INFORMATION TO USERS

The most advanced technology has been used to photograph and reproduce this manuscript from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

U·M·I

University Microfilms International
A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
313/761-4700 800/521-0600

Order Number 9108771

The impact of computer-aided software engineering on programmer productivity and system quality

Granger, Mary J., Ph.D.

University of Cincinnati, 1990

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

THE IMPACT OF COMPUTER-AIDED SOFTWARE ENGINEERING

ON

PROGRAMMER PRODUCTIVITY AND SYSTEM QUALITY

A Dissertation submitted to the

Division of Graduate Studies and Research
of the University of Cincinnati

in partial fulfillment of the
requirements for the degree of

DOCTOR OF PHILOSOPHY

in the Department of
Quantitative Analysis/Information Systems
of the College of Business Administration

1990

by

Mary J. Granger

M.B.A., University of Cincinnati, 1980

Committee Chair: Dr. Roger Alan Pick

UNIVERSITY OF CINCINNATI

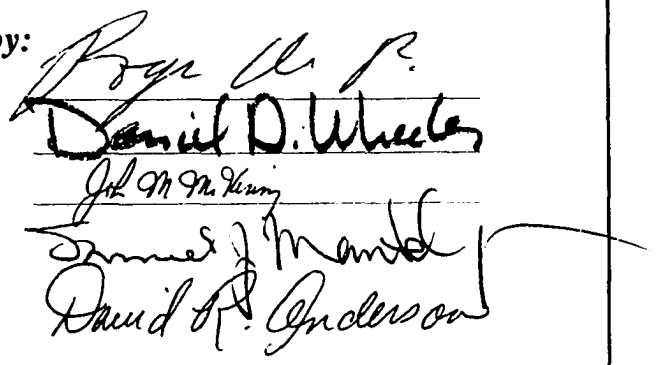
July 19 19 90

I hereby recommend that the thesis prepared under my supervision by Mary J. Granger

entitled "The Impact of Computer-Aided Software Engineering on Programmer Productivity and System Quality"

be accepted as fulfilling this part of the requirements for the degree of Doctor of Philosophy

Approved by:



THE IMPACT OF COMPUTER-AIDED SOFTWARE ENGINEERING

ON

PROGRAMMER PRODUCTIVITY AND SOFTWARE QUALITY

Mary J. Granger

University of Cincinnati, 1990

The high cost of software production has forced organizations to look for new environments to reduce the time required for the development process. Currently, one highly touted route toward improved programmer productivity and increased system quality is Computer-Aided Software Engineering (CASE): the computer-based automation of system development tasks.

Software developers have automated almost every functional area and level of the organization except their own. CASE is an attempt to automate the production of software; computers are being used to enhance the requirements and analysis, design, coding, implementation and maintenance phases of the software development life cycle.

This research addresses two basic questions: when CASE technologies are used to develop software 1) Is programmer productivity improved? 2) Is system quality increased?

An experimental study with a control group (non-CASE) and a treatment group (CASE) was designed to investigate the effects of CASE usage. The same task, Pascal pretty printer, was developed by both groups. Both the developmental processes and the final projects were studied.

This research makes three contributions to the study of software development. First, to our knowledge, it is the first controlled experiment investigating CASE tools. Second, several metrics were identified that can be used to identify and evaluate programmer productivity.

The third contribution of this research is the quantitative measures to the claims of increased programmer productivity and system quality being made by CASE vendors and others. In this study, programmer productivity increased when CASE technologies were used to design a software system. Also in this study, the quality

of the systems improved; more complete systems were developed by the teams that used CASE technologies for system design.

Information technology managers should be encouraged in their quest for increased programmer productivity. A major component of the software crisis is the inability to measure, estimate, and improve programmer productivity. This study indicates that use of CASE tools would improve programmer productivity.

ACKNOWLEDGMENTS

I thank the members of my committee: Dr. David R. Anderson, Dr. John M. McKinney, Dr. Samuel Mantel and Dr. Daniel D. Wheeler for their valuable comments and suggestions.

A very special thanks to my chairperson, Dr. Roger Alan Pick, for all his comments, suggestions, time, effort, interest and patience with this dissertation.

I thank Roger Stuebing for his invaluable help with the statistical analysis. Additional thanks to all my students, especially Roger Anderson, Robert Alfieri, William Glenn Campbell, James Horrell, Timothy Adams, Rahul Bawa and Scott Drew. I appreciate the moral support from fellow graduate students; Rob Rokey, Zaman Matharsha, Fred Besco, Adam Fadlalla, Jiang Jiunn-Yih (J.J.) and Marge Sklar. I thank Dr. John M. McKinney for the initial encouragement and support to begin the doctoral degree.

Thanks to my children, Mary and James, who are glad the perpetual student is finished (for now), for their moral support and encouragement.

Finally special thanks to my husband, Jim Granger, for his editing, love and encouragement.

TABLE OF CONTENTS

LIST OF APPENDICES

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

BACKGROUND

The high cost of software production has forced organizations to look for new environments to reduce the time required for the development process. Currently, one highly touted route toward improved programmer productivity and increased system quality is Computer-Aided Software Engineering (CASE): the computer-based automation of system development tasks.

Software developers have automated almost every level of the organization except their own. CASE is an attempt to automate the production of software; computers are being used to enhance the requirements and analysis, design, coding, implementation and maintenance phases of the software development life cycle.

Much has been written about the virtues of CASE technologies. Many information systems managers realize

that they need help, but do not know if CASE is a solution to their software development problems. Is CASE worth the time and financial investments necessary? Are there real benefits in terms of programmer productivity and system quality for those software developers who use CASE? Will CASE prove to be a significant advantage for the organization that competes in a rapidly changing global environment?

There is an absence of studies that quantitatively evaluate the influence of CASE technologies on programmer productivity or system quality.

CONTENT

This research addresses two basic questions: when CASE technologies are used to develop software 1) Is programmer productivity improved? 2) Is system quality increased?

An experimental study with a control group (non-CASE) and a treatment group (CASE) was designed to investigate the effects of CASE usage. Both the developmental processes and the final projects were studied.

Programmer productivity was measured using the differences in the amount of time required to code the same system. System quality was measured using the differences in the complexity and the levels of completeness of the final systems. The control group designed the system without CASE technologies and the treatment group designed the system with CASE technologies.

This dissertation presents the details of the experiment, the analysis of the data, the results and conclusions of the analysis. Chapter 2 contains an overview of the present literature on CASE, the software crisis, the software development life cycle and structured development techniques (structured requirements and analysis, structured design and structured programming). Chapter 3 presents the details of the experiment, including descriptions of the subjects, the task, the control and the treatment. The fourth chapter states the model that is used for the statistical analysis, including the definitions of the software metrics used to evaluate the final system. Chapter 5 presents the statistical analysis of the data and the results of the analysis. The final chapter summarizes the research, presents the conclusions, suggests ways in which this research can be useful to practitioners and lists future research questions.

CHAPTER 2

LITERATURE REVIEW

Computer-Aided Software Engineering (CASE) technologies automate the software development process. This chapter begins with an introduction of CASE and then presents overviews of the software crisis, the software development life cycle, and software development methodologies. CASE technologies and the goals and objectives of CASE technologies are discussed. Since Excelerator was the CASE tool used in this research, this specific CASE product and its capabilities are reviewed. Finally, there is a review of both the research and practitioner literature on CASE technologies. Halstead's Software Science and related metrics are discussed in Chapter 3.

INTRODUCTION

CASE technologies provide support for one or more software development methodologies. Arthur (1983 pp. 4-5) defines methodology as the "how" of system development and

technology as the "tool kit" used to implement the methodology. Software engineering methodologies (Turner 1984), project management capabilities (Levine 1989), prototyping (Boar 1985) and simulation (Pritsker 1984) are several of the software development methods supported by CASE technologies. Most CASE tools originally evolved from, and continue to support and refine (Messenheimer, 1988), such techniques as structured analysis (DeMarco 1979), structured design methodologies (Yourdon 1979) and data modeling methodologies (Warnier 1981, Chen, P. 1976), which in turn evolved from structured programming (Linger 1979 p.7). Structured techniques apply engineering discipline to system building and make substantial improvements in the design and programming of systems (Martin,J. 1988 p.8). System design is an iterative process that necessitates change and modifications in the documentation used when applying structured techniques. Often the initial design will be suboptimal; but the systems analyst will avoid making design enhancements in order to escape the tedious, manual documentation modifications (Misra, 1988). By automating the structured methods, CASE proposes to be more efficient than manual methods; there is a reduction in the number of iterations during the design phase and necessary revisions are less difficult to implement. Consideration of alternative

designs for the same problem is also more feasible (Necco 1989). Structured methods help analysts master the complexity of problems and are the basis for problem solving with or without the computer. Many authors contend that in order to be effective, CASE must have a foundation in structured methods (Martin, C. 1988a; Wallace 1988; Hausen 1981).

SOFTWARE CRISIS

Information systems support new product development, production, managerial decisions and enable the organization to be more competitive. Investment in information systems software development and maintenance can be a major corporate expense. Productivity improvements of software development have been exceeded by improvements in hardware performance. Because general solutions do not exist, the real problems reside in producing custom-built, application-specific, organization-specific software. There has been an increased demand for larger and more complex application software systems. The term "software crisis" describes the current state of software development: systems that do not meet the client's specifications, systems that are over budget,

systems that are late, systems that are extremely complex and systems that are difficult to maintain. The software crisis involves deficiencies in software quality, programmer productivity, lead time and software development cost. Programmer productivity has only (compared to the rest of the computer industry) been growing at a rate of 5% per year. The last major breakthrough in programmer productivity was in the 1950s with the introduction of language compilers (Frenkel 1985). Shemer (1987) cites the following estimates of the extent of the software crisis:

1. There is an estimated backlog of information systems development of four years, with a hidden backlog (those that are not even requested) of eight years. One to 2.4 million software professionals will be needed in the 1990s compared with 250,000 in 1982 (Martin 1982).
2. The relative cost of the software component of a system is increasing at the same rate that the relative cost of the hardware component is decreasing (90% in the 1950s to 10% in the 1990s) (Schindler 1981; Wasserman 1982).
3. Maintenance costs outweigh development costs two to four times (Lientz 1980; Ramamoorthy 1984).

4. Forty-five percent of maintenance problems are detected after the system is delivered (Martin 1982) and the relative cost of fixing these problems is 50-100 times greater than if the problems were uncovered during the analysis phase (Boehm 1973).

Although recent research puts point 2 in doubt (Frank 1988; Gurbaxani 1987), the fact remains that the crisis is persuasive in the software industry. Systems currently being developed are larger and more complex than previously developed software.

Following the software development life cycle phases and embracing software development methodologies are attempts to alleviate the software crisis. Most recently, advocates of CASE technologies believe that their tools are an even better solution to the software crisis.

SOFTWARE LIFE CYCLE

The software life cycle formally defines the phases in a system development process. It provides a general framework around which to build the system. The life cycle defines a series of top-down development activities for iteratively developing software systems and often

consists of the five basic phases: requirements analysis, design, coding/implementation, testing, and maintenance. The phases are separate, but linked: a phase relies on the previous phase for inputs and in return sends feedback to the previous phase for verification (Smith 1987, p. 21). During the requirements analysis phase the functions and the data requirements necessary to solve the problem are specified and documented. Page-Jones (1988 p. 2) defines design as a "bridge between the analysis of the problem and the implementation to the solution to that problem." The design phase defines and documents 'how' the problem will be solved during the coding/implementation phase. The coding/implementation phase transforms the design into computer code and also involves debugging. Before the software product is delivered, it is tested to insure that it satisfies the specifications established during the requirements analysis phase. Maintenance activities include modifications, enhancements and further removal of errors.

Each phase consists of well-defined, systematic, step-by-step procedures, but one phase need not be completed before the next phase begins. At every phase, there should be a verification of the requirements. The process of building the system requires checking with and feedback

to the previous activities and phases. Therefore, the life cycle should not be viewed as a set of rigidly defined activities, but as a set of guidelines for systems development that can be used to derive procedures appropriate for a particular project. There may be many variations on the five phases listed above.

Lack of user/client involvement until the final stages, output specifications required during the requirements analysis phase, and communication difficulties are some of the problems mentioned during discussions of the system development life cycle (Mahmood 1987). Adherence to the life cycle model can also be time consuming, costly and complex. Other models for systems development include, but are not limited to, prototyping (Boehm 1984a; Necco 1987) and information centers (Necco 1987), but the life cycle model is currently the most widely used (Mahmood 1987) and is often implemented in conjunction with other development methodologies.

SOFTWARE DEVELOPMENT METHODOLOGIES

Structured development methodologies facilitate the system development life cycle. "The methodologies are 80% alike. The major difference is the symbols they use or that they emphasize one part of the software life cycle over another." (Georges quoted by Messenheimer 1988, p. 31) The three methodologies that have impacted systems development and productivity the most are structured programming (Linger 1979 p.7), structured design (Yourdon 1979) and structured analysis (DeMarco 1979).

Structured programming is the writing of a computer program in a standardized manner to decrease the debugging and testing problems, increase documentation and readability, and facilitate maintenance. The emphasis is on writing clear, concise, more readable and less error-prone code. In a narrow sense, structured programming attacks programming complexity and poor programming productivity with three basic programming constructs: sequence, iteration and selection. In order to promote straight-line programming, GOTOs are avoided and one entrance, one exit modules are encouraged. This structure is imposed upon previous ad hoc methods of programming to control the complexity and make the code more

understandable and readable. Because many programmers believe that programming is a creative activity, structured programming was not immediately accepted, but currently it is regarded as a better way to write computer code than more unconstrained methods.

In order to deal with larger and more complex systems, Stevens (1974) and Yourdon (1979) began to formulate the concept of structured design. Yourdon (1979 p. 8) defines structured design as "the art of designing the components of a system and the interrelationship between those components in the best possible way." Structured design looks at the problem at a different level than the programming level (Martin 1988, p. 10). The major graphic design tools for structured design are structure charts and data flow diagrams. Structure charts emphasize the procedural aspects of the system and data flow diagrams concentrate on the flow of data through the system. Structure charts depict the system modules and the interactions between them: a hierarchical order controls the graphic representation of the system. A data flow diagram shows the data that flows between the processes of the system and the way those processes transform the data. A data dictionary defines all the flows, processes, data stores and data sources in the data flow diagram.

Initially, structured analysis defined all the system requirements in narrative form (Yourdon 1989b, p. 123). Currently, structured analysis is a graphical method of interfacing with the user/client. Use of both structure charts and data flow diagrams in the requirements analysis phase help define the system at a higher/corporate level. Both structured analysis and structured design are iterative processes. Manual implementation of these techniques has limited the use and acceptance of structured methods because they are tedious, repetitive and very labor intensive (Chikofsky and Rubenstein 1988); these negative aspects of structured methods often outweigh any improvement gained (Chikofsky 1988; Wallace 1988). Because of this, rework is often avoided and requirements and design documentation become incomplete and inaccurate.

