL-WALLIS TEST WHERE COGNITIVE INTEGRATION IS THE CLASSIFICATION FACTOR

<table>
<thead>
<tr>
<th></th>
<th>HIGH</th>
<th>MEDIUM</th>
<th>LOW</th>
<th>CHI-SQUARE</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>COMPREHENSION:</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mean ranks for</td>
<td>N=10</td>
<td>N=21</td>
<td>N=11</td>
<td></td>
</tr>
<tr>
<td>H5a PROGRAM 1</td>
<td>26.8</td>
<td>20.4</td>
<td>18.8</td>
<td>2.5668</td>
</tr>
<tr>
<td>H5b PROGRAM 2</td>
<td>15.5</td>
<td>22.1</td>
<td>25.7</td>
<td>3.7558</td>
</tr>
<tr>
<td><strong>MODIFICATION:</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mean ranks for</td>
<td>N=9</td>
<td>N=28</td>
<td>N=10</td>
<td></td>
</tr>
<tr>
<td>H6a PROGRAM 1</td>
<td>23.4</td>
<td>23.5</td>
<td>25.8</td>
<td>.2192</td>
</tr>
<tr>
<td>H6b PROGRAM 2</td>
<td>33.7</td>
<td>22.0</td>
<td>20.8</td>
<td>5.5927**</td>
</tr>
</tbody>
</table>

** significant at p < .05 (one-tailed)
can be concluded that the high differentiating group did not perform better than the low differentiating group in comprehending or modifying either of the two experimental programs.

From the information presented in Table XX, classified in terms of their integration scores, there is no significant difference among the three subgroups' performance in comprehending the relatively simple program or the relatively complex one. Also, there is no significant difference among the three subgroups' performance in modifying the relatively simple program. The data analysis, however, showed a significant difference among the three subgroups' performance in modifying the relatively complex program (Chi-square = 5.5927, P < .05 (one-tailed)). The high integrating subgroup did not perform better than the low integrating subgroup in comprehending either of the two programs. The high integrating subgroup performed the same as the low integrating subgroup in modifying the relatively simple program; but the high integrating subgroup performed better than the low integrating one in modifying the relatively complex program. These findings do not support the rejection of hypotheses 5a, 5b, 6a and do support the rejection of 6b.

In summary, the findings resulting from the tests on Group II hypotheses are the following:
(1) The highly differentiating group (a) did not perform significantly better than the low differentiating group in comprehending the relatively simple program or the relatively complex one, (b) did not perform significantly better than the low differentiating one in modifying the relatively simple program or the relatively complex one.

(2) The highly integrating group (a) did not perform significantly better than the low integrating group in comprehending the relatively simple program or the relatively complex one, (b) did not significantly perform better than the low integrating group in modifying the relatively simple program, and (c) performed significantly better than the low integrating group in modifying the relatively complex program.

5.4.3. Testing Group III hypotheses:

Group III hypotheses were formally designed to evaluate the moderating effect of the subjects' background and programming experience on the relationship between cognitive complexity variables and programming performance. Hypotheses 7a and 7b state that when adjusted for programming experience, there is no difference between the performance of programmers who are ranked high in differentiation or integration and the performance of those who ranked low in comprehending a program of a relatively high complexity. Hypotheses 8a and 8b state that when adjusted for
programming experience, there is no difference between the performance of programmers who are ranked high in differ-
entiation or integration and the performance of those who ranked low in modifying a program of a relatively high
complexity.

As discussed earlier, in order to test Group I and II hypotheses, the possible influence of the subjects' back-
ground and experience on the relationship between cognitive complexity and programming performance had to be controlled. Control was achieved by sorting out the effect of the ex-
perience variables which correlated significantly ($P < .10$) with cognitive complexity variables and performance variables (Tables XV and XVI).

To formally evaluate the significance of the influence of the subjects' background and experience on the pro-
gramming performance and to reach a conclusion concerning Group III hypotheses, a two-part decision rule was followed. First, if the data analysis showed that cognitive complexity variables have no mean effects on the programming perform-
ance variables before or after controlling for the influence of experience, then that will be sufficient evidence to fail the rejection of the null hypotheses. Secondly, if the data shows that cognitive complexity variables have significant mean effects on the programming performance variables before or after controlling for the influence of experience, then
further analysis must be carried out to test for the significance of the possible influence of experience on these mean effects.

To provide information for the first part of the above decision rule, the Kruskal-Wallis test was performed on the data before and after adjusting for the effect of subjects' background and experience. The results of such analysis are presented in Table XXI.

From the information presented in Table XXI, the subjects' background and experience influenced the relationship between cognitive complexity and performance variables differently. This influence is evident from comparing the mean ranks for the subgroups using the raw and adjusted performance scores. While the control of experience did not reverse the direction of any of the original relationships, it affected the magnitude of some of them.

Following the first part of the above decision rule, the results did not support the rejection of hypotheses 7a, 7b, and 8a. For hypothesis 7a, the results were in the predicted direction where the highly differentiating subjects performed better than the low differentiating ones in comprehending the relatively complex program. The difference, however, is not significant even when experience was controlled ($P < .14$). For 7b and 8a the results were the opposite of what was predicted.
TABLE XXI

THE RESULTS OF KRUSKAL–WALLIS TEST USING THE PERFORMANCE SCORES BEFORE AND AFTER ADJUSTING FOR THE EFFECT OF THE SUBJECTS' BACKGROUND AND EXPERIENCE

<table>
<thead>
<tr>
<th></th>
<th>HIGH</th>
<th>MEDIUM</th>
<th>LOW</th>
<th>CHI-SQUARE</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>A. COMPREHENSION:</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>classifies by:</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1. Differentiation</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mean ranks for PROGRAM 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>With Raw scores</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>With adjusted scores</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>B. MODIFICATION:</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>classifies by:</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1. Differentiation</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**significant at p < .05**
There is not sufficient evidence to make a decision concerning hypothesis 8b at this stage of the analysis. When the effect of the subjects' background and experience was controlled, the relatively high integrating group performed better than the relatively low integrating group in modifying the relatively complex program ($P < .03$) (Table XXI). Following the second part of the decision rule stated earlier, the significance of the influence of experience (hypothesis 8b) was further evaluated. The portion of the variance in PROGRAM 2 modification scores that is due to experience was computed (the difference between the raw scores and the residual in the regression analysis). Then the Kruskal-Wallis test was applied to test for the

**Table XXII**

THE RESULTS OF KRUSKAL-WALLIS TEST USING THE PORTION OF PROGRAM 2 MODIFICATION SCORES WHICH IS DUE TO THE EFFECT OF THE SUBJECTS' BACKGROUND AND EXPERIENCE

<table>
<thead>
<tr>
<th>(cognitive integration is the classifying factor)</th>
<th>HIGH</th>
<th>MEDIUM</th>
<th>LOW</th>
<th>CHI-SQUARE</th>
</tr>
</thead>
<tbody>
<tr>
<td>N=10</td>
<td>N=21</td>
<td>N=11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mean ranks</td>
<td>29.9</td>
<td>21.9</td>
<td>24.5</td>
<td>2.3165</td>
</tr>
</tbody>
</table>
significance of the difference among the subgroups in terms of that portion, which is due to experience. The results of the analysis are presented in Table XXII.

From the information presented in Table XXII, the difference among the subgroups in terms of the portion of PROGRAM 2 modification scores that was due to experience was not significant (P < .15). This finding suggests that the effect of the subjects' background and experience on the differences among the subgroups in modifying the relatively complex program is not significant. Therefore, the results did not support the rejection of hypothesis 8b.

To summarize, the results of Group III hypotheses evaluation suggest that while experience moderates the relationships between cognitive complexity and performance variables, this moderating role was not significant. Furthermore, the differences in the subjects' experience did not account for the significant differences found among the performance of the subgroups, classified in terms of their cognitive integration, in modifying the relatively complex program.
CHAPTER BIBLIOGRAPHY


CHAPTER VI

CONCLUSION

6.1 SUMMARY

In order to evaluate what has been accomplished in this study, it is instructive to return to Chapter I for re-examination of the original mission undertaken by the study. In that chapter it was stated that this study was motivated by the fact that, although the importance of psychological factors on computer programming behavior is well recognized in the literature, little evidence is available to support this contention. The problem of determining the impact of these factors on the performance of programming-related tasks is complicated by: (1) differences in individuals, (2) variations in the programming tasks, and (3) variations in the programming environments.

To enhance our understanding of the influence of these psychological factors on programming productivity, the objectives of this research were (1) to evaluate the direction and the strength of the association between cognitive complexity (differentiation and integration) and performance in comprehending and modifying computer programs, (2) to evaluate the influence of the individual differences of programmers in terms of their cognitive complexity (differentiation and integration) on their performance in
comprehending and modifying programs of different complexity, and (3) to assess the moderating influence of experience on the relationship between cognitive complexity and program comprehension and modification.

To achieve the above objectives, an experiment was conducted in a classroom setting. Ninety-three graduate and undergraduate students participated in the experiment. The participants represented different levels of computer and programming experience. In the first phase of the experiment, a background questionnaire was administered to collect experience and other demographic information. Also, a domain-specific REP Test was administered to collect cognitive complexity (differentiation and integration) information.

In the second phase of the experiment, the subjects were randomly assigned to either the program comprehension group or to the program modification group. Both groups used the same experimental COBOL programs. The two programs were selected to represent two different levels of complexity as measured by (1) the number of executable lines of code (LOC), (2) Halstead's effort metric (E), and (3) McCabe's cyclomatic number \( V(G) \). Based on these complexity measures, one of the two program is relatively simple and the other is relatively complex.

The program comprehension group was asked to perform two comprehension tasks using the two programs. For each
task, the subjects were asked to use the time allocated to study the program given to them. Then they were asked to turn over the programs before they were asked to recall a functionally equivalent program to the one they studied in the given amount of time.

The program modification group was given the same programs, one at a time, and asked to add and/or modify the necessary statements to each program to satisfy certain requirements. The modification tasks were also timed. The two dependent variables of the study (program comprehension and modification) were represented by the sum of scores received for each comprehension or modification subtask. Two overall scores were given for each subject in the two groups: one for comprehending or modifying the relatively simple program and one for comprehending or modifying the relatively complex one. A discussion of the results is presented in the section which follows.
6.2 DISCUSSION OF RESULTS

The data collected in this experiment were analyzed using parametric and nonparametric tests to evaluate the three groups of research hypotheses (section 5.2). Spearman rank correlation coefficient (rs) and regression analysis were used to test the first group of hypotheses (H1a through H2b) which evaluates the relationships between cognitive complexity—differentiation and integration—on the one hand and program comprehension and modification on the other hand. For the second group of hypotheses (H3a through H6b), the nonparametric Kruskal-Wallis one-way ANOVA was used to evaluate the significance of the difference among the performance of the high, medium, and low subgroups—as classified by their differentiation and integration scores—in comprehending and modifying programs of different complexity. Kruskal-Wallis test was also used to test the third group of hypotheses (H7a through H8b) which evaluates the influence of experience on the difference among the performance of the three subgroups (high, medium, and low) in comprehending and modifying the relatively complex program.

The first set of hypotheses was designed to explore the relationship between cognitive complexity and programming performance variables. The results of evaluating this group of hypotheses suggest that cognitive complexity variables
seem to associate differently with the two dependent variables (program comprehension and program modification). Cognitive differentiation did not associate significantly with either one of the two dependent variables. However, while its positive association with comprehending the relatively complex program is in the predicted direction, its negative association with modifying the relatively complex program was not. It was thought that these findings might be confounded by the subjects' experience. However, controlling for the influence of experience on the relationship of cognitive complexity to program comprehension and modification did not significantly change the original findings.

Modifying the relatively complex program, however, was found to have a significant positive association (.33) with cognitive integration. While this correlation will not seem large to perusers of the engineering or physical science literature, its magnitude is typical of significant results reported in human factors research. This relationship between cognitive integration and the modification of the relatively complex program, however, is nonlinear and the analysis of the data indicated that the best fit of such relationship is a logarithmic function. The function produced by the regression analysis is
LogY = .75 + .34 LogX + error term

whereas Y is the predicted modification performance and X is the cognitive integration score. Therefore, the performance in modifying a program with a level of complexity similar to the one used in this study increases as cognitive integration level increases. The increase in the performance, however, is steeper in the lower range of cognitive integration than it is in the upper range. This conclusion, of course, is tentative and further verification of this finding is needed.

The second set of hypotheses was designed to evaluate the performance differences among the high, medium, and low subgroups when classified in terms of their cognitive differentiation and integration scores separately. When cognitive differentiation was the classifying factor, there was no significant difference among the three subgroups (high, medium, and low) in either comprehending or modifying the relatively simple program. This finding was not surprising, based upon the cognitive complexity hypotheses of Schroder and his associates (12). Programmers tended to apply low differentiating and integrating processes when involved in a programming task (e.g., comprehension and modification) where the environmental complexity was relatively low. Therefore, it seems that comprehending and modifying a relatively simple program does not stimulate
the programmer’s ability to discover and integrate the assertions which describe its operations and functions.

The above conclusion is partially supported by the finding that the highly differentiating subjects performed slightly better than the low differentiating subjects in comprehending the relatively complex program (P < .14). Comprehending a relatively complex program seems to stimulate the high differentiating programmer to apply more differentiation processes to perceive and understand that program than the low differentiating does.

When classified in terms of their cognitive integration scores, one surprising result was found within the program comprehension group. While not significant (P < .08), the highly integrating group performed worse than the low integrating group in comprehending the relatively complex program. This finding is contrary to what was predicted.

The poor performance of the relatively high integrating group in comprehending the relatively complex program may be due to the characteristics of the task itself. Program comprehension is viewed as a perception-intensive domain. In such a domain, an individual can solve problems and tasks mainly in a perceptual-memory fashion (8, p. 116). In doing so, one can expect that programmers may not need as much integration of the learned materials as they need to differentiate and decompose the program into pieces and memorize them. Also, the highly differentiating programmers are
expected to do better than the low differentiating programmers in comprehending relatively complex programs.

This view is further supported by two results found in the study: (1) classified in terms of their cognitive differentiation scores, the highly differentiating subjects performed slightly better than the low differentiating ones in comprehending the relatively complex program ($P < .14$), and (2) the negative correlation ($r_s = .16$, $P < .10$) which was found between cognitive differentiation and cognitive integration. This latter finding is consistent with what is reported by Larreche (8) and Gardner and Schoen (1962), cited in (6, p. 115); however, the measuring instruments are different.

As a result of the negative correlation between cognitive differentiation and integration, within the program comprehension group, the programmers classified as being high in terms of their integration ability were also classified as medium or low in terms of their differentiation ability. Therefore, the relatively high integrating subjects performed poorly in comprehending the relatively complex program, not because of their high integrating ability but, perhaps because of their low differentiating ability, which is more essential in performing such perception-intensive task. This view, however, is tentative and additional research is needed.
On the other hand, the findings for the program modification group were as predicted. There was no significant differences between the performance of the high integrating subjects and low integrating subjects in modifying the relatively simple program. However, the high integrating subjects performed significantly better than the low integrating subjects in modifying the relatively complex program.

Based on these findings, it seems that modifying a computer program is a processing-intensive domain. In this domain, an individual needs to go to a deeper level of intellectual processing to solve a problem or task (8). In modifying a computer program, the high integrating programmers are more likely to apply the integrative ability they possess to perform the task. Such task requires the programmer to perform information search processes to identify and integrate the relevant assertions, which describe the functions of the program, to determine and implement their solutions to the problem.

These findings support the cognitive complexity hypotheses drawn by Schroder and his associates (12). The findings are also consistent with findings of the integrative complexity research in the other decision-making domains (i.e., 4; 8; 16). However, it should be noted that this study did not attempt to evaluate the information processing characteristics of the subjects while performing
comprehension and modification tasks. Rather, the gross outcome of the decision making process was the main concern of the study. Given this qualification, the findings of this study are comparable and consistent with those reached by Claunch (1964), cited in (6, p. 162), and Amernic and Beechy (1) where the high integrating persons did better than the low integrating ones in performing tasks of a relatively high complexity; but the two groups performed similarly in the tasks of relatively low complexity.