SOFTWARE ENGINEERING

Software Engineering is the application of engineering principles to software development; work on software projects is organized as work on engineering projects is organized. Fairley (1985) defines Software Engineering as the systematic production and maintenance of software

products that are developed and modified on time and within cost estimates. A software product is the data, documentation, computers, procedures or programs and any other pertinent entities required to solve a particular problem. Initially, the formal methodologies of Software Engineering (structured analysis, structured design and structured programming) were applied to scientific or technical software such as compilers and operating systems; currently, they are also being used to develop information systems and business application software (Messenheimer, 1988; Norman, 1989a). Application software also has become too complex to develop without formal methods; improved methods and tools are needed. Complexity has increased because the application systems of today are larger, deal with more difficult tasks and process more data. Software developers are busy automating other aspects of information processing, but often neglect to automate the development of software. CASE (Computer-Aided Software Engineering) is the automation of software development.

CASE TECHNOLOGIES

CASE is not another software engineering methodology. It is a set of tools or an environment that supports software engineering methodologies (Burkhard, 1989; Norman, 1989a). CASE encompasses many products which support the design and development of computer-based information systems. These products include structured analysis and design tools, such as graphics tools for drawing and maintaining structure charts and data flow diagrams, automated data dictionaries, interactive debugging aids, programming support libraries, text editors, automated verification systems, test data generators, code generators, etc. (Boehm, 1981, p. 460; Burkhard, 1989). They support totally or partially the system development life cycle.

CASE (Computer-Aided Software Engineering) is perceived by many software developers as the answer to their software problems and the software crisis that has plagued the computer industry (Chikofsky 1988; Martin C. 1988b; Nejmeh 1988). CASE usage is considered a major step toward total automation of software development (McClure 1989, p. 4).

CASE technologies define a new software development environment to improve the way systems are built. The automation of systems development removes some of the drudgery of planning, analyzing, designing, programming and documenting systems, and provides the power to produce large complex application software systems quickly.

Computer-aided software engineering is defined conceptually as the automation of software development throughout the entire life cycle (McClure 1989 p. 5; Mynatt 1989; Hausen 1981). By combining interactive graphics and databases, current integrated CASE technologies support development of the design and analysis phases as well as the implementation and maintenance phases of the software development process (Ramanathan 1988; Martin, C. 1988b). An initial attempt at computer-aided development, PSL/PSA (Teichroew, 1977), automated the generation of documentation during the requirements and analysis phase. Future CASE technologies will help define corporate information needs, and phases such as corporate strategy planning (Rochester 1989) or organization and information system modeling (Chen, M. 1989) will be added. CASE combines techniques and tools aimed at building and maintaining software systems of all

types - large and small, commercial and scientific, on-line, real-time, and batch.

Stratland (1989) and Voelcher (1988) divide CASE into two levels; upper CASE or front-end technologies which support the analysis and design phases of the life cycle and lower CASE or back-end technologies for automated code generation and project support (Stratland 1989; Voelcher, 1989). Most CASE tools today support either upper or lower CASE but not both (Aranow 1988) and, therefore, are not fully integrated. The second generation of CASE technologies will be integrated and standardized (Yourdon 1988). CASE tools with an expert system component are already on the market and eventually CASE tools may have natural language capabilities. Then, in order for the CASE tool to generate a data flow diagram, a list of data flows and their processes may be all that is necessary (Martin C. 1988b). Eliot(1986) sees the evolving technologies as assisting but not replacing software engineers; Wallace (1988) believes CASE tools will perform the mechanical aspects of analysis and design, freeing software engineers' time for creativity (Wallace 1988).

Objectives/goals of CASE technology

Two major objectives of CASE technology are improving the productivity of programmers during system development and improving the quality of the software system being developed (Chikofsky 1989; McClure 1989 p.6). Bachman (1988) proposes that CASE should also be instrumental in systems' maintenance, the most time consuming information systems task.

Productivity refers either to the amount of output from a specified resource input or the amount of resource input to produce a given output. This simple concept can be difficult to define in an operational sense. Most discussions about CASE technology do not attempt to provide a definition of productivity; there is no consensus among those who do attempt a definition. Often increased productivity implies an increased number of lines of code per time period; but fast coding is not enough to make a productive programmer (Robinson 1988). Humphrey (1988 p.77) claims that "productivity data is generally meaningless unless explicitly defined" and that the number of lines of code per period can vary by "100 times or more." Chikofsky (1989) believes lines of code per period is such a weak indicator of productivity that

he suggests improving typing skills would improve productivity. Productivity can be enhanced by reducing the programming efforts and streamlining development procedures (de la Torre 1988; Boehm, et al. 1984). Jones (1986 pp.7-8) reminds us that higher-level languages use fewer lines of code in order to accomplish the same task; higher-level languages are easier to use and fewer source lines of code per period are written; higher-level languages define collections of machine instructions as a single function or name (Jones 1986, pp. 48-49). Thus productivity, when measured by lines of code per period, would appear to decrease when higher level languages are used. But the number of systems developed per period would increase. Companies that adopt CASE technologies often do not establish the current level of productivity as a benchmark for measuring increased productivity. That is, they do not have data on current productivity against which to measure productivity using CASE. It is also not economically feasible to run both methods in parallel within a controlled commercial environment. "There are lots of quotes for efficiency gains," said one CASE vendor, "but few that are meaningful." (Voelcker 1988, p. 27). There is an absence of measurable data that can be used to evaluate the productivity increases of CASE.

The second major objective of CASE technologies is improved system quality. Quality implies a dependable system and includes checks on completeness, consistency, accuracy and redundancy (Chikofsky 1989; McClure 1989; de la Torre 1988). The system should meet the users' needs. Card (1988, p. 82) suggests a two-part definition of software quality: satisfying product requirements effectively and system efficiency. Poor quality means "errors and discrepancies with the requirements" and efficiency implies a method of system production that "minimizes development costs and rework while maximizing maintainability." CASE usage should foster better project control and enforcement of standards (Stratland 1989), creating a system with increased quality. Inconsistencies and omissions should be detected earlier in the life cycle so that requests for change can be incorporated with less effort (Misra 1988).

Software quality is difficult to quantify. Software Science (Halstead 1977) was an initial attempt to measure software quality. Halstead proposed measures of the complexity levels of both the algorithm needed to solve the problem and the computer program written to implement the algorithm. Additional research on software metrics (Gordon 1979a; Elshoff 1984) built on Halstead's metrics.

There is, however, an absence of measurable data that can be used and is used to evaluate the improvements in the quality of systems developed with CASE. This study attempts to generate such data.

EXCELERATOR

Excelerator, a product of the Index Technology Corporation, was the first IBM PC based CASE product and currently is the most widely used microcomputer CASE tool. Mirsa (1988) compared Excelerator with two other CASE products (Structured Architect Version 1.2 and Design/I Version 3.50) and found that it had the most integrated environment. It is an upper CASE tool, supporting the analysis and design phases of the software development life cycle and built around an integrated data dictionary. All the information about the system is maintained in one location and can be used by a number of people working on developing the system.

Excelerator provides full capabilities for creating and maintaining the following types of graphs (Whitten 1987):

data flow diagram (Yourdon and Gane/Sarson symbol sets)
structure charts (Yourdon/Constantine)
structure diagrams (Michael Jackson)

entity-relationship data model (both Chen and MEIRSE sets)
data model diagrams (Bachman techniques)
presentation graphics (a superset of ANSI system)

The structured methodologies supported by Excelerator are listed after each type of graph. These graphs can be exploded or expanded into different information levels and the relationships among the elements in the graphs are maintained and cross-referenced in the data dictionary. There is extensive verification checking against the structured methodology rules (Topper 1990; Mirsa 1988).

RECENT STUDIES

Very little research about the effectiveness of CASE technology exists (Carey 1988, Norman 1989a). Although recent studies (Acly 1988; de la Torre 1988) report increased productivity and improved system quality, few formal measurements exist. There is reason to suspect that reports appearing in the popular literature may be biased. Due to large financial and resource investments in CASE technologies by their organizations, many of those claiming success would be in uncomfortable positions if they did not have favorable results. Other affirmative reports (Martin 1988a; Chikofsky 1989) have come from the

developers of commercial products. Often, reported productivity and quality increases are managers' perceptions rather than based on objective data. There is a need for an unbiased examination of CASE technologies, and this study attempts such an evaluation.

The following studies from recent literature are from two major sources: the research community and the practitioner community. Studies written by and for the research community are very systematic; those written for the practitioner community are anecdotal and informal. The only published empirical study in the research community is one done by Ronald J. Norman and Jay F. Nunamaker, Jr. and reported in the Communications of the ACM (September 1989), the Proceedings of the Ninth International Conference on Information Systems and the Proceedings of the Twenty-second Annual Hawaii International Conference on Systems Sciences (1989). Loh and Nelson published their empirical study in Datamation (July 1989). Reports of success in implementing CASE technologies within specific corporations are chronicled in practitioner journals; Carma McClure in Byte (April 1989) and Jack B. Rochester in IS Analyzer (October 1989).

Norman and Nunamaker (1988, 1989a. 1989b)

Norman and Nunamaker conducted a survey of Information Systems analysts. Findings are reported in Norman and Nunamaker (1988, 1989a. 1989b)

Ninety-nine users of the Excelerator CASE tool volunteered to participate in the survey. The subjects were from a representative cross-section (over a dozen standard industry codes) of 47 various size companies in the United States and Canada and had 56 different titles. Ninety-one respondents finished the questionnaire. Seventy-nine percent of the respondents reported that they had been working with Excelerator 18 months or less. Each company that participated was sent at least one diskette that contained the questions for the survey and software to administer it. Included with the questions on CASE were questions on demographic information and on-line 'help' with software definitions. After completion of the survey the diskettes were mailed back to the researchers. The respondents answered 136 paired comparison questions consisting of 15 different CASE functions and 2 additional behavioral functions.

The behavioral functions were:

Project member's communication via [CASE product]
Project standardization

The CASE functions were:

Analysis - Graph Analysis
Analysis - Entity List
Analysis - Report Writer
[CASE product] works on both PC and mainframe
Data Dictionary
Data Flow Diagrams (Gane & Sarson, Yourdon)
Entity/relationship data model (Chen or MERISE)
Import and/or Export Facility
LAN support
Logical Data Model diagram (IBM)
Presentation Graphics
Record Layout Generation
Screen/Report Design
Structure Charts (Constantine)
Structure Diagrams (Jackson)

The two questions asked for each pair of functions are "Of the following items which one most increases your productivity over manual methods:" and "Rate how similar these items are in their effect on your productivity (1-7, 7 - very different)? Enter 1 to 7?" (Norman 1989b). The data from the survey was evaluated from several different aspects.

In both (1989a) and (1988), Norman addresses the question of users' perceptions of CASE tool functions' productivity compared to manual methods' productivity. He used multidimensional scaling to yield a dominance ranking for the CASE functions and a cluster analysis for the behavioral functions. The main focus of these two articles (1988 and 1989a) was the perceived dominant

productivity functions of CASE tools. The main focus of the article (1989b) was on the behavioral functions associated with CASE tools.

Norman and Nunamaker (1989b) use the data from the survey to pose three research questions:

> 1. Can I.S. professionals prioritize the component parts of CASE products that contribute the most to increasing their productivity over manual methods, such that this prioritizing is not just a random event?
> 2. Is there any agreement among I.S. professionals regarding a prioritizing of the component parts of CASE technologies?
> 3. Does CASE technology provide greater technological improvement or behavioral improvement compared to manual methods?

(Questions 1 and 2 are also discussed in the first two articles.) In the third article they established the hypotheses for the three questions and gave additional detail about the statistical methods used.

Question 1 was analyzed with individual responses using Kendall's coefficient of consistence which is designed to determine the consistency of an individual's responses to paired comparisons. The coefficient of consistence was converted to a chi-square value; a significant chi-square value indicated that the responses cannot be attributed to chance. Respondents were able to prioritize the CASE functions that lead to perceived increased productivity.

Question 2 was analyzed using Kendall's coefficient of agreement, which was employed to evaluate the amount, if any, of agreement among the respondents. The coefficient of agreement was converted to a chi-square value; a significant chi-square value indicated that there was agreement among the respondents. There is agreement regarding prioritization of CASE product component parts with respect to perceived improved productivity.

The third research question compared two behavioral functions to the perceived technological improvements of CASE usage. The first behavioral function was adherence to organizational standards and the second was increased communications. The two behavioral functions were used to determine whether or not either one has an impact on productivity in comparison to technological CASE functions. Each behavioral function was compared to all of the other 16 components collectively. Then, each behavioral function was evaluated against each of the other functions on an individual basis. Neither of the null hypotheses were rejected for each CASE functions.

Norman and Nunamaker conclude (research questions 1 and 2) that there is consistency within the individuals'

responses to the paired comparisons and also there is agreement among the respondents about perceived productivity of the CASE functions and the behavioral functions.

The overall conclusion (research question 3) was that the respondents perceived an increase in technical productivity rather than in the behavioral functions. The respondents perceived an adherence to standards to contribute more to productivity than other CASE functions. Communication was perceived to contribute less to productivity than other CASE functions. The one-to-one comparisons produced mixed results: some of the CASE functions were perceived to contribute more to productivity than the adherence to standards and some were contributing more to the adherence of standards. The results were the same for the communication function, but fewer functions were considered stronger and more considered weaker. For both behavioral functions, there were some CASE functions that showed significant differences.

Standards adherence was stronger than:

1. Structure Diagrams
2. Record Layout Generation
3. Analysis -> Entity List
4. Lan support
5. Import and/or Export facility
6. Communication
7. CASE product works on both the PC and mainframe

Standards adherence was weaker than:

1. Data Flow Diagrams
2. Data Dictionary

Communication was stronger than:

1. LAN support for the CASE product
2. CASE product works on both the PC and mainframe

Communication was weaker than:

1. Data Flow Diagram
2. Entity/Relationship data model
3. Presentation Graphics
4. Data Dictionary
5. Screen/Report Design
6. Project standardization
7. Analysis -> Report Writer

Norman and Nunamaker appear to have a good cross section of different organizations and I.S. professionals. An unanswered question is whether the responses to the questions identified with the respondents or companies after the data is collected. There is no mention of confidentiality in any of the three studies. It is also possible that the respondents would report a productivity increase with CASE tools because they might have been instrumental in obtaining the funds or authorization for

their acquisition. I.S. personnel would look relatively foolish if they just spent $10,000 (approximate cost of Excelerator) to buy software that did not increase their productivity. A major contribution of the study is a ranking (by I.S. professionals using Excelerator) of CASE tool component parts that increase productivity over manual methods; there was agreement on those component parts that are perceived to contribute to increased productivity. These professionals also perceive that they receive more technological rather than behavioral improvement when CASE tools are used. There are no figures, nor did Norman try to collect them, for the amount of productivity that is gained by using CASE tools instead of manual implementation.