The third set of hypotheses was designed to evaluate the influence of experience on the performance differences among the different subgroups as classified in terms of the two dimensions of cognitive complexity—differentiation and integration. It was found that experience variables associated \((P < .10)\) differently with cognitive complexity and performance variables. More specifically, cognitive differentiation was found to associate positively with grade point average (GPA) and the number of COBOL courses; but it associated negatively with work experience in developing programs and in familiarity and use of programming methodologies and techniques. Cognitive integration was found to associate positively with the number of computer related courses, formal training in COBOL, and work experience
in program maintenance but to associate negatively with GPA, age, student classification, and the number of degrees obtained.

While the correlates of cognitive differentiation were not expected to be the same correlates of cognitive integration (5), the negative association between cognitive integration and age is surprising. Harvey et al. (7) hypothesized that integrative complexity increases with age. However, the results of this study as well as the results of the studies reported in Goldstein and Blackman (6, p.155) do not support this hypothesis. The results of this study, therefore, support the conclusion that there are insufficient data to reach a clear conclusion on the relationship of integrative complexity to such demographic variables (6, p.156).

The performance of the program comprehension group associated positively with a number of experience variables such as familiarity with and use of the common programming methodologies and techniques, student classification, GPA, number of COBOL courses, number of computer related courses, work experience in program development, and the number of lines of code in the largest COBOL program they had ever written. On the other hand, the program comprehension performance was also found to associate negatively with the number of common programming methodologies and techniques the subjects were unfamiliar with. The performance of the
modification group was found, however, to associate positively only with the number of computer related courses.

The resulting correlations between computer and programming experience variables and programming performance, especially program comprehension, are consistent with those reported in (9; 10; 11; 15) where measures similar to those reported in this study were used. However, sorting out the influence of the experience variables from the raw data scores slightly changed the original results of the relationship between cognitive complexity and program comprehension and modification. The significant difference in the performance of the high integrating subjects and the low integrating subjects in modifying the relatively complex program was not due to the difference in the subjects' experience.

A further analysis was performed to evaluate the overall influence of experience on program comprehension and modification. Experience variables were included as covariates in unbalanced analysis of covariate models using the rank data for cognitive complexity and performance variables. The resulting covariate model for program modification explained 43% of the variance in program comprehension performance ($P < .001$). The subjects' GPAs and their unfamiliarity with programming methodologies and
techniques were the two experience factors which significantly (P < .05) contributed most to the model. The covariate models for program modification, however, explained 21% (P < .21) of the variance when differentiation was the classifying factor and 30% (P < .15) when integration was the classifying factor. The number of computer related courses was the only experience variable that contributed significantly (P < .05) to the model.

The findings of the covariate models are consistent with the findings reached in testing the first and second set of hypotheses. While cognitive complexity variables—differentiation and integration—produced little explanation to the variance in program comprehension performance, such variance was explained better by the experience variables. To the contrary, while cognitive differentiation and experience did not produce sufficient explanation to the variance in program modification, this variance was better explained by cognitive integration.

Overall, the results of evaluating the research hypotheses suggest the following:

1) There is a rather weak support to the applicability of cognitive complexity theory to the domain of computer program comprehension situations. The little evidence obtained from the study, however, suggests that program comprehension is a perception-intensive domain in which a programmer perceives and comprehends a program in a
perceptual-memory fashion. In performing such tasks, programmers are more likely to apply more differentiating processes rather than integrating processes.

(2) There is a strong support to the applicability of cognitive complexity theory to the domain of computer program modification situations. The evidence obtained from this study suggests that program modification is a processing-intensive domain in which a programmer needs to use a deeper level of intellectual processing to solve the modification problem. In performing such tasks, programmers are more likely to apply more integrating processes rather than differentiating processes.

(3) Programmer's experience seems to explain more variance in program comprehension situations than it does in program modification situations; and it does not seem to significantly moderate the relationship of cognitive complexity to program comprehension and modification.
6.3 **IMPlication of the Study**

The purpose of this study was to extend the basic knowledge related to the influence of programmers' cognitive complexity (differentiation and integration) and experience on programming-related tasks under different levels of environmental complexity.

The results of this study were obtained in a quasi-experimental situation with the subjects being graduate and undergraduate students in the Business Computer Information Systems program at North Texas State University. This sample is certainly not exactly representative of the total population of professional programmers in various organizations. Differences in the programming behavior of individuals in field situations should be larger and should be more heterogeneous than the studied sample.

If it is assumed that the results of the present study may be generalized to field situations, the implications for the findings is very important indeed. The findings of this study should benefit both researchers in academia and practitioners in industry.

For the researchers in academia, the findings of this study offer some support to the applicability of cognitive complexity theory in the domain of computer programming, especially in program modification situations. This theory
lends itself to the researchers in the psychology of pro-
gramming field of research to provide some explanation to
one of the important issues of computer programming—that
is, the differences in programming behavior and the influ-
ence of psychological factors on that behavior. The results
of this study suggest that the programmer's cognitive com-
plexity is one of the determining factors of his or her
performance in modifying programs of relatively high
complexity.

The potential impact of the programmer's psychological
factors on programming-related tasks is self-evidence in
the computer programming literature. The ultimate goal of
the research efforts has been the development of a psycho-
logical theory to explain the programmer's behavior while
performing programming-related tasks (i.e., 2; 3; 14). The
problem, however, Sheil (13) explains, is "...although some
psychological theory is very suggestive, it usually lacks
the robustness and precision required to yield exact pre-
dictions for behavior as complex as programming" (p. 102).

The evidence indicated in this study relating to
the predictions that can be made about the programmer's
performance in program modification situations should con-
tribute to the continuing efforts to develop such "robust"
theory of programming behavior. This potential theory should
embrace at least three factors and their relationships: (1)
the psychological characteristics of the programmer, (2)
the programming tasks' characteristics, and (3) the characteristics of the programming environment. The findings of this study should be helpful from this theory building prospective.

For practitioners in industry the applicability of the results of this study is apparent. Given the growing importance of programming in general and software maintenance in particular, the findings of this study should be of interest to the policy makers in industry as well as in educational institutes.

The results of this study suggest that the high integrating programmers are more likely to perform better than the low integrating ones in program maintenance-related tasks. Also, there is modest evidence that the high differentiating programmers are more likely to perform better than the low differentiating ones in program comprehension situations.

Therefore, cognitive complexity measures should be used to predict programmers' performance in these programming-related tasks. Effective hiring and training policies of computer programmers should acknowledge the cognitive complexity characteristics of those programmers before assigning them to specific training programs and programming tasks. In addition, in educational institutions, the students' cognitive complexity characteristics may be
considered as a factor in predicting students' performance in computer and programming relating courses.
6.4 FUTURE RESEARCH

Any research failing to suggest future directions for follow-on research should probably be considered remiss. According to Nobel Prize Winner Alfred Kaster, "All knowledge is provisional—never final." This is certainly the case in this field where many questions remain unanswered. It is believed that this research points out several areas requiring more explicit and intensive investigations. Some of the areas for future research are these:

1) Replication. Unlike the physical science, the general consensus in the social sciences demotes replication of previous research to second-class standing. This is unfortunate because it is only through replication of scientific inquiry that generalization of research results is facilitated. For this field of research—the psychology of programming—a replication of this study in a business setting environment that permits professional programmers to participate will promote the findings of this study.

2) A further investigation of the information processing characteristics in performing the programming-related tasks. While this research was designed to investigate the "gross" outcome of the performance of programming-related tasks, there is no information available on the impact of the differences in the programmers' cognitive complexity on
the information search and use. A research approach which permits the collection of protocol data during the performance of programming-related tasks should lead to a better explanation of the differences in programming behavior.

(3) An extension of the scope of the research to include other programming-related tasks such as analysis, design, coding, testing, debugging, and documentation. As pointed out earlier in this study, cognitive complexity theory relates the personality variable to the environmental variable in an articulated manner. Individuals (i.e., systems analysts, systems designers, programmers... etc.) vary in the complexity of their abilities to process information; the environment varies in the complexity of the information it contains. These two dimensions are related to each other, and the results should be investigated in the performance of the different tasks throughout the software life cycle. The evidence of this possible research should improve our capability to match the different persons to the different tasks in managing and controlling software projects.

(4) An investigation of the validity of the cognitive complexity measuring instrument—the REP Tet—developed and used in this study. The major purpose of the developed instrument was to provide a measure of cognitive
complexity in the Data Processing and Programming domain. The "predictive validity" of the instruments is partially indicated in predicting the subjects' performance of the program modification tasks. Further investigations, however, are needed to test the "content validity" and "construct validity" of the instrument. At least two validation tests of the developed instrument are recommended: (a) Repeating the same instrument parallel with another one which uses Data Processing and Programming stimuli different from the ones used in this study. A comparison of the results of the two instruments should test the content validity of the original instrument. (b) Repeating the same instrument parallel with one or more of the cognitive complexity measures originally used in Psychology research. A factor analysis should provide informative results on the construct validity of the instrument used in this study.

(5) An investigation of the possibility of developing comprehensive metrics which acknowledge the different aspects of program complexity. The notion of optimum level of environmental complexity is an important aspect of cognitive complexity as viewed by Schroder et al. (12). Applied to programming-related situations, this concept means that programmers will benefit more when using programs that generate levels of environmental complexity.
closer to their own optimum level of environmental complexity. Measuring the optimum level of program complexity for a given programmer, however, is certainly a complex problem; and to relate the characteristics of a program to the level of complexity it generates is another problem. While the first problem should be interesting to cognitive complexity researchers in general, the second one should be interesting to software engineering researchers in particular. The current program complexity metrics measure certain aspects of program complexity. Comprehensive complexity metrics should be more informative in classifying programs into different categories in terms of their complexity. Further investigations, therefore, are needed to develop such metrics for program complexity as a necessity for the matching of a programmer's cognitive complexity to an "optimum" level of program complexity.
CHAPTER BIBLIOGRAPHY


APPENDIX A

DOMAIN-SPECIFIC REP TEST
THE ROLE CONSTRUCT REPERTORY TEST

This packet consists of a RESPONSE SHEET (in a matrix form) and the instructions to fill it out. The RESPONSE SHEET is composed of two components: (1) the columns of the matrix represent various constructs used in data processing and programming, and (2) the rows of the matrix represent the required associations between various members of these constructs in terms of the respondent-supplied descriptors. The matrix consists of 100 cells (10 rows by 10 columns).

You should evaluate the likeness (similarity) and the unlikeness (difference) of the various programming constructs. This is accomplished by following the procedure demonstrated in the following example:

EXAMPLE:

This example demonstrates the procedure to be used to complete the RESPONSE SHEET though the constructs used are quite different. In this example organizational positions are employed as constructs.

FIRST STEP:

Beginning with Row A, notice that Row A has two yellow cells. This indicates that you are first to consider the two organizational positions whose names appear above the yellow circles. Think carefully about these two positions and the activities they require.

ARE THE TWO ORGANIZATIONAL POSITIONS ALIKE IN SOME ONE WAY?

If they seem alike to you in some one way, write the way in which these two positions are alike in the pink space (Row A, Column 1).

RESPONSE SHEET

Example: Auditor Bookkeeper Controller Tax Attorney Cost Accountant

Row A

Column 1

Column 2

accurate
Now look across the other positions contained in this list. Is one of these positions different from the two which are alike? If so, write in the green space (Row A, Column 2) the way in which this one is unlike the two that are alike.

### Example:

<table>
<thead>
<tr>
<th></th>
<th>Auditor</th>
<th>Bookkeeper</th>
<th>Controller</th>
<th>Tax Attorney</th>
<th>Cost Accountant</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Row A</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Column 1</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>accurate</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>creative</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

If you cannot find another position which is different from the two which are alike, leave the green space blank (Row A, Column 2).

### Example:

<table>
<thead>
<tr>
<th></th>
<th>Auditor</th>
<th>Bookkeeper</th>
<th>Controller</th>
<th>Tax Attorney</th>
<th>Cost Accountant</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Row A</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Column 1</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>accurate</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>-----</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

After you finish Row A, complete Row B, Row C, etc. Follow the same instructions.

**Are the two positions different in some way?**

If you cannot find a way in which the two positions are alike, think about them once again. If they are not alike in some way, perhaps the two positions are different in some one way. If you see that the two positions are different in some way, write in the pink space (Row A, Column 1) the description which fits the position in the left circle.

### Example:

<table>
<thead>
<tr>
<th></th>
<th>Auditor</th>
<th>Bookkeeper</th>
<th>Controller</th>
<th>Tax Attorney</th>
<th>Cost Accountant</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Row A</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Column 1</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>repetitious</td>
</tr>
</tbody>
</table>
Then, write in the green space (Row A, Column 2) the description which fits the position in the right circle.

Example:

<table>
<thead>
<tr>
<th></th>
<th>Auditor</th>
<th>Bookkeeper</th>
<th>Controller</th>
<th>Tax Attorney</th>
<th>Cost Accountant</th>
</tr>
</thead>
<tbody>
<tr>
<td>Column 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Column 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Row A</td>
<td></td>
<td>repetitious</td>
<td></td>
<td></td>
<td>versitile</td>
</tr>
</tbody>
</table>

If you cannot see a similarity or a difference for the two positions, leave blanks.

AFTER YOU FINISH ROW A, COMPLETE ROW B, ROW C, ETC. FOLLOW THE SAME INSTRUCTIONS.

SECOND STEP:

For Row A, look over the pink description you wrote under Column 1 and the green description you wrote under Column 2. Notice that between your two descriptions is a rating scale -6 -5 -4 -3 -2 -1 0 +1 +2 +3 +4 +5 +6, use your descriptions and this rating scale to give your impression of each position in Row A.

Example:

<table>
<thead>
<tr>
<th></th>
<th>Auditor</th>
<th>Bookkeeper</th>
<th>Controller</th>
<th>Tax Attorney</th>
<th>Cost Accountant</th>
</tr>
</thead>
<tbody>
<tr>
<td>Column 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Column 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Row A</td>
<td>+2</td>
<td>-5</td>
<td>0</td>
<td>+4</td>
<td>-4</td>
</tr>
<tr>
<td>Row B</td>
<td>-2</td>
<td>-6</td>
<td>+3</td>
<td>+5</td>
<td>0</td>
</tr>
</tbody>
</table>

In the example above, both Auditor (+2) and Tax Attorney (+4) are rated as being "creative". Since Tax Attorney has a higher rating than Auditor, this indicates that tax attorney is more "creative" than auditor. On the other hand, both Book Keeper (-5) and Cost Accountant (-4) are rated as being "accurate". Since Book Keeper has a higher rating than Cost Accounting, this indicates that Book Keeper is more "accurate" than Cost Accountant.
Begin on Row A and give your impression of the first programming construct using the rating scale. Then give your impression of the second one and so on until all of the spaces (cells) in Row A are filled. Then go to Row B. Use your descriptions for Row B and the rating scale to give your impression of the first programming construct, then the second and so on. Go on to Row C, etc. until you have filled in all of the squares. Of course, if there is no description under Column 2 (say you left it blank), no rating can be made using the right portion of the scale.

**RO (0) RATING:**

Use a 0 rating when you do not know the construct well enough to give your impression.

Use a 0 rating when neither descriptor fits the construct you are trying to rate.

**AN EXERCISE**

Before moving on and filling out the RESPONSE SHEET, be sure that you understand the above instructions. If the instructions are still unclear ask for more information. As an exercise, fill out the following 3 by 3 sample matrix. In this matrix, three data processing positions are used as constructs.