Loh and Nelson (1989)

Loh and Nelson (1989) conducted a survey of 40 programmers, analysts and systems designers from 12 different organizations. Twenty-six different CASE tools were used by the respondents, with most of the organizations having more than one CASE tool available. There is no information about the types of companies, the locations of the companies, nor the departments where the CASE tools were being used. This lack of detail raises

questions about the representation of different types of organizations and the type of software being written (application software versus system software). Additionally, there is no mention of the form of survey nor how it was administered. Based on the ratings reported, the assumption is made that there were some scaling measures.

The major finding of the research was that productivity gains vary depending upon the CASE tool and programmer acceptance. Other findings include:

CASE usage often requires changing methodologies.

Training is not trivial.

Emphasis is shifted from the coding phase (lower end) of the life cycle to the requirements and design phases (front end).

CASE effectiveness is selective and is affected by:

programmer proficiency

system size

integration of CASE tools

One of the more interesting findings is that CASE tools are more effective on small, simple projects than on large (longer than 2 years), complex projects. This finding was mentioned under the section discussing CASE failures and

was attributed to a lack of integration of CASE tools and a lack of data sharing.

All the respondents used CASE for the requirements and analysis phases of the system development life cycle, but only 50% used CASE for the coding and maintenance phases. Twenty-five percent of the respondents said they used CASE throughout the entire life cycle.

Some of the reasons that CASE tools were not used or not adopted include:

- difficulty in sharing data
- lack of integrated/compatible tools
- a long learning curve
- lack of user involvement in tool selection
- lack of management support
- poor training

This research claims that there are productivity gains with CASE usage. This statement is based on the results of the survey, however, there are no figures to support this claim. The respondents report that they are more productive, but do not report any measurements or benchmarks of productivity. They also claim that when CASE is used, more time is spent in the requirements and

analysis phase and less in the coding; the time spent during the front end of the life cycle has risen from 44% to 55%. It would have been useful if Loh and Nelson had specified where and how these measures were derived. There are no qualitative measures to support the productivity claims.

McClure (1989)

McClure (1989) briefly chronicled the experiences of three companies, Touche-Ross, Deere & Co. and DuPont as examples of successful implementation of CASE technology. Each example was a concise description of the type of CASE tool the company used and the type of application that was being implemented with the tool.

Touche-Ross adopted a variety of CASE tools, including Excelerator, Information Engineering Facility (IEF), POSE, Visible Analyst and DesignAid, for creating the requirements specification for custom-built information systems. Managers at Touche-Ross are strong believers in learning the methodology before the CASE tool could be used effectively. They believed that they produced high-quality software that is easier to maintain, with CASE

tools that assisted in systems planning, requirements analysis, system design and code generation.

Managers at Deere & Co. believed that automation of the software process lead to both increased productivity and quality and reduced maintenance. They were using Information Engineering Workbench, IEW and APS, which generates COBOL code. They initially began using CASE tools as a way to manage their data and felt that it is better to have the data stored, in addition to being in their analysts' heads, in a repository. Deere & Co. believed that their productivity has increased, but they did not have any pre-CASE figures to use for comparison.

DuPont reported productivity gains that range from 3 to 1 to 6 to 1, with savings reaching $2 million and maintenance costs sometimes decreasing by 75 percent. They used CASE tools for custom-built software for both internal and external clients and felt that if the end-user is not involved, the project is a good candidate for failure. CASE tools enabled DuPont to involve the end user earlier in the systems development cycle. DuPont had their own CASE tool, RIPP, which is a prototyping approach to systems development.

All three of the above companies reported 'productivity' gains and savings during development and maintenance. DuPont did give some dollar figures or percentages of saving, but no indication where or how these numbers were developed. It would have been useful if McClure had specified where and how DuPont derived the numbers, 3 to 1 and 6 to 1, for productivity increases. In all three examples, the companies were excited about using CASE products for their system development, but they did not give any specific examples of previous versus current project development times, quality measures or maintenance requirements.

Rochester, 1989

Rochester, in the I/S Analyzer, developed the theme of "Building More Flexible Systems" with examples from DuPont Cable Management Services and Scott Paper Company.

DuPont Cable Management Services needed a flexible system in order to keep track of new kinds of equipment and to be able to sell the service to other companies. They employed DuPont Information Engineering Associates (also chronicled by McClure (1989)) which was a user of an in-house CASE prototyping tool, RIPP. DuPont developed the

cable management system within nine months; the expected development time, as reported by other telecommunications executives, for other similar systems was between two and three years. DuPont had been able to add additional capabilities to the system and also to sell the system to other companies; therefore DuPont met its original objective.

Scott Paper Company selected a single integrated CASE tool, Integrated Engineering Facility (IEF), which followed a specific methodology, rather than two different, non-integrated CASE tools. IEF from Texas Instruments had a total view of the organization and began with a corporate strategic plan. The article then described the seven phases used by IEF and Scott Paper in order to develop the system. Developers at Scott Paper did not have to use the CASE tool and CASE did not become a standard way to develop systems. Scott Paper concluded that:

1. Not all system development is appropriate for CASE
2. Not all projects need all seven steps
3. Since CASE is a new way of system development, it may take three to five years to get CASE fully accepted

4. CASE is not just a development tool; it is also a strategic tool for the whole corporation.

If the estimates from other telecommunications executives are accurate, the DuPont system appeared to have significantly improved productivity. Nothing was said directly about the quality of the system that was finally developed but since DuPont was been able to modify the system easily and also to sell it to other companies, it was implied that the system was of high quality. Some data that would be helpful for the productivity comparisons include the sizes of the other companies, the ranks of the other executives, the sizes of their development teams and use or non-use of CASE tools. Was the reported productivity CASE tool specific? In the Scott Paper synopsis, there was no mention of increased productivity or quality. It was simply a report of the way Scott Paper has begun integrating a CASE tool into their systems development area. Similar to McClure's (1989) accounts of three different systems, these were brief descriptions of two companies' experience with CASE tools. This article continued with more detail on CASE tools and how they assisted in building more flexible systems and in maintaining systems.

There are several other practitioner articles about specific companies adopting CASE technology, but they are written along the same vein as the previous examples. Admittedly, these are brief descriptions of the companies' experience with CASE tools, but other articles in the practitioner journals also appear to be lacking any hard data. They are subjective, anecdotal and descriptive, filled with managers' perceptions and comments. There are no quantitative productivity and quality measures to support their claims. This is what future users of CASE tend to believe when they begin to examine CASE technologies for their organizations.

THIS RESEARCH

This research is more quantitative than any of the previous studies on CASE. Benchmarks for both programmer productivity and system quality without the use of CASE technologies are established. The same system is developed using CASE technologies and the data collected is compared to the benchmarks. Statistical tests determine whether there is an increase in either programmer productivity or system quality. After a review

of the current literature, it appears that there is a need for a more quantitative approach for evaluating the effectiveness of CASE technologies.

CHAPTER 3

METHODS

THE EMPIRICAL STUDY - USAGE OF CASE TECHNOLOGY

The purpose of this research is to evaluate the effectiveness of the use of CASE tools during the design phase of software system development. The phased or waterfall model of the software development life cycle consists of requirements analysis, design, implementation or coding, testing and maintenance (Fairley 1985, p. 38; Pressman 1982, p. 129). These phases overlap and are often iterative. The primary activities during the design phase are the identification of the software modules or functions, data streams and data stores, and the definition of their relationships and connections (Yourdon 1979, p. 7). Freeman (1983) considers the design phase the central activity of the software development life cycle but coding, testing, and maintenance concerns also should be taken into account during the design of the system (Freeman 1983).

There are a number of reasons for studying Excelerator (Index 1987). The CASE tool currently available at the College of Business Administration of the University of Cincinnati is Excelerator. Excelerator was named Software Product of the Year in 1987 by the American Federation of Information Processing Societies (Hanna, 1990) and also has been the most widely used CASE product (Fersko-Weiss, 1990). Excelerator is the selected CASE tool for this research because of its emphasis on the design phase and the support of structured methodologies. McClure (1989, p. 158) introduces Excelerator as a "productivity tool aimed at designing and documenting information systems and real-time systems." The concept of Excelerator, according to the Excelerator User Guide (1987, p. 1-1), is it "provides all the capabilities you need to design and document systems." Excelerator supports the popular structured design methodologies of Yourdon (1979) and DeMarco (1979) which are taught throughout the first two years of the Information Systems curriculum and in the Systems Analysis and Design course.

According to advocates of CASE technologies (Acly 1988, Frenkel 1985, Gibson 1989, Lewis 1988), the final design should take less time to develop, have fewer errors and internally be more consistent than a design created

without the use of CASE technologies. Therefore, it may be inferred that the system resulting from a design developed with CASE tools also should have fewer iterations in the design phase and be less complex than a system developed from a design developed without CASE tools. But these perceived benefits of CASE use may or may not be real. The research question to be answered is, "Are there significant measurable differences in either the development process or in the final product?"

SCOPE OF THE EXPERIMENT

This research is a replicated project study with a control group; its purpose is to study the effect of different technologies, CASE versus non-CASE, when used to develop a software system. "Replicated project studies examine objective(s) across a set of teams and a single project." (Basili 1986, p. 735).

This study is a controlled experiment using student subjects implementing a classroom project. Controlled experiments in an organizational environment are too costly and time consuming (Myers, 1978). Real-world projects will not be replicated by software developers

because of financial and practical considerations; neither the system nor the programming teams are the same. This means it is often difficult to isolate and evaluate the effect of the technology being studied (Boehm 1981; Glass 1982; Attewell 1984).

Due to variations in task complexity caused by product differentiation, two commercial projects are rarely comparable (Humphrey 1988; Eliot 1986; Haas 1989; Basili 1981). Because different systems are being built, it is difficult to evaluate programmer productivity. This research attempts to determine programmer productivity on replications of the same system. One of the major determinants of productivity is project size (Behrens 1983, Boehm 1984b). Iterations of the same project allows this variable to be held constant. Additional determinants of productivity are the computer system and the programming language (Behrens 1983). In this research these variables are also kept constant. Therefore, the same task was given to a number of teams of university students enrolled in two sections, across two quarters, of an Information Systems course titled Software Engineering.

Data collection is easier with student subjects: practitioners are reluctant to change their way of doing business (Card 1988). Students are a convenient sample (Beath 1988) and since these students are Information Systems majors, they are also a representative sample of future users of CASE technologies. Haas (1989) used university students as subjects to study measures of problem size proposed by DeMarco (1982). Maintenance tasks performed by students were used by Rombach (1987) in a controlled experiment to evaluate the effect of software structure on program maintainability. In order to understand software development methodologies, Basili and Reiter (1981) also used student programming teams. The Cleanroom approach developed by IBM was evaluated using university students (Selby 1987). Other project studies that used university students as subjects include Gannon (1977), Basili (1983), Boehm (1984a), Hall (1986), Knight (1986) and Joyce (1987).

THE COURSE

In this research, the two groups of subjects are students in the same course, in different quarters, in a near lock-step curriculum. Consequently, the subjects have similar

characteristics individually and teams were chosen to try to insure team similarity. The same instructor conducted both sections of the course in order to maintain consistency for the presentation of the course material. The experiment did not interfere with the quality of the instruction given to the students. The main objective of the course is to gain knowledge of structured methodologies and to become familiar with "programming in the large" by working in a programming team environment and implementing a small (600-2,000 line) system. Although 600-2,000 lines is not "programming in the large," this is the size that is appropriate for the limited amount of time in a ten week quarter. The students used structured methodologies to implement and complete a given project.

THE TASK

The experiment was conducted over the two quarters using the same task. The project consisted of designing, coding, testing, debugging and documenting a pretty printer for Pascal programs. The project is of moderate difficulty and length; it is a non-trivial problem, resulting in an average of 1500-2000 lines of code.

A pretty printer is a computer program that reformats computer programs (Cameron 1988; Oppen 1980; Rubin 1983). The new reformatted version of a computer program should be easier to understand and read. The original version of the computer program is stored in a text file on a VAX 6350 under the VMS operating system and is treated as input in the form of character strings. It is not the purpose nor the responsibility of the pretty printer to detect syntax errors. The pretty printer may assume that the input is a text file containing a syntactically correct Pascal Program. The output must also be a syntactically correct program with the same execution behavior as the input.

The pretty printer should add or modify appropriate line breaks, spacings and indentations. If there are several Pascal statements on a single line, that line should be separated into several lines, with one statement per line. If there is uneven indentation for a certain construct, e.g., an IF THEN ELSE, the indentation should be made consistent. Blank lines should be inserted between sections and procedure and function declarations. For a complete set of requirements see Appendix A.

The variable names and constant names should be alphabetized within their respective sections. PROCEDURES also should be alphabetized, but here there is a choice of how that is to be accomplished; either rearrange the PROCEDURE code into alphabetical order or create an index of the PROCEDURE names. If PROCEDURES are physically alphabetized, either the author of the original Pascal program must have included FORWARD statements in the program, or the pretty printer has to recognize that there are no FORWARD statements and create them. The FORWARD statements are needed because the PROCEDURE that is being called must be physically ahead of any procedure or function from which it is being called. If the PROCEDURES are physically alphabetized, there is a strong possibility that this rule will be violated. Use of the FORWARD statement eliminates the need for this ordering and the PROCEDURES can be arranged in any sequence. The second method of alphabetizing involves creating an index of the PROCEDURE names and their line numbers. PROCEDURE names must be alphabetical in the index. The purpose of either method is to facilitate a search for PROCEDURE code.

The final modified Pascal program is either stored in another file, displayed on the screen or printed. Any combination of the three options may be required by the

user of the pretty printer. The choice is determined by the needs of the person who is submitting the Pascal program to the pretty printer. For complete specifications see Appendix A.

Identical detailed requirements prepared manually by the instructor were given to the students at the beginning of each quarter. The requirements were not prepared using CASE technologies; use of CASE might provide an advantage to CASE users and have a negative impact on non-CASE users. All students had available the same computer resources, the same programming implementation language, the same debugging tools and were constrained by the 10-week-quarter time period.

The task is broken down into two major phases: design and implementation, with the emphasis on the design phase (first five weeks). Students in the spring quarter did not use CASE technologies while they learned structured methodologies; those in the autumn quarter used CASE technologies.

THE SUBJECTS

Participants in the study are junior level Information Systems majors enrolled in the Software Engineering course in the College of Business Administration, University of Cincinnati. Most of the students majoring in Information Systems will be the systems analysts of tomorrow; therefore, the results are likely to generalize to the entire population of professional systems analysts. All students enrolled in the two sections participated. In order to enroll in the course the students must have completed all their freshmen and sophomore level Information Systems courses: Introduction to Data Processing, Principles of Structured Programming, COBOL I, COBOL II, and Data and File Structures. The structured programming concepts included in these previous Information Systems courses are formalized and expanded within the scope of a larger project in the Software Engineering course.