<table>
<thead>
<tr>
<th>Systems Analyst</th>
<th>Junior Programmer</th>
<th>Computer Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Column 1</td>
<td>0</td>
<td>Column 2</td>
</tr>
<tr>
<td>-1 -2 -3 -4 -5 -6</td>
<td>+1 +2 +3 +4 +5 +6</td>
<td></td>
</tr>
<tr>
<td>-1 -2 -3 -4 -5 -6</td>
<td>+1 +2 +3 +4 +5 +6</td>
<td></td>
</tr>
<tr>
<td>-1 -2 -3 -4 -5 -6</td>
<td>+1 +2 +3 +4 +5 +6</td>
<td></td>
</tr>
</tbody>
</table>

Now go on and fill out the RESPONSE SHEET.
<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
<th>I</th>
<th>J</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Memory Buffer Processing</td>
<td>Data File</td>
<td>Data Index</td>
<td>Data Computation</td>
<td>Search Key</td>
<td>Data Editing</td>
<td>Database</td>
<td>Logical Record</td>
<td>Physical Block</td>
</tr>
<tr>
<td>2</td>
<td>Column 1</td>
<td>Column 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>-6-5-4-3-2-1</td>
<td>+1+2+3+4+5+6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>-6-5-4-3-2-1</td>
<td>+1+2+3+4+5+6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>-6-5-4-3-2-1</td>
<td>+1+2+3+4+5+6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>-6-5-4-3-2-1</td>
<td>+1+2+3+4+5+6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>-6-5-4-3-2-1</td>
<td>+1+2+3+4+5+6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>-6-5-4-3-2-1</td>
<td>+1+2+3+4+5+6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>-6-5-4-3-2-1</td>
<td>+1+2+3+4+5+6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>-6-5-4-3-2-1</td>
<td>+1+2+3+4+5+6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Code: 

101
Respondent Preliminary Questionnaire

Answer the following questions in the spaces provided. Try to select the best answer. If you find the answers do not adequately indicate your personal situation, use the one answer that comes closet.

(1) Soc. sec. #: ____________________________

(2) Class: __ Section __________

(3) Age: ______ Years

(4) Sex: Male ___ Female ___

(5) Nationality: ____________________________

(6) Current major: _______ Minor _______

(7) Classification: Freshmen__ Junior__ Sophomore_

                    Senior__ Graduate__

(8) College degree(s) held (e.g., BA, BS, BBA, MBA, etc.):

   Degree  Major  Minor(s)  Year granted
   _____  _____  _____  _________
       _____  _____  __________

(9) Grade-point average at the beginning of the current semester: ______

(10) The academic courses completed in Information Systems and Computer Science:

   (a) At NTSU:

      1. ___________________________  6. ___________________________
      2. ___________________________  7. ___________________________
      3. ___________________________  8. ___________________________
      4. ___________________________  9. ___________________________
      5. ___________________________ 10. ___________________________

   (b) At other schools: (Please write course code and course description).

   Course Code  Course description
   ____________  __________________________
   ____________  __________________________
   ____________  __________________________
(11) Formal programming-training sessions attended (beyond college courses):

<table>
<thead>
<tr>
<th>Subject matter</th>
<th>Time (in days)</th>
<th>Language(s) used</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(12) Have you ever been employed in a full time or part time programming job?

Yes ______ No ______

If your answer is "No", please go on to Question 14.

(13) Write the number of months experience as employed in a programming job.

<table>
<thead>
<tr>
<th>Number of months in program development</th>
<th>Number of months in program maintenance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Systems programmer</td>
<td></td>
</tr>
<tr>
<td>2. Applications programmer</td>
<td></td>
</tr>
<tr>
<td>3. Programmer/analyst</td>
<td></td>
</tr>
<tr>
<td>4. Systems analyst</td>
<td></td>
</tr>
<tr>
<td>5. Other (please specify):</td>
<td></td>
</tr>
</tbody>
</table>

(14) Indicate below the languages you are familiar with. Check only the languages where you have actually written at least one program.

<table>
<thead>
<tr>
<th>COBOL</th>
<th>LISP</th>
</tr>
</thead>
<tbody>
<tr>
<td>BASIC</td>
<td>C</td>
</tr>
<tr>
<td>FORTRAN</td>
<td>ALGOL</td>
</tr>
<tr>
<td>PL/1</td>
<td>APL</td>
</tr>
<tr>
<td>Ada</td>
<td>Other (please specify):</td>
</tr>
<tr>
<td>SNOBOL</td>
<td></td>
</tr>
<tr>
<td>ASSEMBLER</td>
<td></td>
</tr>
<tr>
<td>PASCAL</td>
<td></td>
</tr>
</tbody>
</table>

(15) What is the largest program you have ever written and in what language?

1. < 500 lines
2. => 500 and < 1000 lines LANGUAGE:_______
3. => 1000 and < 2000 lines
4. > 2000

(16) What is the largest program you have ever written in COBOL?

1. < 500 lines
2. => 500 and < 1000 lines
3. => 1000 and < 2000 lines
4. > 2000
(17) I have used structured-programming techniques in

____ All the programs I have written.
____ Most of the programs I have written.
____ Some of the programs I have written.
____ None of the programs I have written.

(18) Have you had to follow specific structured-programming techniques in developing and maintaining your programming assignments?

____ Yes (please specify):

____ No

(19) The following is a list of programming methodologies and techniques that you may have been exposed to at some time in your programming career. Some of these you will have used in the programs you have written or modified and they will be very familiar to you. Others are methodologies or techniques that are new to you or you are familiar with the name but are not sure how it works. Then, there will be some methodologies and techniques that you have not actually programmed with them yet.

So, for each item, write a number (1, 2, or 3) that most represents your familiarity with that methodology or technique.

1 = I understand this methodology/technique and have programmed with it.
2 = I understand this methodology/technique but not programmed with it.
3 = I am not familiar with this methodology/technique.

HIPO
Warnier/Orr Diagrams
Flowcharts
Decision tables
Pseudocode
Data Flow Diagrams
Narratives
Block diagrams
Program analyzers
Code generators
Debug packages
Top-down development
Structured walkthrough

Others (please specify):

____
____
____
____
DATA DIVISION.

FILE SECTION.

FD ORDER-FILE
LABEL RECORDS ARE OMITTED.

01 ORDER-RECORD.
  05 PURCHASE-ORDER-IN PIC X(05).
  05 PURCHASE-DATE-IN PIC X(06).
  05 VENDOR-ID-IN PIC X(03).
  05 VENDOR-NAME-IN PIC X(20).
  05 ITEM-NUMBER-IN PIC 9(06).
  05 DESCRIPTION-IN PIC X(20).
  05 PURCHASE-QUANTITY-IN PIC 9(04).
  05 COST-IN PIC 9(04)V99.
  05 TAX-IN PIC 9(03)V99.
  05 DISCOUNT-IN PIC 9(03)V99.

FD REPORT-FILE
LABEL RECORDS ARE OMITTED.

01 REPORT-RECORD PIC X(133).

WORKING-STORAGE SECTION.

01 WORKING-RECORD.
  05 FILE-STATUS PIC X(04) VALUE SPACES.
  05 PURCHASE-ORDER-W8 PIC X(05) VALUE SPACES.

01 HEADING-1.
  05 FILLER PIC X(36) VALUE SPACES.
  05 FILLER PIC X(43) VALUE 'PURCHASE ORDER LISTING'.

01 HEADING-2.
  05 FILLER PIC X(02) VALUE SPACES.
  05 FILLER PIC X(27) VALUE 'PURCHASE ORDER'.
  05 FILLER PIC X(25) VALUE 'VENDOR'.
  05 FILLER PIC X(14) VALUE 'INVENTORY ITEM'.

01 HEADING-3.
  05 FILLER PIC X(02) VALUE SPACES.
  05 FILLER PIC X(14) VALUE ALL '—'.
  05 FILLER PIC X(04) VALUE SPACES.
  05 FILLER PIC X(26) VALUE ALL '—'.
  05 FILLER PIC X(03) VALUE SPACES.
  05 FILLER PIC X(28) VALUE ALL ' —'.
  05 FILLER PIC X(03) VALUE SPACES.
  05 FILLER PIC X(10) VALUE 'QUANTITY'.
  05 FILLER PIC X(08) VALUE 'COST PER'.

01 HEADER-4.
  05 FILLER PIC X(01) VALUE SPACES.
  05 FILLER PIC X(10) VALUE 'NUMBER'.
  05 FILLER PIC X(09) VALUE 'DATE'.
  05 FILLER PIC X(06) VALUE 'ID'.
  05 FILLER PIC X(23) VALUE 'NAME'.
  05 FILLER PIC X(08) VALUE 'NUMBER'.
  05 FILLER PIC X(22) VALUE 'DESCRIPTION'.
  05 FILLER PIC X(13) VALUE 'PURCHASED'.
  05 FILLER PIC X(12) VALUE 'ITEM'.
  05 FILLER PIC X(06) VALUE 'TAX'.
  05 FILLER PIC X(08) VALUE 'DISCOUNT'.

PROCEDURE DIVISION.

000-MAIN-CONTROL.
PERFORM 100-INITIALIZATION.
PERFORM 300-HEADING.
PERFORM 500 PROCESS
UNTIL FILE-STATUS EQUAL TO 'DONE'.
PERFORM 900-CLOSE.
STOP RUN.

100-INITIALIZATION.
OPEN INPUT ORDER-FILE OUTPUT REPORT-FILE.
READ ORDER-FILE AT END
MOVE 'DONE' TO FILE-STATUS.

300-HEADING.
WRITE REPORT-RECORD FROM HEADING-1
AFTER ADVANCING TOP-OF-PAGE.
WRITE REPORT-RECORD FROM HEADING-2 AFTER ADVANCING 2 LINES.
WRITE REPORT-RECORD FROM HEADING-3 AFTER ADVANCING 1 LINES.
WRITE REPORT-RECORD FROM HEADING-4 AFTER ADVANCING 1 LINES.

500-PROCESS.
MOVE PURCHASE-ORDER-IN TO PURCHASE-ORDER-OUT.
MOVE PURCHASE-DATE-IN TO PURCHASE-DATE-OUT.
MOVE VENDOR-ID-IN TO VENDOR-ID-OUT.
MOVE VENDOR-NAME-IN TO VENDOR-NAME-OUT.
MOVE ITEM-NUMBER-IN TO ITEM-NUMBER-OUT.
MOVE DESCRIPTION-IN TO DESCRIPTION-OUT.
MOVE PURCHASE-QUANTITY-IN TO PURCHASE-QUANTITY-OUT.
MOVE COST-IN TO COST-OUT.
MOVE TAX-IN TO TAX-OUT.
MOVE DISCOUNT-IN TO DISCOUNT-OUT.
PERFORM 700-WRITE-DETAIL.
READ ORDER-FILE AT END
MOVE 'DONE' TO FILE-STATUS.

700-WRITE-DETAIL.
IF PURCHASE-ORDER-IN NOT EQUAL TO PURCHASE-ORDER-WS
WRITE REPORT-RECORD FROM DETAIL-LINE
AFTER ADVANCING 2 LINES
ELSE
WRITE REPORT-RECORD FROM DETAIL-LINE
AFTER ADVANCING 1 LINES.
MOVE PURCHASE-ORDER-IN TO PURCHASE-ORDER-WS.

900-CLOSE.
CLOSE ORDER-FILE REPORT-FILE.
APPENDIX D

THE SECOND EXPERIMENTAL PROGRAM (PROGRAM 2)
DATA DIVISION.
FILE SECTION.
FD EMPLOYEE-FILE
LABEL RECORDS ARE OMITTED.
 01 EMPLOYEE-RECORD.
    05 EMPLOYEE-NUMBER-IN PIC X(04).
    05 SOC-SEC-NUM-IN PIC X(09).
    05 EMPLOYEE-NAME-IN PIC X(20).
    05 DEPARTMENT-IN PIC 9(03).
    05 PAY-RATE-IN PIC 9(05)V99.
    05 PAY-TYPE-IN PIC X(01).
    05 PAY-FREQUENCY-IN PIC X(01).
    05 EMPLOYMENT-DATE-IN.
       10 MM-EMPLOYED-IN PIC 9(02).
       10 DD-EMPLOYED-IN PIC 9(02).
       10 YY-EMPLOYED-IN PIC 9(02).
    05 HEALTH-IN PIC X(04).
    05 RETIREMENT-IN PIC X(03).
    05 SAVINGS-PLAN-IN PIC X(03).
    05 SAVINGS-AMOUNT-IN PIC 9(05).
    05 FILLER PIC X(14).

FD REPORT-FILE
LABEL RECORDS ARE OMITTED.
 01 REPORT-RECORD.
    05 FILLER PIC X(03).
    05 REPORT-LINE PIC X(130).

WORKING-STORAGE SECTION.

 01 WORKING-VARIABLES.
    05 FILE-STATUS PIC X(04) VALUE 'NO'.
    05 DEPARTMENT-PREVIOUS PIC 9(03) VALUE ZERO.
    05 EMPLOYEE-NUMBER-PREVIOUS PIC 9(04) VALUE ZERO.
    05 EMPLOYEE-NUMBER-WS PIC 9(04) VALUE ZERO.
    05 ERROR-CHECK PIC X(01) VALUE 'N'.
    05 CURRENT-DATE-CHECK.
       10 MM-CURRENT PIC 9(02) VALUE ZERO.
       10 FILLER PIC X(01).
       10 DD-CURRENT PIC 9(02) VALUE ZERO.
       10 FILLER PIC X(01).
       10 YY-CURRENT PIC 9(02) VALUE ZERO.

 01 HEADING-1.
    05 FILLER PIC X(20) VALUE SPACES.
    05 FILLER PIC X(24) VALUE 'Employee Records for The'.

 01 HEADING-2.
    05 FILLER PIC X(20) VALUE SPACES.
    05 DEPARTMENT-OUT PIC X(14) VALUE SPACES.
    05 FILLER PIC X(10) VALUE 'Department'.

 01 ASTERISK-LINE.
    05 EMPLOYEE-NUMBER-AST PIC X(04) VALUE SPACES.
    05 SOC-SEC-NUM-AST PIC X(09) VALUE SPACES.
    05 EMPLOYEE-NAME-AST PIC X(20) VALUE SPACES.
    05 DEPARTMENT-AST PIC X(03) VALUE SPACES.
    05 PAY-RATE-AST PIC X(07) VALUE SPACES.
    05 PAY-TYPE-AST PIC X(01) VALUE SPACES.
    05 PAY-FREQUENCY-AST PIC X(01) VALUE SPACES.
    05 EMPLOYMENT-DATE-AST.
       10 MM-EMPLOYED-AST PIC X(02) VALUE SPACES.
       10 DD-EMPLOYED-AST PIC X(02) VALUE SPACES.
       10 YY-EMPLOYED-AST PIC X(02) VALUE SPACES.
    05 HEALTH-AST PIC X(04) VALUE SPACES.
    05 RETIREMENT-AST PIC X(03) VALUE SPACES.
    05 SAVINGS-PLAN-AST PIC X(03) VALUE SPACES.
    05 SAVINGS-AMOUNT-AST PIC X(05) VALUE SPACES.
PROCEDURE DIVISION.

000-MAIN-CONTROL.
  PERFORM 100-INITIALIZATION.
  PERFORM 400-PROCESS
    UNTIL FILE-STATUS = 'DONE'.
  PERFORM 700-CLOSE.
  STOP RUN.

100-INITIALIZATION.
  OPEN INPUT EMPLOYEE-FILE
  OUTPUT REPORT-FILE.
  READ EMPLOYEE-FILE
  AT END MOVE 'DONE' TO FILE-STATUS.

400-PROCESS.
  PERFORM 405-DEPARTMENT-CHECK.
  PERFORM 415-SOC-SEC-NUM-CHECK.
  PERFORM 420-EMPLOYEE-NAME-CHECK.
  PERFORM 425-PAY-TYPE-CHECK.
  PERFORM 430-PAY-FREQUENCY-CHECK.
  PERFORM 433-PAY-RATE-CHECK.
  PERFORM 440-EMPLOYMENT-DATE-CHECK.
  PERFORM 445-HEALTH-CHECK.
  PERFORM 450-RETIREMENT-CHECK.
  PERFORM 455-SAVINGS-CHECK.
  PERFORM 460-ERROR-LINE.
  READ EMPLOYEE-FILE
  AT END MOVE 'DONE' TO FILE-STATUS.