Almost all of the students are familiar with the computer system and the implementation language. Most previous Informations Systems programming courses use the VMS operating system and two of the prerequisite courses,

Introduction to Data Processing and Data and File Structures, are Pascal-based courses. The students were not familiar with a team concept of programming nor formal structured software development methodologies.

During the 1989 spring quarter, the student teams implemented a pretty printer using structured methodologies, including top-down design and structured programming methods (Yourdon 1979; Page-Jones 1988). They did not use any of the integrated CASE technologies. All of the teams decided, independent of class instruction, to learn and use FLOW (Patton 1986) for their structure charts and data flow diagrams. FLOW is a graphics word-processor. All the checking and verifying within the data dictionaries, structure charts and data flow diagrams were done manually (without any computer aided integration). These seven three-person teams are the control group or benchmark for the experiment.

During the following autumn quarter, the same task was implemented. Student teams received the same instruction and used the same structured methodologies as in the 1989 spring quarter but also were required to use the available integrated CASE technologies during the design phase. It should be noted that the subjects in the autumn quarter

were novices in CASE; novices are not always as successful as seasoned users (Basili 1981).

Teams worked independently, with no collaboration among teams, during all phases of project. There was a friendly spirit of competition; each team was attempting to design and code the best pretty printer. Students in the autumn quarter were aware that they were working on the same project as the students in the spring quarter; there might have been some sharing of ideas, but the designs in the autumn were different from those in the spring. Any communication across quarters probably did not have a significant impact on the autumn projects. Also, students tend not to plagiarize on large scale projects. It has been observed by the faculty that, while students may give students in a following quarter a small, several hundred line program, they are not willing to do the same with a full-quarter project. Students were aware they were being monitored, but did not know why or exactly what was being observed. Measures were taken to insure privacy; after the data was collected and the grades for the quarter assigned, the data was not associated with identifiable individuals.

TEAM COMPOSITION

In a previous study using student teams, Rombach (1987) ranked students on their educational performance (grades), experience (industry) and relative programming talent. In this study, educational performance and programming talents were combined as a measure of the students' programming ability and were used in conjunction with work experience in determining team composition. Each team had a "more experienced" member, a "less experienced" member and an "inexperienced" member. "More experienced" is defined as either several quarters of co-op work experience or more than a year of part-time work and familiarity with several operating systems. "Less experienced" students have one or two quarters of co-op experience or less than one year of part-time work and familiarity with one or two operating systems. The students classified as "inexperienced" have little or no practical work experience; most of their knowledge about the field has been acquired from their courses. Demographic data, age and sex, and information about the level of experience was collected using a pretest questionnaire (Appendix C) and the level of experience was evaluated by the instructor, a graduate student and a senior Information Systems major (grader). Data regarding

students' ability was also gathered from several of the students' previous instructors and the grades (Basili 1981) earned in their prerequisite Information Systems courses. As long as the teams maintained this mix of experience and ability, some consideration was given to students' preferences in regard to team members.

In the spring quarter there were seven three-person teams and in the autumn quarter there were three three-person teams and one four-person team. The four person team had one "more experienced" member, one "less experienced" member and two "inexperienced" members. This team was formed as a three-person team, but two students from another team dropped the course and the remaining student had to be placed on a team. This particular team was selected because the students' level of ability was judged to be slightly lower than the other three teams. The two groups were not different in course background, work experience, age or grade point average (GPA). One subject in the treatment group did not turn in a questionnaire and the following levels of significance (using t-tests) are calculated using the scores for 20 students in the treatment group and 13 students in the control group.

The control group (non-CASE) consisted of twenty-one students which formed seven three person teams. The average age for the group was 23.05 years, with a range from 21 through 29. The average GPA for Information Systems courses was 3.44 out of 4.0 and the average GPA for all courses was 3.13.

The treatment group (CASE) consisted of thirteen students which formed four teams, three three-person teams and one four-person team. The average age for the group was 22.92 years, with a range from 19 through 31. The average GPA for Information Systems courses was 3.43 and the average GPA for all courses was 2.91. The p values and t scores for age, overall GPA and Information Systems GPA are shown in Table 3.1.

TABLE 3.1

SIGNIFICANCE VALUES AND T SCORES FOR GROUP CHARACTERISTICS

(31 degrees of freedom)

VARIABLES	T SCORES	P VALUES
AGE	0.120	0.905
GPA (overall)	1.830	0.077
GPA (I.S. only)	0.048	0.962

Using a two-tail test, none of the means were significantly different at the 0.05 level.

All the data were transformed into z-scores, that is, the values for the team variables were normalized. Using the SPSSX RELIABILITY procedure with the ALPHA model, Cronbach's alpha was calculated for each of the two groups for the complexity, size and time categories. See Chapter 4 for additional information about the three categories. Each group's alpha values for each category are shown in Table 3.2. The SPSSX code and relevant results are in Appendix I for group 1 and Appendix J for group 2.

TABLE 3.2

CONSISTENCY MEASURES (CRONBACH'S ALPHA)

ALL SUBJECTS

	CONTROL GROUP	TREATMENT GROUP
COMPLEXITY	0.8737	0.9277
SIZE	0.7921	0.9394
TIME	0.7180	0.7336

Any alpha value greater than .8 is a good indicator of internal consistency. Internal consistency or less

variance among the subjects in the same group is an indicator that the subjects tended to perform in a similar manner for the same variables. All the alpha values for the treatment group were well above the .8 level; those for the control group were above or close to the .8 level. However, the alpha values for the treatment group were higher than those for the control group. This is an indicator of more internal consistency and less variance within groups, and although the treatment group appears to have greater consistency than the control group, there is little variance within either group. See Appendices H and I for copies of SPSSX code and relevant output.

THE PROCESS

Students were given the specifications (Appendix A) for the pretty printer during the first week of class. Those in the treatment group (those that were required to use Excelerator) were told that they should start learning how to use that software. Those in the control group (without Excelerator) were told that they could use any software that was available; there were no restrictions. However, Excelerator or other integrated CASE products were not available. Class lectures for the first four weeks of the

quarter covered structure charts, data flow diagrams, data dictionaries, module specifications and interface specifications.

Time for questions was allotted at the beginning of each class period; there were two 75 minute class meetings per week. The control group had so many 'what if' questions that really should not have been of concern, that six assumptions (Appendix B) about the input data were generated, thus eliminating some of their trivial questions. These assumptions were given to the Excelerator group during the end of the second week of the quarter. At that point, several of the same questions had arisen; possibly additional questions were eliminated with the distribution of these assumptions.

Design walkthroughs were conducted near the end of the third week. This allowed ample time for major modifications, gave everyone the chance to experience a walkthrough, made sure that everyone was keeping up with their work and enabled others to look at other designs. These walkthroughs were not graded; it was a 'free' look at designs. Classmates made positive suggestions, often catching omissions and errors. Often there were comments like, "Why didn't we think of that," "We want to see that

finished project," or "We don't like that approach," but the walkthroughs were conducted in a very positive atmosphere, perhaps because there was no grade attached. The presentations were all very well done, with good use of visual aids and handouts. It was almost as if each team was trying to impress the others.

During the autumn quarter, Excelerator was installed on four Zenith personal computers with a 808386 central processing unit chip. For the autumn quarter, any questions on Excelerator were answered in class. If the instructor did not know the exact answer, usually someone else in the class already had reached the same stage and already had a solution. However, except for distributing passwords and project designations, the only class time spent on Excelerator was that time used to answer students' questions.

Designs were collected, graded and returned during the fifth week. Included in the design package were the structure charts, data flow diagrams, data dictionaries, module specifications, interface specifications and a brief verbal description of the team's design with an explanation of a pretty printer. In both instances, designs were handed in on Tuesday and returned, with

comments and a grade, on Thursday. The CASE generated data flow diagrams and structure charts were more complete than the non-CASE; they were not missing any of the 24 specifications, were fully labeled and were not missing inputs or outputs. Although the data dictionaries generated by Excelerator contained definitions for all the data and processes from the data flow diagrams, the format was harder to understand than the manually generated data dictionaries. Grades for both quarters ranged from 84 to 94.

The coding phase then began. Class lectures concentrated on modularity, cohesion, coupling, fan-in, fan-out and external procedures in Pascal on the VAX. All pretty printer procedures were required to be external; no procedure was allowed to have another procedure nested within. There were two reasons for this restriction. First, the students should be thinking of "programming in the large" with other programmers using their code and creating libraries of code that can be used for other systems. Many of the procedures that are nested within another procedure have the potential for use by other routines and if nesting occurs, this sharing of code cannot take place. Secondly, external procedures, without any nested procedures, were the input for the Pascal

metrics program that was used for token counting; procedures with nested procedures would cause erroneous tabulation.

Students set up VMS command files to facilitate the compiling and re-compiling, linking and running of their external procedures. Command (COM) files are programs written using VMS DCL statements that can perform any of the commands that can be individually written at the VMS "$" prompt. By combining all these commands into a file, the file can be run and all the commands in the file are then executed with just the one statement. Not all students were familiar with command files and some class time was spent, both quarters, reviewing the setup of these files and their functions. The students also used the COM files as an interface between the user and their final pretty printer. These programs queried the user as to the input file that was to be sent to the pretty printer and the medium for the output: file, screen, printer or any combination of the three outputs.

During the coding phase some teams' members specialized. Some teams divided the work for the project into coding and documentation, while others had three divisions; coding COM files, Pascal coding and documentation. Still

other teams had all members working on all phases of the implementation. This division of labor did not occur during the design phase, but in recognition of some of the members' abilities, responsibilities during the coding phase were often separated and distributed. All these decisions were made by the teams themselves, without any consultation with the instructor.

The project was was handed in during the tenth week of the quarter. Required in this phase was the Pascal code, a user's manual, a programmer's manual and any changes made to the original design and an explanation as to why the changes were necessary.

During exam week, the 11th week, the students worked on a take home final and the testing of another team's project. They were allowed to work as a team or as individuals, since some of the team members were definitely tired of each other. Either way, they wrote sample programs to test the pretty printer, and wrote a few pages describing the accomplishments and shortcomings of the project tested. They were told to treat the project that they had to test as a software package that they just purchased. They were given copies of the users' manual which should correspond to the documentation that

accompanies purchased software. Part of the evaluation consisted of critiquing the quality of the users' manual. Some of the best evaluations were those that compared the Pascal program before and after the pretty printer; copies of each version were turned in and the students marked directly on the listings what the changes were. They referred to the original specifications and listed successes and failures. They enjoyed looking at other students' work and seeing the results.

DATA COLLECTION

The question of "what to measure" always arises. Yourdon (1989a) emphasizes that measuring the "process" used in developing the product is just as important as measuring the "product." In order to determine the effect of the use of CASE technology during the design phase of the software development life cycle, aspects of both process and product were measured. Data was collected both manually and automatically during the process and on the developed system (Card 1987).

DATA COLLECTION - THE PROCESS

The process was evaluated using the data collected during the design phase and information collected automatically during the implementation phase.

I felt that some part of the students' final grades should depend upon their progress reports and personal logs, otherwise the students would be unlikely to keep track of their activities nor turn in any reports of those activities. Data involving the design phase was collected using weekly progress reports, minutes of team meetings and personal logs. The progress reports and the minutes were turned in weekly; the logs at the end of the quarter. Evaluation was based on completeness, not correctness; correctness would be very difficult to determine. The students were told that their logs were supposed to contain a record of all the activities associated with the Software Engineering course and the amount of time spent on each activity. This information should have been recorded daily or whenever the students work on the course. Also included in the logs should have been copies of progress reports and the team minutes.

There was no precise format recommended for the progress reports, minutes or logs and an exact format would have been helpful to both the students and myself. The students work better with formal guidelines and a precise format would have facilitated tallying the data in the logs. Logs were also to include any decisions that were made about the design. Students were told that their logs should be so complete that if someone would replace them on the project, their logs would serve as an introduction and a clarification of the work already in process and explain both how and why things were being done.

In order to prevent the logs being done at the end of the quarter and therefore not being an accurate picture of the individual processes, they were date stamped every week. Requirements for the logs are included in Appendix D. Table 3.3 lists and defines the variable names that were assigned to the data accumulated from the student logs. This data was accumulated by a senior in Information Systems who examined students' logs and tallied their reported individual times and team times spent for each phase of the project.

TABLE 3.3

TIME VARIABLE NAMES

COLLECTED FROM STUDENT LOGS

NAME	DESCRIPTION/DEFINITION
INDDES	The amount of time an individual reported that he/she working by his/herself spent on the design phase of the project.
INDCOD	The amount of time an individual reported that he/she working by his/herself spent on the coding/implementation phase of the project.
GRPDES	The amount of time an individual spent with his/her team working on the design phase of the project. This time is from the team's minutes for its meetings.
GRPCOD	The amount of time an individual spent with his/her team working on the coding/implementation phase of the project. This time is from the team's minutes for its meetings.

TABLE 3.3 (continued)

TIME VARIABLE NAMES

COLLECTED FROM STUDENT LOGS

TOTDES	The total amount of reported time that a team spent on the design phase of the project. Sum of GRPDES and INDDES for all the team members
TOTCOD	The total amount of reported time that a team spent on the coding/implementation phase of the project. Sum of GRPCOD and INDCOD for all the team members
TOTTME	The total amount of reported time that a team spent on the entire project - design and coding/implementation phases. Sum of TOTDES and TOTCOD

Data from during the coding phase was automatically collected. Each logon, compilation, link, run and logoff was recorded. The total amount of time on the system, and the counts for the compilations, links and runs are used to determine whether use of CASE technology has reduced

the number of iterations necessary to implement the system. This data was collected for each individual student and combined to form the team totals.

Several programs have been created that collect this data without the students' interaction. The students knew that data about their project was being collected, but they did not know how or why. Table 3.4 lists and defines the variable names that were assigned to the data accumulated from these programs.

TABLE 3.4

TIME VARIABLE NAMES

COLLECTED AUTOMATICALLY

NAME	DESCRIPTION/DEFINITION
ICOMPL	The number of compiles for an individual student
GCOMPL	The number of compiles for a team
ILINKS	The number of links for an individual student
GLINKS	The number of links for a team
IRUNS	The number of runs for an individual student

TABLE 3.4 (continued)

TIME VARIABLE NAMES

COLLECTED AUTOMATICALLY

GRUNS	The number of runs for a team
ITIME	The amount of time an individual student spend logged onto the VMS system.
GTIME	The amount of time a team spend logged onto the VMS system
ILOGON	The number of times an individual student logged onto the VMS system
GLOGON	The number of times a team logged onto the VMS system

DATA COLLECTION - THE PRODUCT

The product was evaluated using several different programs that collect data about the finished product and which compute "software metrics" (Halstead 1977).

Most of the seminal research on software metrics has been performed by Maurice Halstead (1972) and Thomas McCabe (1976) (Curtis 1983); both Halstead's and McCabe's metrics attempt to define and measure the complexity of a software

task. The software metrics used do not depend on any specific type of development methodology (Grady 1987). One of the major results of a study done by Basili, Selby and Hutchens (1986) was that metrics could differentiate between systems developed using different methodologies. It has been shown that metrics can also differentiate between systems developed using different development technologies; keeping the development methodologies constant. In this instance, the comparison is between a software system developed without CASE technologies and one developed using CASE technologies.

Some of the metrics of interest include, but are not limited to, the number of executable lines, the number of modules, the number of decisions, the number of tokens, the average number of statements per module and program data coupling. Gilb (1977 p.88) counts the number of IF statements in order to measure the logical complexity of a computer program.

The complexity of a program is of interest because it affects the development time, the number of defects and the ease of modification (Weyuker 1988). However, it is not always easy to determine what to measure. As previously stated, an improved design should reduce the

complexity of the product. The number of lines of source code is the most often used and most intuitive indicator of complexity (Boehm 1987). There is often discrepancy about the composition of the "lines of source code." Blank lines, comments and declarations may or may not be counted. Arthur (1983 p.133) defines lines of code as those lines with "action verbs." Teams may adopt various "comment" philosophies and this will affect both the number of lines in a program and the amount of time spent on the "machine." Commenting, as well as coding, involves cognitive activity and should be part of the time spent developing the system.

In this study, separate counts of actual "action" lines and comment lines were taken. If the teams themselves do not establish standards, there also may be variance within teams as to the commenting philosophies of the team members. As long as we are consistent in our definition of "source code" and are comparing size within the same language and for the same application, size does measure complexity and productivity (Humphrey 1989, p. 317).

Other measures of complexity involve counting the number of operators and operands in a program. Operands are constants and variables (Halstead p. 5). Halstead defines

operators as any symbols or combinations of symbols that affect the value or ordering of an operand (Halstead p. 5): they may be function calls, mathematical symbols, delimiters or keywords. Jones (1986, p. 108) suggests a data complexity measure that uses the number of comparison operators as a complexity indicator. The more decisions that have to be made to solve the algorithm, the greater the complexity of the algorithm.

Elshoff (1984) studied 20 candidate measures of program complexity and located a set of four measures (using a measurement system developed by Elshoff) that would define program complexity and could be used to classify programs. His four measures are length of the program, number of unique operators, data difficulty and the number of unique operands. Length is the sum of the number of operators and the number of operands (Halstead 1977). Data difficulty is defined as the total number of operands divided by the number of unique operands.

Stevens, Constantine and Meyers (1974) studied the structural complexity of programs. They define absolute structural complexity as the number of modules in the system and relative structural complexity as the ratio of the number of linkages to the number of modules. They

propose that fewer connections imply a less complex system. Jones (1986, p. 108) uses the term "functional complexity" to define the flow of control and the number of linkages.

A less complex program is easier to understand and maintain; a better design should produce a clearer program. Halstead (1977, p.19) defines program volume as the size of an algorithm expressed in bits or the count of the number of mental comparisons required to write a program (Halstead 1977, p. 47). If the same algorithm is coded in a different language, the volume of the program will change. Lower level languages will require more operators and operands to program the same algorithm than a higher level language.

Halstead (1977, p.9) defines program length as the sum of the total number of operators (N_1) and total number of operands (N_2). This measure is the actual implementation length; how many operators and operands were used to implement the algorithm. Program length, most often calculated as the sum of the number of operators and the number of operands, is considered a direct observation. Actual program length is calculated as:

$$N = N_1 + N_2$$

Halstead intuitively used thermodynamics and information theory to estimate program length as follows:

$$n_1 * \log_2 n_1 + n_2 * \log_2 n_2$$

where n_1 is the number of unique operators in the program and n_2 is the number of unique operands. This estimated program length assumes good program construction; a pure or polished program was written to implement the algorithm (Fitzsimmons 1978). If the program is well-written, the observed program length should not differ greatly from the estimated program length.

A measure of program clarity is proposed by Gordon (1979a): the measure of program clarity is Halstead's program volume divided by Halstead's estimated program level in units of elementary mental discriminations; the number of individual mental calculations needed to solve or understand a problem. Program level is calculated as follows:

$$(2 * n_2)/(n_1 * N_2)$$

Gordon (1979b) measures the amount of mental effort needed to understand a program and supports his claim that effort is a measure of clarity. The less mental effort the clearer the program; the better constructed program

requires less effort, therefore is clearer, than a poorly constructed program.

Fitzsimmons and Love (1978) reviewed and evaluated Halstead's software science theories. They concluded that "software science is a possible tool for answering" questions about properties of software development projects and the difficulties of programming.

In this study the following metrics were obtained using a program that used the actual finished project procedures as input character strings and counted the number of unique tokens and the total number of occurrences for each token. Counts were obtained for each module in a project; these counts were then summed for total project count for each metric for the team project. This was done for each of the eleven projects, not for each individual student. The student teams handed in a single project and in most cases the author of individual modules was not identified. The major interest is in the completed pretty printer. Table lists 3.5 the variable names that were assigned to each of the counts and an explanation of each of the counts.

The number of unique operators (n_1) was incremented by 1 each time a new token was encountered; for this metric the same token is not counted more than once. The following operators were counted: PROCEDURE, FUNCTION, REPEAT, WHILE, UNTIL, FOR, BEGIN, END, CASE, IF, THEN, ELSE, REWRITE, RESET, READ, READLN, WRITE, WRITELN, AND, OR, NOT, WITH, DIV, MOD, all mathematical and Boolean symbols, and all delimiters. The total number of operators (N_1) used throughout the system is the sum of the frequency of the unique operators (FEXTPR, FEXTFN, LOOPS, SELECT, OPNOUT, OPNIN, READ, WRITE, LOGIC, MTHSYM, RCDS, SETS, SYMBOLS).

The number of unique operands (n_2) is the sum of the number of unique constants (CONST), the number of unique variables (VARBL) and the number of unique literals (LITERAL). The total number of operands (N_2) used throughout the system is the sum of the frequency of the unique operands (FCONST, FVARBL, FLITERAL).

TABLE 3.5

VALUES COUNTED DIRECTLY FROM THE FINAL SYSTEM

NAME	DESCRIPTION/DEFINITION
LOC	Number of lines of code (excluding comment lines and blank lines)
CMMNTS	Number of lines of comments
CONST	Number of unique constants - from the CONST section
FCONST	Constant frequency - number of total constants - from the CONST section and usage throughout the project.
VARBL	Number of unique variables - from the VAR section
FVARBL	Variable frequency - number of total variables - from the VAR section and usage throughout the project.
LITERAL	Number of unique literals used throughout the project.
FLITERAL	Literal frequency - number of total literals used throughout the project.

TABLE 3.5 (continued)

VALUES COUNTED DIRECTLY FROM THE FINAL SYSTEM

PRMTRS	Number of unique parameters - from the heading of a PROCEDURE
FPRMTR	Parameter frequency - number of total parameters - from the heading of a PROCEDURE and usage throughout the procedure.
EXTPRC	Number of external procedures declared
FEXTPR	Procedure frequency - number of external procedures declared and invoked
EXTFNT	Number of external functions declared
FEXTFN	Function frequency - number of external functions declared and invoked
LOOPS	Loop frequency - number of REPEAT, WHILE and FOR loops used
SELECT	Selects frequency - number of CASE, IF and ELSE statements used THEN is part of the IF statement

TABLE 3.5 (continued)

VALUES COUNTED DIRECTLY FROM THE FINAL SYSTEM

OPNOUT	Total number of times files are opened for output using the
REWRITE	statement.
OPNIN	Total number of times files are opened for input using the RESET statement
READ	Total number of READ or READLN statements
WRITE	Total number of WRITE or WRITELN statements
LOGIC	Total number of AND, OR, NOT, >, <, >=, <=, =, <> tokens used throughout the project.
MTHSYM	Total number of mathematics symbols (+, *, -, /, DIV, MOD) used throughout the project.
RCDS	Total number of times a record field is used - counting the WITH statements and the '.' for qualifying the record field.
SETS	Total number of times a set is used - counting IN statements.

TABLE 3.5 (continued)

VALUES COUNTED DIRECTLY FROM THE FINAL SYSTEM

SYMBLS	Total number of symbols (:, :=, ',', (*, *), [,], (,), ;) used throughout the project.

In this research the above measures are used to define three areas of interest: complexity, size and time. Complexity comprises of the number of lines of executable code (LOC), Gordon's clarity measure (CLARITY), Halstead's number of unique operands (n_2) and number of unique operators (n_1), Elshoff's level of data difficulty (DATADIFF), the number of selection statements (SELECTS), the number of iteration statements (LOOPS), the number of blocks of code (BLOCKS),and the level of difficulty (DIFFICULTY). Lines of executable code (LOC), the number of lines of comments (CMMNTS),the number of modules (MODULES) and Halstead's volume (VOLUME), vocabulary (VOCAB), implementation level (IMPLEVEL), estimated length (ESTN) and program length (LENGTHN - sum of all occurrences of operators and all occurrences of operands) form size. Time is a composite of the number of times a team logged onto the system (GLOGON), the number of compiles (GCOMPL), the number of links (GLINKS), the

number of runs (GRUNS), the amount of time spent on the system (GTIME), the amount of reported time spent on design (TOTDES), the amount of reported time spent on coding (TOTCOD) and the total reported time spent during the quarter (TOTTME). All the variables defining the time category are team measures; they have been summed for the students within the team. The number of lines of executable code (LOC) are included in evaluating both complexity and size. Intuitively, the complexity of a system increases as the number of lines of executable code increases. The number of lines of executable code is also an indicator of the size of the system.

Defects in the final product can be evaluated according to Yourdon's (1989a) three-way breakdown of defects: defects due to coding, design, and/or documentation. The number of defects is a measure of the quality of the system. Each system was tested for completeness, how well the system met the initial requirements, using several different Pascal programs as input data, each having different types of syntax that need to be reformatted. Copies of the test programs can be found in Appendix E.

Systems that did not handle all the required formatting are considered incomplete. Incorrect attempts at

formatting are considered defects and further investigation was made to determine whether the defect was in the implementation or in the design. Defects were attributed to logic errors, poor data definitions, poor module interfaces and so forth (Humphrey 1989, p. 314).

A senior in Information Systems evaluated the projects for completeness. He was given all eleven projects to evaluate, but did not know which projects were developed using CASE and which were developed without CASE. The projects were shuffled and the seven without CASE were randomly merged with the four that used CASE. The instructor kept a key to the projects. The original specifications were used to determine the completeness of each project. On each of the different specification, a "0 - 1 - 2" scale was used to record the completeness. A score of 0 indicates that no attempt was made to accomplish that specification. A score of '1' indicates that some attempt was made, but it was not a totally successful attempt. A score of '2' indicates a totally successful attempt at a particular specification. Once the scales were returned to the instructor, they were re-connected with their appropriate team.

Any inconsistencies in either the programmer's manual or the user's manual are considered documentation defects. These defects may be internal to the documentation itself, or external, that is, conflicting with the actual coded system. Since the senior Information Systems major was using the manuals to understand and test the projects, he also was in a position to evaluate their effectiveness. This evaluation is in the form of comments about the manuals' effectiveness, clearness, appropriateness and information content.

A system not only consists of the actual code, but also the documentation that accompanies the code. If CASE is effective, the system developed using CASE technologies should have fewer defects, be more complete, have better documentation and be of higher quality than the system developed without CASE.

CHAPTER 4

RESEARCH MODEL

Two major objectives of CASE technology are "improving the productivity" during system development and "improving the quality" of the software system being developed. (Chikofsky, 1989; McClure 1989 p.6). This research used twenty-seven variables to test "increased productivity" and "improved quality." The twenty-seven variables were divided into four major categories: completeness, complexity, size and time. The completeness level is a single variable giving the number of requirements met by each of the systems. A complete system would have a completeness level of 48, twice the number of requirements. There were eight variables in the time category, eleven in the complexity category and eight in the size category. Because lines of code (LOC) is a measure of both the size of a program and the complexity of a program, this variable was included in both the complexity category and the size category. The hypotheses were tested using the difference in the means

for variables in the control group (non-CASE) and the treatment group (CASE).

This chapter presents the model for this research. First, the two major research questions are stated. Next the formal hypotheses (A, B) that support each of the questions is given. Each of the formal hypotheses (A, B) is then operationalized by two sub-hypotheses (A1, A2, B1, B2). Finally the categories and variables that test these hypotheses are described.

RESEARCH QUESTIONS

The following two questions are addressed in this study:

Belief 1: Use of CASE technologies increases the productivity of the programmer.

Belief 2: Use of CASE technologies increases the quality of the system/program being developed.

BELIEF ONE

The formal hypothesis that tests the first belief is:

(A)

H_0: There is no difference in the productivity of the programmer who uses CASE technologies and the productivity of the programmer who does not use CASE technologies.

H_1: Programmer productivity is greater when CASE technologies are used than when CASE technologies are not used.

This hypothesis is made operational with the following hypotheses:

(A1)

H_0: There is no difference in the amount of time required to produce a system using CASE technologies and the amount of time required to produce the same system without using CASE technologies.

H_1: The amount of time required to produce a system using CASE technologies is less than the amount of time required to produce the same system without using CASE technologies.

(A2)

H_0: There is no difference in the completeness of the system designed with CASE technologies and the system designed without CASE technologies developed in the same or less amount of time.

H_1: The level of completeness of the CASE produced system is greater than the level of completeness of the non-CASE produced system for the same or less amount of time.

The first operational hypothesis (A1) was tested using the amount of time that the control group (non-CASE) and the treatment group (CASE) used to code the pretty printer.

Eight variables were used to define time. The number of logons (GLOGON), the number of compiles (GCOMPL), the number of runs (GRUNS), the number of links (GLINKS) and the amount of time (GTIME) spent on the VMS operating system were collected automatically. The other three time variables were the self-reported design time (TOTDES), the self-reported coding time (TOTCOD) and the total self-reported time (TOTTME). Table 4.1 lists and defines the variables that were used to form the time category.

TABLE 4.1

VARIABLES USED TO DEFINE THE TIME CATEGORY

NAME	DESCRIPTION/DEFINITION
TOTDES	The total amount of reported time that a team spent on the design phase of the project.
TOTCOD	The total amount of reported time that a team spent on the coding/implementation phase of the project.
TOTTME	The total amount of reported time that a team spent on the entire project - design and coding/implementation phases
GCOMPL	The number of compiles for a team
GLINKS	The number of links for a team
GRUNS	The number of runs for a team
GTIME	The amount of time a team spend logged onto the VMS system
GLOGON	The number of times a team logged onto the VMS system

The number of compiles (GCOMPL), the number of links (GLINKS), the number of runs (GRUNS) and the number of logons (GLOGON) were included in the time category

because they may be used to explain how the time spent on the VMS operating system was being used or what activities were going on during that time period.