405-DEPARTMENT-CHECK.
  IF DEPARTMENT-IN NOT EQUAL TO 100 AND 200 AND 300
  AND 400 AND 500
    MOVE ALL ' »' TO DEPARTMENT-AST
    MOVE 'Y' TO ERROR-CHECK.
  IF DEPARTMENT-IN EQUAL TO DEPARTMENT-PREVIOUS
    PERFORM 410-EMPLOYEE-NUMBER-CHECK.
  IF DEPARTMENT-IN GREATER THAN DEPARTMENT-PREVIOUS
    AND (DEPARTMENT-IN EQUAL TO 100 OR 200 OR 300
    OR 400 OR 500)
    PERFORM 300-HEADIN0
    MOVE DEPARTMENT-IN TO DEPARTMENT-PREVIOUS
    MOVE ZERO TO EMPLOYEE-NUMBER-PREVIOUS.
  IF DEPARTMENT-IN LESS THAN DEPARTMENT-PREVIOUS
    MOVE ALL ' »' TO DEPARTMENT-AST
    MOVE 'Y' TO ERROR-CHECK.

410-EMPLOYEE-NUMBER-CHECK.
  IF EMPLOYEE-NUMBER-IN NOT NUMERIC
    MOVE ALL ' »' TO EMPLOYEE-NUMBER-AST
    MOVE 'Y' TO ERROR-CHECK
  ELSE
    MOVE EMPLOYEE-NUMBER-IN TO EMPLOYEE-NUMBER-WS.
    IF EMPLOYEE-NUMBER-WS GREATER THAN EMPLOYEE-NUMBER-PREVIOUS
      MOVE EMPLOYEE-NUMBER-WS TO EMPLOYEE-NUMBER-PREVIOUS
    ELSE
      MOVE ALL ' »' TO EMPLOYEE-NUMBER-AST
      MOVE 'Y' TO ERROR-CHECK.

415-SOC-SEC-NUM-CHECK.
  IF SOC-SEC-NUM-IN NOT NUMERIC
    MOVE ALL ' »' TO SOC-SEC-NUM-AST
    MOVE 'Y' TO ERROR-CHECK.

420-EMPLOYEE-NAME-CHECK.
  IF EMPLOYEE-NAME-IN EQUAL TO SPACES
    MOVE ALL ' »' TO EMPLOYEE-NAME-AST
    MOVE 'Y' TO ERROR-CHECK.

425-PAY-TYPE-CHECK.
  IF PAY-TYPE-IN NOT EQUAL TO 'H' AND 'S'
    MOVE ALL ' »' TO PAY-TYPE-AST
    MOVE 'Y' TO ERROR-CHECK.
430-PAY-FREQUENCY-CHECK.
   IF PAY-FREQUENCY-IN NOT EQUAL TO 'W' AND 'B' AND 'S' AND 'M'
      MOVE ALL '*' TO PAY-FREQUENCY-AST
      MOVE 'Y' TO ERROR-CHECK.

435-PAY-RATE-CHECK.
   IF PAY-TYPE-IN = 'H' AND PAY-RATE-IN NUMERIC
      PERFORM 437-HOURLY-RATE-CHECK.
   IF PAY-TYPE-IN = 'S' AND PAY-RATE-IN NUMERIC
      PERFORM 439-SALARY-RATE-CHECK.
   IF PAY-RATE-IN NOT NUMERIC
      MOVE ALL '*' TO PAY-RATE-AST
      MOVE 'Y' TO ERROR-CHECK.

437-HOURLY-RATE-CHECK.
   IF PAY-RATE-IN LESS THAN 5.00 OR GREATER THAN 200.00
      MOVE ALL '*' TO PAY-RATE-AST
      MOVE 'Y' TO ERROR-CHECK.

439-SALARY-RATE-CHECK.
   IF PAY-FREQUENCY-IN EQUAL TO 'W AND
      (PAY-RATE-IN < 200.00 OR > 800.00)
      MOVE ALL '*' TO PAY-RATE-AST
      MOVE 'Y' TO ERROR-CHECK.
   IF PAY-FREQUENCY-IN EQUAL TO 'B' AND
      (PAY-RATE-IN < 400.00 OR > 1600.00)
      MOVE ALL '*' TO PAY-RATE-AST
      MOVE 'Y' TO ERROR-CHECK.
   IF PAY-FREQUENCY-IN EQUAL TO 'S' AND
      (PAY-RATE-IN < 500.00 OR > 2000.00)
      MOVE ALL '*' TO PAY-RATE-AST
      MOVE 'Y' TO ERROR-CHECK.
   IF PAY-FREQUENCY-IN EQUAL TO 'M' AND
      (PAY-RATE-IN < 1000.00 OR > 4000.00)
      MOVE ALL '*' TO PAY-RATE-AST
      MOVE 'Y' TO ERROR-CHECK.

440-EMPLOYMENT-DATE-CHECK.
   MOVE CURRENT-DATE TO CURRENT-DATE-CHECK.
   IF MM-EMPLOYED-IN LESS THAN 1 OR GREATER THAN 12
      MOVE ALL '*' TO MM-EMPLOYED-AST
      MOVE 'Y' TO ERROR-CHECK.
   IF DD-EMPLOYED-IN LESS THAN 1 OR GREATER THAN 31
      MOVE ALL '*' TO DD-EMPLOYED-AST
      MOVE 'Y' TO ERROR-CHECK.
   IF YY-EMPLOYED-IN NOT EQUAL TO YY-CURRENT
      MOVE ALL '*' TO YY-EMPLOYED-AST
      MOVE 'Y' TO ERROR-CHECK.

445-HEALTH-CHECK.
   IF HEALTH-IN NOT EQUAL TO 'NONE' AND 'HIGH' AND 'LOW'
      MOVE ALL '*' TO HEALTH-AST
      MOVE 'Y' TO ERROR-CHECK.

450-RETIREMENT-CHECK.
   IF RETIREMENT-IN NOT EQUAL TO 'YES' AND 'NO'
      MOVE ALL '*' TO RETIREMENT-AST
      MOVE 'Y' TO ERROR-CHECK.

455-SAVINGS-CHECK.
   IF SAVINGS-PLAN-IN EQUAL TO 'YES'
      AND SAVINGS-AMOUNT-IN NOT NUMERIC
      MOVE ALL '*' TO SAVINGS-AMOUNT-AST
      MOVE 'Y' TO ERROR-CHECK.
   IF SAVINGS-PLAN-IN EQUAL TO 'NO'
      AND SAVINGS-AMOUNT-IN NOT EQUAL TO SPACES
      MOVE ALL '*' TO SAVINGS-AMOUNT-AST
      MOVE 'Y' TO ERROR-CHECK.
   IF SAVINGS-PLAN-IN NOT EQUAL TO 'YES' AND 'NO'
      MOVE ALL '*' TO SAVINGS-PLAN-AST
      MOVE 'Y' TO ERROR-CHECK.
460-ERROR-LINE.
    IF ERROR-CHECK EQUAL TO 'Y'
        MOVE EMPLOYEE-RECORD TO REPORT-LINE
        WRITE REPORT-RECORD FROM EMPLOYEE-RECORD
        AFTER ADVANCING 2 LINES
        MOVE ASTERISK-LINE TO REPORT-LINE
        WRITE REPORT-RECORD FROM ASTERISK-LINE
        AFTER ADVANCING 1 LINES.
        MOVE SPACES TO ASTERISK-LINE.
        MOVE 'N' TO ERROR-CHECK.