In addition to evaluating each of the individual variables, a combined time category was tested for significance. The eight time variables were converted to z scores. Discriminant analysis (SPSSX) summed the eight transformed time variables and calculated the significance for the time category. In order to reject the first operational hypothesis H_0 for A1 for productivity, the values for all or most of the time variables should be significantly less for the treatment group (CASE) than for the control group (non-CASE). If the same system was developed in less time using CASE technologies than without CASE technologies, we can claim increased programmer productivity.

The second operational hypothesis H_0 for A2 was tested using a combination of the time category and the level of system completeness. The systems were rated on the level of completeness of 24 logical functions (Appendix A) that were specified in the requirements. A "0 - 1 - 2" scale was used for the rating. A totally complete system, one

that completely met all the requirements for all functions, had a rating of 48.

If a more complete system, one that met more of the initial requirements, was produced in the same or less amount of time using CASE technology than a system produced without CASE technology, then the second operational hypothesis H_0 for A2 for productivity will be rejected, and we can claim increased programmer productivity. If the level of completeness for the CASE group was better than the level of completeness for the non-CASE group in the same amount of time, then it can be said that the CASE group was more productive than the non-CASE groups. If the level of completeness was the same for the two groups, but the CASE group used less time than the non-CASE group, then it can be said that the CASE group was more productive than the non-CASE group. If the level of completeness was higher or the same for the treatment group (CASE), but more time was required to produce the system, then the null hypothesis cannot be rejected and no conclusions can be made about programmer productivity.

BELIEF 2:

The formal hypothesis that tests the second belief is:

(B)

H_0: There is no difference in the quality of the system/program developed using CASE technologies and the quality of the system/program developed without CASE technologies.

H_1: System quality is greater when CASE technologies are used than when CASE technologies are not used.

This hypothesis is made operational with the following hypotheses:

(B1)

H_0: There is no difference in the complexity of a system produced with CASE technologies and the complexity of a system produced without CASE technologies.

H_1: A system produced with CASE technologies is less complex than a system produced without CASE technologies.

(B2)

H_0: There is no difference in the completeness of a system produced with CASE technologies and the completeness of a system produced without CASE technologies.

H_1: A system produced with CASE technologies is more complete than a system produced without CASE technologies.

The quality of the final system was tested using the complexity, size and completeness of the final system. A higher quality system should be less complex and more complete than system of lower quality.

The first operational hypothesis H_0 for B1 was tested by comparing the complexity and size of the systems developed without CASE technology and the complexity and size of the systems developed with CASE technology.

Although size is an additional indicator of the complexity of a system, for the sake of clarity, size measures are discussed separately from complexity measures. Size is an indicator of complexity; the larger the system the greater the potential for increased complexity. The systems developed for this research should not differ in size; they were all developed using the same requirements. Therefore, it was not expected that size would contribute to the complexity measure of quality. However, if the sizes of the systems are the same and less time was used to develop the same size

system, it could be an indication of increased programmer productivity. Table 4.2 lists and defines the eight variables that were used to form the size category.

The number of lines of code (LOC), the number of comments (CMMNTS) and the number of modules (MODULES) were directly counted in the final systems. LENGTHN, ESTN, IMPLEVEL, VOLUME and VOCAB are Halstead's Software Science measures and were all calculated from values that were counted in the final systems. They were included in the size category because they are different ways of expressing the length or volume of a prcject.

In addition to evaluating each of the individual variables, a combined size category was tested for significance. The eight size variables were converted to z scores. Discriminant analysis (SPSSX) summed the eight transformed size variables and calculated the significance for the size category. In order to reject a null hypothesis that the sizes were equal, the values for all or most of the size variables should be significantly different between the control group (non-CASE) and the treatment group (CASE). We would not expect the system sizes to be significantly different; they were all developed from the same requirements. We can claim

TABLE 4.2

VARIABLES USED TO DEFINE THE SIZE CATEGORY

NAME	DESCRIPTION/DEFINITION
LOC	Number of lines of code (excluding comment lines and blank lines)
CMMNTS	Number of lines of comments
MODULES	Number of unique procedures and unique functions
LENGTHN	Sum of the number of operators and operands $N = N_1 + N_2$
ESTN	Estimated program length is calculated and differs from N, the program length from direct observation $n_1 * \log_2 n_1 + n_2 * \log_2 n_2$
IMPLEVEL	Estimated program level or level of implementation contributes to the level of understanding and effort required to write a computer program $(2 * n_2)/(n_1 * N_2)$
VOLUME	Program volume - the size of any implementation of any algorithm $N * \log_2 n$

TABLE 4.2 (continued)

VARIABLES USED TO DEFINE THE SIZE CATEGORY

VOCAB	Sum of number of unique operators and unique operands $n = n_1 + n_2$

increased system quality if the size of the system developed using CASE technology is significantly less than the size of the system developed using non-CASE technology.

Eleven variables were used to define complexity. The number of lines of code (LOC), the number of procedures and functions invoked (CALLS), the number of iterations statements (LOOPS), the number of selections statements (SELECTS) and the total number of separate blocks within the code (BLOCKS) were counted in the final systems. LOC, BLOCKS and CALLS are direct indicators of length; the longer the project the greater the potential for complexity. LOOPS and SELECTS are indicators of the number of decisions made to accomplish the task; the more decisions and iterations, the greater the complexity of the system. Halstead's number of unique operators (n_2) and number of unique operands (n_1) are also directly

counted in the final project. Halstead's basic metrics, n_1 and n_2, are also two of Elshoff's set of complexity measures, the other two being data difficulty (DATADIFF) and length (LENGTHN). Gordon's CLARITY and Halstead's EFFORT are calculated from values counted in the final system. CLARITY is the amount of effort required to understand a computer program and EFFORT is the amount of mental activity necessary to convert an algorithm to a computer program; a more complex program should require more effort to understand and code. Elshoff's data difficulty (DATADIFF) and difficulty (DIFFICULTY) were also calculated from values counted in the final system. DATADIFF is the average number of variables, constants or literals; the more variables are used, the more chance for additional complexity. DIFFICULTY measures the number of errors in a program due to problems in understanding the program; a less complex program should have fewer errors in comprehension and therefore a lower level of DIFFICULTY. Table 4.3 lists and defines the eleven variables that were used to form the complexity category.

TABLE 4.3

VARIABLES USED TO DEFINE THE COMPLEXITY CATEGORY

NAME	DESCRIPTION/DEFINITION
LOC	Number of lines of code (excluding comment lines and blank lines)
n_2	Number of unique operands - the sum of the number of unique constants (CONST), unique variables (VARBL) and unique literals (LITERALS)
CALLS	Number of procedures (FEXTPRC) and functions (FEXTFN) invoked
LOOPS	Number of REPEAT, WHILE and FOR loops used
SELECTS	Number of CASE, IF and ELSE statements used
BLOCKS	The total number of PROCEDURE, FUNCTION, BEGIN, IF, WHILE, CASE, FOR, REPEAT statements
n_1	Number of unique operators - the sum of the unique number of iteration statements (REPEAT, WHILE, FOR), selection statements (CASE, IF, THEN, ELSE), symbols (mathematical and Boolean), delimiters, sets, records,

TABLE 4.3 (continued)

VARIABLES USED TO DEFINE THE COMPLEXITY CATEGORY

	read statements, write statements, function and procedure calls, the number of begin and end statements, and the number of open and close statements.
CLARITY	The amount of mental effort required to comprehend a computer program $\frac{((n_1 * \log_2 n_1 + n_2 * \log_2 n_2) * \log_2 n)}{(N * \log_2 n)}$ $(n = n_1 + n_2;\ N = N_1 + N_2)$ n is the vocabulary of a computer program N is the length (amount of source code) of a computer program
EFFORT	The mental activity required to reduce an algorithm to an actual computer program $\frac{((n_1 + n_2) * \log_2 n)}{(2 * n_2)/(n_1 * N_2)}$
DATADIFF	The average number of variable appearances N_2/n_2

TABLE 4.3 (continued)

VARIABLES USED TO DEFINE THE COMPLEXITY CATEGORY

DIFFICULTY	One half of the product of the unique operators and the data difficulty. Corresponds to the number of errors in a program due to the level of effort required to understand the program $(n_1 * DATADIFF)/2$

In addition to evaluating each of the individual variables, a combined complexity category was tested for significance. The eleven complexity variables were converted to z scores. Discriminant analysis (SPSSX) summed the eleven transformed complexity variables and calculated the significance for the complexity category. In order to reject the first operational hypothesis H_0 for B1, the values for all or most of the complexity variables should be significantly different between the treatment group (CASE) and the control group (non-CASE). If the systems have a given functionality, we can claim increased system quality if the complexity of the system developed using CASE technology is significantly less than the complexity of the system developed using non-CASE technology.

The second operational hypothesis H_0 for B2 was tested using a level of system completeness. As described above, a system was more complete if it met a greater number of the requirements than another system. In order to reject the second operational hypothesis H_0 for B2 for system quality, the levels of completeness of the CASE developed systems and the non-CASE developed systems should be significantly different. We can claim increased quality if the level of completeness of the CASE developed systems were higher than the level of completeness of the non-Case developed systems.

If the null hypotheses can be rejected, then the research will support the original beliefs. The above hypotheses were tested at a 0.10 level using statistical analyses to compare the control group (non-CASE) and the treatment group (CASE).

BELIEF 1

USE OF CASE TECHNOLOGIES INCREASES
THE PRODUCTIVITY OF THE PROGRAMMER

(A)H_0: THERE IS NO DIFFERENCE IN PROGRAMMER PRODUCTIVITY

(A1)H_0: THERE IS NO DIFFERENCE IN TIME

(A2)H_0: THERE IS NO DIFFERENCE IN THE LEVEL OF COMPLETENESS (SAME/LESS TIME)

AND

TIME

TIME COMPLETENESS

THE TIME CATEGORY IS DEFINED BY: GCOMPL; GRUNS, GLINKS, TOTTIME, TOTDES, TOTCOD, GLOGON, GTIME

COMPLETENESS IS DEFINED BY: 24 REQUIREMENTS IN SPECIFICATIONS

FIGURE 4.1 BELIEF 1 AND SUPPORTING HYPOTHESES

BELIEF 2

USE OF CASE TECHNOLOGIES INCREASES
THE QUALITY OF THE SYSTEM/PROGRAM BEING DEVELOPED

(B)H_0: THERE IS NO DIFFERENCE IN THE QUALITY OF THE SYSTEM

(B1)H_0: THERE IS NO DIFFERENCE IN COMPLEXITY

(B2)H_0: THERE IS NO DIFFERENCE IN THE LEVEL OF COMPLETENESS

SIZE COMPLEXITY COMPLETENESS

THE SIZE CATEGORY IS DEFINED BY: LOC, CMMNTS, MODULES, LENGTHN, EXTN, IMPLEVEL, VOLUME, VOCAB

THE COMPLEXITY CATEGORY IS DEFINED BY: LOC, EFFORT, CALLS, LOOPS, BLOCKS, SELECTS, CLARITY, N_1, N_2, DATA DIFFICULTY

COMPLETENESS IS DEFINED BY: 24 REQUIREMENTS IN SPECIFICATIONS

FIGURE 4.2 BELIEF 2 AND SUPPORTING HYPOTHESES

CHAPTER 5

DATA ANALYSIS

This chapter presents the analysis of the data collected during the experiment. Additional information about the methods used and the model for the experiment may be found in Chapters 3 and 4 respectively. Appendices F through H contain copies of the SPSSX statements and relevant output.

THE HYPOTHESES

Increased productivity can be defined as: an improved product is produced in the same amount of time or the same product is produced in less time. Increased system quality can be defined as: a more complete, consistent and accurate project with more complete, consistent and accurate documentation; there are fewer defects in the final system.

Given these two definitions, the following two beliefs have prompted this research:

Belief 1: The use of CASE technologies increases the productivity of the programmer.

Belief 2: Use of CASE technologies increases the quality of the system/program being developed.

The hypotheses that test these beliefs are defined in Chapter 4. If the null hypotheses can be rejected, then the research will support the original beliefs.

The hypotheses were tested at a 0.10 level using statistical analysis to compare the control group and the treatment group. Data were collected about both the process and the product. The data about the process consisted of measures of time; both self-reported and automatically collected during the coding process. The data about the products were evaluated using size, complexity and completeness variables.

SAMPLE SIZE

There were seven teams in the control group (non-CASE) and four teams in the treatment group (CASE). The sample size was determined by the number of students enrolled in each quarter. The small sample size limited the statistical analysis; more significant p values might have been obtained with a larger sample size.

TYPE OF ANALYSIS

Discriminant analysis is a statistical technique that can be used to classify subjects into groups based on a series of variables and is often used to determine whether a single variable maximizes the difference between two groups. If a single variable determines the difference between the groups, that variable is removed from the analysis and the rest of the variables are re-evaluated. This process is repeated until none of the remaining variables maximize the separation between the two groups.

Discriminant analysis (SPSSX) used twenty-six variables

for each team; seven teams from the control group (non-CASE) and four teams from the treatment group (CASE). The variables were grouped to form three categories: time, size and complexity. Since none of the variables maximized the difference between the two groups, the analysis was a two group univariate test. The F values were calculated and the levels of significance (p values) were then calculated for each of the individual variables. The small sample size indicated that t tests were an appropriate measure of the difference between the means of the two groups. The statistic, t^2 is equal to the F value calculated with the discriminant analysis (SPSSX) and the p values obtained were used.

The systems were also rated on the level of completeness of twenty-four logical functions (see Chapter 3 for complete details). A 0-1-2 scale was used for the rating. There were 24 requirements and a totally complete system, one that completely met all the requirements for all the functions had a rating of 48.

RESULTS OF DATA ANALYSIS

The levels of significance (p values) are reported for each of the variables within the three categories: time,

size and complexity. The variables were combined within the three categories and an overall category level of significance was calculated and reported. Then the results of the evaluation of the level of completeness are described.

DATA ANALYSIS - TIME CATEGORY

The self-reported time for both the design phase (TOTDES) and the coding phase (TOTCOD) is the amount of time that the students described in their progress reports, programmers' logs and minutes of meetings. The reported individual times and group times spent for each of the two project phases were tallied. Data were automatically collected during the coding phase. For each subject, every logon (GLOGON), compile (GCOMPL), link (GLINKS), run (GRUNS) and logoff was recorded. The logon and logoff times were used to calculate the amount of time (GTIME) spent on the VMS operating system. If a student did not logoff correctly or was logged off by the system, the amount of time was calculated using the last time recorded before the next chronological logon. For complete definitions of the variables, see Chapter 4.