500-HEADING.
    IF DEPARTMENT-IN = 100
        MOVE ' ACCOUNTING' TO DEPARTMENT-OUT.
    IF DEPARTMENT-IN = 200
        MOVE ' SALES' TO DEPARTMENT-OUT.
    IF DEPARTMENT-IN = 300
        MOVE ' MANUFACTURING' TO DEPARTMENT-OUT.
    IF DEPARTMENT-IN = 400
        MOVE ' PURCHASING' TO DEPARTMENT-OUT.
    IF DEPARTMENT-IN = 500
        MOVE ' RECEIVING' TO DEPARTMENT-OUT.
    WRITE REPORT-RECORD FROM HEADING-1
        AFTER ADVANCING TOP-OF-PAGE.
    WRITE REPORT-RECORD FROM HEADING-2
        AFTER ADVANCING 2 LINES.
    MOVE SPACES TO DEPARTMENT-OUT.

700-CLOSE.
    CLOSE EMPLOYEE-FILE
    REPORT-FILE.
APPENDIX E

PROGRAM COMPREHENSION EXERCISE BOOKLET
INSTRUCTIONS

This experiment consists of two sections. In each section, you will be asked to do a comprehension exercise in a fixed amount of time. For each exercise, you will be given a COBOL program to study for a short time using any method you wish. While studying the program, emphasis should be on understanding the program and its functions rather than just memorizing its syntax. Then you will be asked to use the enclosed response sheets to reconstruct a functionally equivalent COBOL program which you think it functions as the same as the original one. In doing so, you should reconstruct only the PROCEDURE DIVISION of the original program.

The following example explains the type of response desired in each reconstruction task:

EXAMPLE:

1- The original program segment:

   WORKING-STORAGE.
   01 I PIC 99 VALUE ZERO.

   PROCEDURE DIVISION.
   PERFORM PAR VARYING I FROM 1 BY 2
   UNTIL I > 10.

   PAR.
   DISPLAY A(I).

   The above program segment outputs odd array elements A(1) through A(9). The following is one way to reconstruct a program segment which produces the same output as the original one:

2- A reconstructed program segment equivalent to the original one.

   PROCEDURE DIVISION.
   MOVE 1 TO I.
   PERFORM PAR UNTIL I > 10.

   PAR.
   DISPLAY A(I).
   ADD 2 TO I.

Note that in this example only the PROCEDURE DIVISION is reconstructed.
SECTION 1

PROGRAM 1 COMPREHENSION

PART 1: STUDYING PROGRAM 1. (TIME: 6 minutes)

When instructed to do so, study the program you have been given until time is called. Then turn over the program and go on to PART 2 when instructed to do so.

PART 2: RECONSTRUCTING PROGRAM 1. (TIME: 12 minutes)

Use "RESPONSE SHEET 1" in the next pages to reconstruct the program you just read until time is called. Remember that you should reconstruct only the PROCEDURE DIVISION.
END OF SECTION 1

GO ON TO THE NEXT PAGE ONLY WHEN INSTRUCTED TO DO SO
SECTION 2
PROGRAM 2 COMPREHENSION

PART 1: STUDYING PROGRAM 2. (TIME: 12 minutes)

When instructed to do so, study the program you have been given until time is called. Turn over the program and go on to PART 2 when instructed to do so.

PART 2: RECONSTRUCTING PROGRAM 2. (TIME: 25 minutes)

Use "RESPONSE SHEET 2" in the next pages to reconstruct the program you just read until time is called. Remember that you should reconstruct only the PROCEDURE DIVISION.
END OF PROGRAM COMPREHENSION EXERCISE
APPENDIX F

PROGRAM MODIFICATION EXERCISE BOOKLET
INSTRUCTIONS

This exercise consists of two sections. In each section, you will be given a COBOL program and will be asked to make specific modifications to the program in a fixed amount of time. You should first study the program and understand its functions. Then use the minimal number of added and/or changed COBOL statements to modify the program to satisfy given requirements. You should rewrite only the portion(s) of the program which are modified by adding new statements and/or changing original statements with their corresponding sections names using the enclosed response sheets. Be sure to write the modified statements proceeded by the letter "M" and the added statements proceeded by the letter "A".

The following example explains the type of response desired in each modification task.

EXAMPLE:

1- The original program segment:

```
WORKING-STORAGE.
  01 I PIC 99 VALUE ZERO.

PROCEDURE DIVISION.
  PERFORM PAR VARYING I FROM 1 BY 2
       UNTIL I > 10.

PAR.
  DISPLAY A(I).
```

The function of the above program segment is to output odd array elements A(1) through A(9). The following is what should be shown in the response sheet to modify the program segment to output the squared odd array elements A(1) through A(9).

2- The modified portions of the original program segment.

```
WORKING-STORAGE.
  01 SQUARED-VAL PIC 9999 VALUE ZEROS.

PROCEDURE DIVISION.

PAR.
  COMPUTE SQUARED-VAL = A(I) ** 2.
  M DISPLAY SQUARED-VAL.
```

Note that the unmodified portions of the program are not rewritten on the response sheet. The new added statements are marked with the letter "A" while the modified statements are marked with the letter "M".

GO ON TO THE NEXT PAGE WHEN INSTRUCTED TO DO SO.
SECTION 1

PROGRAM 1 MODIFICATION

TIME: 18 minutes

When instructed to do so modify the program you have been given to satisfy the following requirements:

1. Let the program check that the DISCOUNT amount is zero when the purchased QUANTITY is less than 2000 units.

2. Let the program calculate and print the COST, TAX, and DISCOUNT subtotals for each vendor.

3. Let the program calculate and print the Cost, TAX, and DISCOUNT grandtotals.

Use "RESPONSE SHEET 1" in the next pages to show your modifications.
RESPONSE SHEET 1
PROGRAM 1 MODIFICATIONS

PAGE 1
SECTION 2

PROGRAM 2 MODIFICATION

TIME: 28 minutes

When instructed to do so modify the program you have been given to satisfy the following requirements:

1. Let the program check the matching between EMPLOYEE NUMBER and DEPARTMENT NUMBER using the following classification.

<table>
<thead>
<tr>
<th>Department number</th>
<th>Valid employee number</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>1 to 1999</td>
</tr>
<tr>
<td>200</td>
<td>2000 to 3999</td>
</tr>
<tr>
<td>300</td>
<td>4000 to 5999</td>
</tr>
<tr>
<td>400</td>
<td>6000 to 7999</td>
</tr>
<tr>
<td>500</td>
<td>8000 to 9999</td>
</tr>
</tbody>
</table>

2. Let the program calculate and print the SAVING amounts subtotals for each DEPARTMENT as well as SAVING amount grandtotal.

3. Let the program determine and print the highest and lowest PAY RATE for every PAY TYPE.

Use "RESPONSE SHEET 2" in the next pages to show your modifications.
RESPONSE SHEET 2
PROGRAM 2 MODIFICATIONS

PAGE 1
END OF PROGRAM MODIFICATION EXERCISE
APPENDIX G

RESIDUAL PLOTS OF COGNITIVE COMPLEXITY--DIFFERENTIATION AND INTEGRATION VERSUS PROGRAM COMPREHENSION AND MODIFICATION
COGNITIVE DIFFERENTIATION vs. PROGRAM I COMPREHENSION
COGNITIVE DIFFERENTIATION vs. PROGRAM II COMPREHENSION
COGNITIVE INTEGRATION vs. PROGRAM I COMPREHENSION
COGNITIVE INTEGRATION vs. PROGRAM II COMPREHENSION
COGNITIVE DIFFERENTIATION vs. PROGRAM I MODIFICATION
COGNITIVE DIFFERENTIATION vs. PROGRAM II MODIFICATION
COGNITIVE INTEGRATION vs. PROGRAM MODIFICATION
COGNITIVE INTEGRATION vs. PROGRAM II MODIFICATION
BIBLIOGRAPHY

Books


Articles


Curtis, B.; S. B. Sheppard; P. M. Milliman; M. A. Borst; and T. Love, "Measuring the Psychological Complexity of Software Maintenance Tasks with Halstead and McCabe Metrics," IEEE Transactions on Software Engineering, SE-5, No. 2 (March 1979), 96-104.


Miller, G. A., "The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information," The Psychological Review, 63, No. 2 (1956), 81-97.


Unpublished materials


Landfield, A. W. and M. A. Barr, "Ordination: A New Measure of Concept Organization," Unpublished Manuscript, University of Nebraska (1976), 1-33.