The levels of significance (p values) for the difference in means and the means for each of the eight variables for the time category are listed in Table 5.1, and the SPSSX code and relevant results are in Appendix F.

TABLE 5.1

LEVEL OF SIGNIFICANCE (P VALUES) AND MEANS

TIME VARIABLES (ALL TEAMS)

NAME	P VALUES	MEANS CONTROL GROUP	MEANS TREATMENT GROUP
TOTDES	0.1473	81.71	50.00
TOTCOD	0.1552	122.71	23.50
TOTTME	0.1091	204.43	73.50
GCOMPL	0.0589	3912.71	1652.50
GLINKS	0.0190	1057.86	559.75
GRUNS	0.0174	1023.14	536.25
GTIME	0.0326	210.86	164.00
GLOGON	0.1164	220.29	165.00

The differences in the number of runs and links are significant at a 0.05 level. The amount of time spent logged on to the VMS operating system was significant at the 0.05 level and the number of compiles was significant

at the 0.10 level. The treatment group (CASE) spent less time on the VMS operating system, and compiled, linked, and ran their system fewer times than the control group (non-CASE). The other four variables, GLOGON, TOTCODE, TOTDES and TOTTME were very close to the 0.10 level of significance. Therefore, the CASE group used less time to code the system than the non-CASE group.

However, there was one student who did not always follow directions and at times managed to avoid the program that counted the number of links and runs. The portion of the program that counted the number of compiles, logons and logoffs could not be circumvented. The discriminant analysis was run without the team with the student who circumvented the counting program and the levels of significance and means for the number of links and runs are listed in Table 5.2, and the SPSSX code and relevant results are in Appendix G.

The number of links and the number of runs were no longer significant at the 0.05 level, but were still significant at the 0.10 level and do not affect the analysis. The CASE group used a fewer number of links and runs than the non-CASE group.

TABLE 5.2

LEVEL OF SIGNIFICANCE (P VALUES) AND MEANS

NUMBER OF LINKS AND RUNS - WITHOUT TEAM 3

(TREATMENT GROUP)

NAME	P VALUES	MEANS CONTROL GROUP	MEANS TREATMENT GROUP
GLINKS	0.0585	1057.86	717.00
GRUNS	0.0534	1023.14	686.33

The amount of time spent by the treatment group (CASE users) was significantly less than the amount of time spent by the control group (non-CASE users). Therefore, the null hypothesis that there is no difference in the amount of time used to develop the system was rejected. The amount of time spent by CASE users was less than the amount of time spent by non-CASE users to develop the system.

DATA ANALYSIS - COMPLEXITY CATEGORY

Although size is an additional indicator of the complexity of a system, size measures will be discussed separately from complexity measures (see Chapter 4). There are

eleven variables used to indicate the complexity of the final system. The number of executable lines of code (LOC), the number of iteration statements (LOOPS), the number of select statements (SELECTS), the number of procedure or function calls (CALLS), the number of unique operators (n2), the number of unique operands (n1) and the number of blocks (Elshoff) of code (BLOCKS) were extracted from the projects' Pascal code. Gordon's clarity (CLARITY), Halstead's effort (EFFORT), Elshoff's data difficulty (DATADIFF) and Elshoff's difficulty (DIFFICULTY) are calculated variables, depending upon two or more of the measures accumulated from the actual Pascal code. For complete definitions of the variables, see Chapter 4.

Table 5.3 lists the levels of significance (p values) for the difference in means and the means for each of the eleven variables for the complexity category, and the SPSSX code and relevant results are in Appendix F.

None of the differences in the complexity variables were significant. Therefore, the null hypothesis that there is no difference in the complexities of the two systems could not be rejected. The complexity of the system developed by CASE users could not be proved different from the complexity of the system developed by non-CASE users.

TABLE 5.3

LEVEL OF SIGNIFICANCE (P VALUES) AND MEANS

COMPLEXITY VARIABLES (ALL TEAMS)

NAME	P VALUES	MEANS CONTROL GROUP	MEANS TREATMENT GROUP
LOC	0.3296	1751.00	1437.50
n2	0.7860	181.29	165.50
CALLS	0.3671	69.15	45.50
LOOPS	0.8501	61.57	56.50
SELECTS	0.2113	181.86	112.00
n1	0.4798	34.14	33.25
BLOCKS	0.2909	311.86	211.75
CLARITY	0.4025	2078688.57	1413530.00
EFFORT	0.2119	6401098.86	3848281.75
DATADIFF	0.2304	10.63	7.93
DIFFICULTY	0.1809	180.47	132.29

DATA ANALYSIS - SIZE CATEGORY

The number of modules (MODULES), the number of executable lines of code (LOC) and the number of comments (CMMNTS)

are extracted from the projects' Pascal code. The length of the project (LENGTHN), estimated length (ESTN), implementation or program level (IMPLEVEL), volume (VOLUME) and vocabulary (VOCAB) are calculated from different measures that were extracted from the code. These all are Halstead's metrics.

The levels of significance (p values) for the difference in means for each of the eight variables for the size category are listed in Table 5.4, and the SPSSX code and relevant results are in Appendix F.

None of the differences in the size variables were significant. Therefore, the null hypothesis that there is no difference in the sizes of the two systems could not be rejected. The size of the system developed by CASE users could not be proved different from the size of the system developed by non-CASE users.

TABLE 5.4

LEVEL OF SIGNIFICANCE (P VALUES)

SIZE VARIABLES (ALL TEAMS)

NAME	P VALUES	MEANS CONTROL GROUP	MEANS TREATMENT GROUP
LOC	0.3296	1751.00	1437.50
CMMNTS	0.7259	188.15	221.00
MODULES	0.2421	34.00	25.75
LENGTHN	0.4608	4655.71	3694.75
ESTN	0.7508	1567.71	1398.25
IMPLEVEL	0.1878	0.006	0.008
VOLUME	0.5016	36553.71	28439.25
VOCAB	0.7767	215.43	198.75

DATA ANALYSIS - SIGNIFICANCE OF CATEGORIES (COMBINING VARIABLES WITHIN CATEGORY)

The values for each of the variables were transformed into z scores. These z scores for each category were summed using Discriminant Analysis procedure (SPSSX) and Table 5.5 lists the significance levels (p values) for each

category. The SPSSX code and relevant results are in Appendix H.

TABLE 5.5

LEVEL OF SIGNIFICANCE (P VALUES)

VARIABLES COMBINED INTO CATEGORIES (ALL TEAMS)

CATEGORY	P VALUES
COMPLEXITY	0.2110
SIZE	0.6532
TIME	0.0050

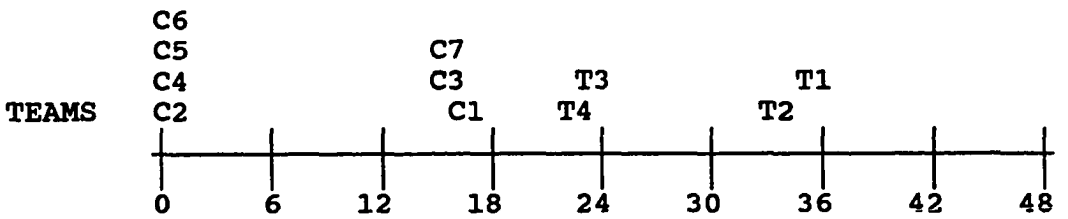
The differences in the summed variables for the time category were significant. The differences in both the combined variables for the size category and the combined variables for the complexity category were not significant. Since some of the complexity measures were nearly significant, but the combined measure was not significant, it might be concluded that the selected complexity measures measure different aspects of the program which probably are not related to each other. None of the individual size measures were significant and neither was the combined measure. These results should not be too surprising since all the systems were designed and coded to meet the same requirements. Almost all of

the time measures were significant and the combined measure was also significant. Less time was spent coding the CASE produced design; therefore, less computer resources and human resources were used.

DATA ANALYSIS - COMPLETENESS OF THE SYSTEM

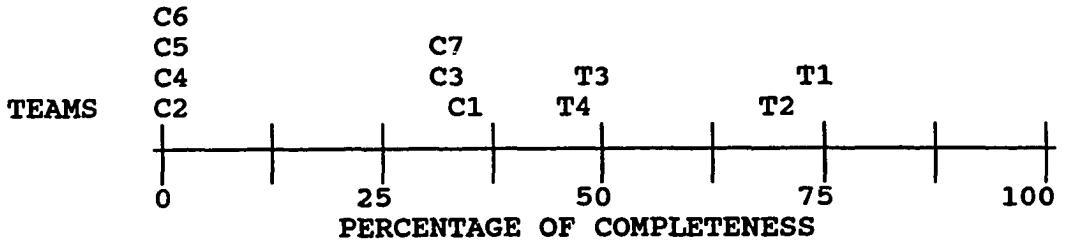
A system is complete if it meets the requirements. The systems were rated on the level of completeness of 24 logical functions (Appendix E) that were specified in the requirements. A "0 - 1 - 2" scale was used for the rating. A system was rated a "0" if it is totally inoperable for the function being tested. A rating of "1" indicated that the system partially fulfilled the requirements and a rating of "2" signified that the system completely fulfilled the requirements. These ratings were summed. The requirements were presented to the students as equally important. A totally complete system, one that completely met all the requirements for all the functions has a rating of 48. Figure 5.1 indicates the level of completeness for the eleven projects; Figure 5.2 the percentage of completeness. The four projects, all from the control group, that received a '0' rating for all 24 specifications had run time errors caused by either a

stack dump error or an access violation error; none of the projects had compile errors. The three remaining projects from the control group were each approximately 33% complete. The four treatment group projects ranged from 44% to 75% complete. The documentation, users' manuals and programmers' manuals, were of little or no help for the implementation of any of the projects and no conclusions about the system could be drawn from them.



SUM OF RATING ON THE 24 SPECIFICATIONS
('T' indicates the treatment group (CASE) and 'C'indicates the control group (non-CASE). The numbers following the 'T' or 'C' indicate the team within the group.)

FIGURE 5.1
LEVEL OF COMPLETENESS



SUM OF RATINGS ON THE 24 SPECIFICATIONS

PERCENTAGE OF 24 SPECIFICATIONS COMPLETE
WITH A TOTAL OF 48 BEING PERFECTLY COMPLETE
('T' indicates the treatment group (CASE) and 'C'indicates the control group (non-CASE). The numbers following the 'T' or 'C' indicate the team within the group.)

FIGURE 5.2
PERCENTAGE OF COMPLETENESS

The level of significance (p value) was calculated using the mean ratings for both groups. Then the level of significance (p value) was calculated omitting those teams that had run time errors. Both values were significant at the 0.05 level.

TABLE 5.6

SIGNIFICANCE (P VALUES) FOR SYSTEM COMPLETENESS

P VALUE (ALL TEAMS)	P VALUE (WITHOUT 4 TEAMS WITH RUN-TIME ERRORS)
0.002	0.045

The differences in the level of completeness between the treatment group and the control group were significant. Therefore, the null hypothesis that there is no difference in the completeness of the two systems could be rejected. The system developed by CASE users was more complete than the system developed by non-CASE users.

CONCLUSIONS

The null hypotheses that there is no difference between programmer productivity and there is no difference between the quality of a system produced using CASE technologies and the same system produced not using CASE technologies were both rejected.

From the above analysis, the null hypothesis concerning a difference in the size of the final systems has not been rejected. Therefore, it can be said the sizes of the systems cannot be shown to be different and that conclusion is to be expected since the project was designed and coded from the same requirements.

The null hypothesis that there is no significant difference in the time required to code the system was rejected. The

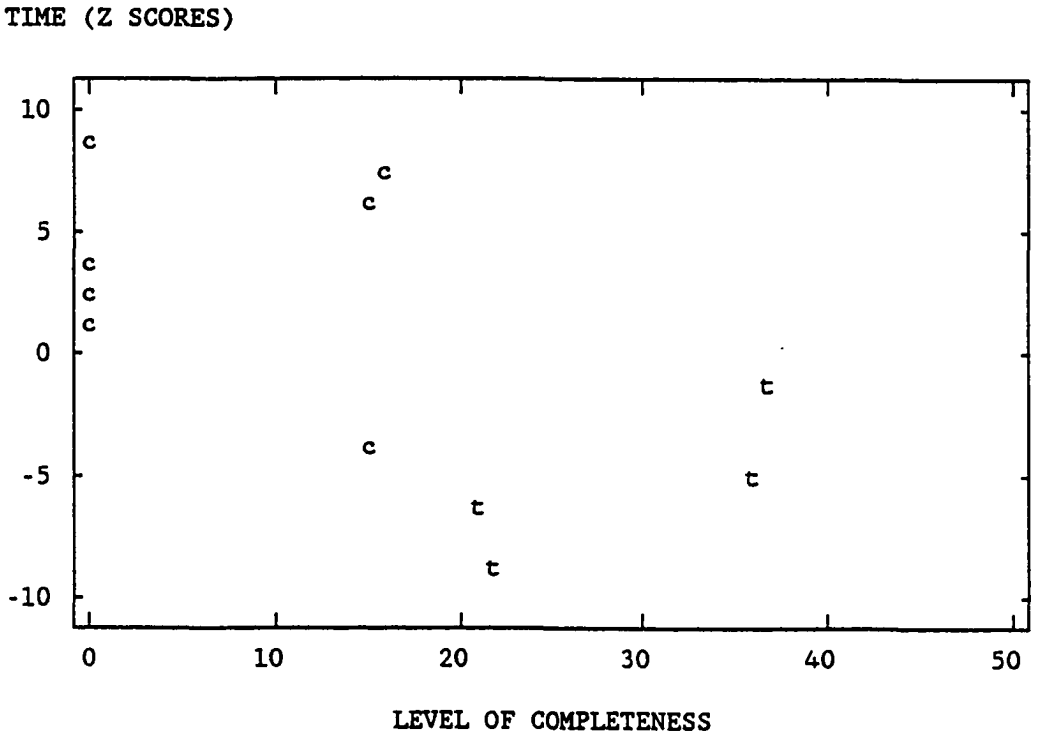
treatment group, CASE, used less time than the control group, non-CASE. Since productivity can be defined as producing the same product in a less amount of time, it was concluded that the productivity of the teams was increased by the use of CASE technologies for the design phase.

The level of completeness of the products between the control group and the treatment group was different. The treatment group (CASE users) produced a more complete product. Since the treatment group also used less time to produce the product, we concluded that a more complete product was produced in less amount of time using CASE technology.

The z values for the time variables for each team were summed. In Figure 5.3, the sum of the z scores for the time variables was plotted against the level of completeness for each team.

Complexity and completeness measure the quality of the final system. A difference in the complexity of the final systems was not rejected, however there was a difference in the level of completeness of the final systems. The treatment group (CASE) produced more complete systems. Although, the differences in the complexity of the systems could not be

rejected, the level of product completeness increased. There was an increase in one aspect of the quality of the CASE developed system over the non-CASE developed system. The CASE developed systems were better able to meet the specifications than the non-CASE developed systems.



SUMMED Z SCORES FOR TIME VARIABLES
BY
LEVELS OF COMPLETENESS
('T' indicates the treatment group (CASE) and 'C' indicates the control group (non-CASE). The numbers following the 'T' or 'C' indicate the team within the group.)

FIGURE 5.3
TIME BY COMPLETENESS

SUMMARY

The null hypothesis, there is no difference in the productivity of the programmer who uses CASE technologies and the productivity of the programmer who does not use CASE technologies, was rejected. The productivity of the programmer increased when CASE technologies were used. The sizes of the systems were not significantly different, however, more complete systems were produced in less time by the treatment group (CASE) than the control group (non-CASE).

The null hypothesis, there is no difference in the quality of the system/program developed using CASE technologies and the quality of the system/program developed without CASE technologies, was rejected for the completeness aspect of system/program quality. However, this hypothesis could not be rejected for the complexity aspect of system/program quality. The quality of the product/system with respect to completeness increased when CASE technologies were used. The more the finished system fulfilled the requirements, the higher the quality of the system. There was not a significant difference in complexity between CASE developed

and non-CASE developed systems. If quality was judged solely on that one aspect no conclusions could be drawn with respect to quality. This hypothesis was partially supported.

Although the sample sizes in this research were small, the productivity of the programmer and the quality of the systems improved when CASE technologies were used. The increase in productivity was shown because more complete products were produced in a less amount of time by Case users. The increase in one aspect of the quality of the systems produced by the CASE users was shown because the level of completeness of the CASE produced systems was greater than the non-CASE produced systems. The analysis supported the original research beliefs about programmer productivity and system quality.

CHAPTER 6

CONCLUSION

The main purpose of this research was to examine the impact of Computer-Aided Software Engineering (CASE) on programmer productivity and system quality. Was programmer productivity increased when the system was designed using CASE technologies? Was system quality improved when the system was designed using CASE technologies? An additional objective for this research was to establish software metrics that could be used to evaluate programmer productivity and system quality. This chapter discusses the findings of the research and the extent to which the goals were fulfilled. There is a section on the impact this research might have on MIS managers. The chapter concludes with some ideas for future research.

This study makes three contributions to the study of software development. First, to our knowledge, it is the first controlled experiment investigating CASE tools. The same software system was developed with and without CASE

technologies. Previous studies have involved CASE in commercial settings and the same system is never developed twice in a commercial setting.

Second, several metrics were identified that can be used to identify and evaluate programmer productivity. However, automated production performance monitors in a commercial environment may alienate the system analysts/programmers and any attempt to collect data automatically must be implemented very carefully. The number of iterations during different phases of the system life cycle could impact time metrics. Again, this study was limited by the ten-week quarter and coding was limited to the second five weeks. The subjects may be able to use more discretion than commercial programmers with respect to the amount of time spent on the project.

The third contribution of this research is the quantitative measures to the claims of increased programmer productivity and system quality being made by CASE vendors and others. In this study, programmer productivity increased when CASE technologies were used to design a software system. Also in this study, the quality of the systems improved; more complete systems were developed by the teams that used CASE technologies for

system design. The designs developed with CASE technologies were better than the designs developed without CASE technologies. They fulfilled the requirements specifications, and the finished system was more complete. The designs developed with CASE technologies were easier to understand. Programmers used less time to code the CASE developed design than the non-CASE developed design.

SUMMARY OF FINDINGS

The same task, coding a pretty printer, was accomplished in less time, using less computer resources by the treatment group (CASE). The p values obtained from the t tests on the time variables indicated there was a significant difference in the times required to code the pretty printer between the control group (non-CASE) and treatment group (CASE). The number of logons (GLOGON) and the amount of time (GTIME) spent on the VMS operating system for the control group (non-CASE) averaged almost one and a half times the average for the treatment group (CASE). The control group used more computer resources, with twice as many compiles (GCOMPL) and almost twice as many links (GLINKS) and runs (GRUNS) as the treatment group. For exact figures, see Table 5.1.

The significant times only include the time and computer resources used while coding on the VMS operating system. It does not include time and computer resources consumed while designing with the CASE tool on the PC workstation. Usually when less time is spent coding the system, more time is spent on the design. In this study the average self-reported design time for the CASE group (50 hours) was less than the average self-reported design time for the non-CASE group (80 hours). There is further discussion of the self-reported times in this chapter. Programmer productivity during the coding phase increased when CASE technologies were used.

There was also a significant difference in the levels of completeness of the systems produced using CASE and those not using CASE. The treatment group (CASE) used less time to produce a more complete program than the control group (non-CASE). The CASE-developed systems met the specifications better than the non-CASE developed systems; therefore, the CASE-developed systems were more complete than the non-CASE developed systems. If the non-CASE developed systems had been more complete than the CASE developed systems, the claim for increased programmer productivity based solely on less coding time would not be valid. We can, however, state that programmer

productivity was increased when the system was designed using CASE technologies; our original belief about increased programmer productivity was supported.

No statistically supported conclusions could be made from the t tests on the complexity variables. There was no significant difference in the complexity of the systems developed using CASE technologies and those developed without CASE technologies. However, the control group (non-CASE) developed systems that contained a greater number of lines of code (LOC), operators (n_1), operands (n_2), select statements (SELECTS), iteration statements (LOOPS), PROCEDURE or FUNCTION calls (CALLS), and blocks of code (BLOCKS) than systems developed by the treatment group (CASE). The level of program clarity (CLARITY), the level of program difficulty (DIFFICULTY), and the amount of effort (EFFORT) required to reduce an algorithm to implementation in a language were higher for the non-CASE developed system. Data difficulty (DATADIFF), the ratio of the total number of operands to the number of unique operands, was higher for the non-CASE group. The means for all of these variables were uniformly (but not significantly) higher for the systems developed by the non-CASE group. For exact figures, see Table 5.3.

There was a significant difference in the levels of completeness of the systems. Since increased system quality was defined as being less complex and more complete, it can be concluded that system quality was increased with respect to the level of system completeness, but that more research is needed to make any statements about system complexity. It seems likely that had the non-CASE groups produced systems as complete as the CASE groups, their software would have been more complex. Our original belief about improved system quality was partially supported by the significant difference in completeness.

METRICS

This is the first time that Software Science metrics have been used to evaluate CASE technologies in a controlled environment. Other studies have automatically collected data about the process, but none have also evaluated the product using complexity metrics. One of the goals of this research was to uncover some metrics that could be used to evaluate either programmer productivity or system quality.

The self-reported times are often inconsistent and, in some instances, incomplete. While self-reported times are the least costly to collect, this data is often not accurate. The amount of self-reported time spent on coding should have reflected all the time spent both logged on and off the system during the coding phase. The self-reported times were found to underestimate the amount of time actually spent on the VMS operating system and, therefore, cannot be considered accurate. The self-reported times during the design phase cannot be verified with any data collected automatically. There is reason to believe that the design self-reported times may be more reliable than the coding self-reported times. The design times were collected at the beginning of the quarter and the students appeared to be more conscientious during that period. Second, the students knew that data was automatically being collected during the coding phase and may have felt that their reports were redundant.

The data collected automatically was complete, except for the one student who circumvented the program that was accumulating the data. The number of logons, compiles, links and runs are indicators of what activities were performed while the subjects were logged onto the VMS operating system; the amount of time spent on the VMS

operating system was used as an indicator of the time spent coding the system. These measures, however, did not take into account any of the time that was used to code while not logged onto the system. Time not logged on the system could have been used to write code or debug, and such time use should have been included in the self-reported time to code the system.

Results from this research on the complexity measures were inconclusive; further research is required. The metrics appeared to be measuring similar aspects of complexity, and overall the variables selected were consistent. More studies similar to Elshoff (1984) need to be conducted in order to determine the most significant complexity measures and to determine which metrics measure similar components of complexity. The complexity aspect of quality of a system is difficult to define, evaluate and measure.

MANAGERIAL IMPLICATIONS

Information technology managers should be encouraged in their quest for increased programmer productivity. A major component of the software crisis is the inability to

measure, estimate, and improve programmer productivity. This study indicates that use of CASE tools could improve programmer productivity.

The control group (non-CASE) spent an average of 210.86 hours on the VMS operating system during the coding phase, and the treatment group (CASE) spent an average of 164.00 hours. The treatment group saved 22.23% of the coding time. A very comprehensive CASE workstation with Excelerator costs up to $15,000. The salary and benefits for novice programmers, recent Information Systems graduates, is approximately $36,000 yearly. Therefore, within 3 years, the savings on programmers' salaries will pay for the investment in CASE. These very conservative productivity results should be useful to MIS managers who must decide (or convince their parent organization) that CASE technologies may increase programmer productivity and help alleviate the "software crisis." CASE technologies are cost effective on the basis of programmer productivity.

Many of the students currently majoring in Information Systems will be the applications-systems analysts of tomorrow. Therefore, the results of this study may be generalized to the entire population of professional

system analysts. It should be noted that the students in the Autumn 1989 quarter were novices in CASE and in software engineering. Since they were not experienced systems analysts or programmers, they might have been more receptive to the new CASE technology than system analysts or programmers with years of design or programming experience without CASE technology. Programmers with several years of experience are often reluctant to change their style of programming or designing, often claiming that their work is creative and that automation of the process will stifle creativity. One of the researchers involved with this study predicted less programmer productivity with CASE due to an expected long learning curve for novices. This did not occur in this study. Other studies involving students learning new technologies (Mirsa 1989; Mynatt 1989) also report a learning curve far shorter than expected or experienced in a commercial setting. The learning period might be longer for the entire population of expert analysts than for the subjects in this study. Gains in this study should be attainable in a commercial setting but may take longer because students are more adaptable. On the other hand, professionals already know how to plan well, students do not. It is possible that Excelerator provided a

disciplined environment for students that would not be needed by professionals.

Nonetheless, information technology managers should be mindful when applying these findings to a commercial setting that the subjects were novices with CASE technologies, the sample size was small and the project was not a true "programming in the large" project.

FUTURE RESEARCH

Only one CASE tool, Excelerator, was used and the experiment should be repeated using different CASE products. Excelerator, a product of Index Technology Corporation, was the first IBM PC based CASE product and currently is the most widely used microcomputer CASE tool. Excelerator is an upper CASE tool used to aid in the requirements analysis and design phases of the system development life cycle and is built around an integrated data dictionary. Excelerator supports several different structured design methodologies; the subjects in this research used the Gane/Sarson data flow diagrams, the Yourdon/Constantine structure charts, and the integrated data dictionary. There is extensive verification checking

against the structured methodologies' rules. Can the results of this research be generalized to other upper CASE tools, or lower CASE tools (code generators)? It appears that similar results would be obtained if a different upper CASE tool was used and the study repeated. Use of a lower CASE tool would require re-definition of data collection methods and evaluation criteria.

This study reported on the results from only one task. Future research and analysis is planned to include the data from additional projects and additional CASE tools.

There is a possibility of confounding in that the students were taught by the same instructor in two separate classes in two separate quarters. There may be a history effect in terms of the professional growth of the instructor. Moreover, the instructor may have unconsciously done a better job of answering questions and anticipating problems during the second quarter (treatment group - CASE). This possibility will be eliminated by repeating the study with the treatment group (CASE) system development first and the control group (non-CASE) system development second.

Future research also should include direct investigation of the design process and the design documentation. Data could be collected about the length of time required to design a project and the design itself could be evaluated (Haas 1988). Instead of self-reported times for the design, different types of data collection could be implemented, e.g., video taping or verbal protocol analysis. Additionally, the amount of time spent, and the activities performed on the Excelerator workstation could be automatically monitored in a manner similar to this research's monitoring on the VMS operating system during the coding phase. In this study the effects of the design were evaluated using the final software systems and the coding process. This research was an indirect measure of the design and further research should focus on the direct measurement of the design activities.

In some sense, the systems (products) delivered by the CASE and non-CASE teams were different. This is demonstrated by the different levels of completeness between the two systems. A future study might try to obtain identical project completion levels by using a less challenging task. This might cause a greater discrepancy in the time and complexity variables.

This study might be expanded to measure the effect of CASE on the maintenance phase of the software development life cycle. Since we currently have projects that were developed with and without CASE technologies, they could be used as systems that require maintenance and students enrolled in a future Software Engineering course would perform the required updating. Since the systems in this study implemented the same task with and without CASE, the same enhancement or maintenance could be performed a system from the control group and a system from the treatment group. The designs originally developed before the coding phase would be used to aid in the maintenance tasks. In order to utilize the same maintenance task, an identical level of completeness is required for both systems.

A field experiment to confirm the findings of this study would be useful. The effects of different CASE tools or combinations of CASE tools could be evaluated throughout the entire software development life cycle. Use of CASE technologies would no longer be the only variable studied during commercial software development. Several of the variables that could no longer be controlled are the type of task, programming language, operating system, stability of subjects with respect to numbers and turnovers, and the

background of the subjects. In spite of some of the control difficulties, the findings of such a study should be richer in detail and insight than the results of a controlled design experiment.

SUCCESSFUL RESEARCH

Was the research successful? This research accomplished some of its objectives and partially met others. In spite of the limited time, the small sample size, and subjects without prior CASE knowledge or experience, significant differences were found in both programmer/system analyst productivity and system quality. Although there were no significant differences between the complexity of the systems developed by the two groups of students, there were significant differences in the completeness of the systems. It is probable that if the level of completeness for the CASE and non-CASE systems was the same that there would have been differences in the complexity of the two systems. Future research needs to be done in the area of software quality and the relationship between completeness and complexity.

There were significant improvements when CASE tools were used during the design phase. The results should be useful for MIS managers considering the adoption of CASE tools. This study should be viewed as a beginning for establishing some metrics about the process and the product. More research is needed on both CASE and software metrics. Studies in commercial settings are difficult, but once some standards for evaluating the process and the product are established, this type of research should be conducted in a commercial environment.

BIBLIOGRAPHY

Acly, Ed (1988) "Looking Beyond CASE." IEEE Software, March 1988, 39-43.

Aranow, Eric (1988) "When is CASE The Right Choice?" Business Software Review, April 1988, Vol. 7, No. 5., 14-17.

Arthur, Lowell Jay (1983) Programmer Productivity: Myths, Methods and Murphology: A Guide For Managers, Analysts and Programmers. John Wiley & Sons, New York.

Attewell, Paul and Rule, James (1984) "Computing and Organizations: What We Know and What We don't know," Communications of the ACM, Vol 27, No. 32, December 1984, 1184-1192.

Bachman, Charlie, (1988) "A CASE for Reverse Engineering," Datamation, July 1, 1988, Vol. 34, No. 13, 49-56.

Basili, Victor R. and Reiter, Robert W. (1981) "A Controlled Experiment Quantitatively Comparing Software Development Approaches." IEEE Transactions on Software Engineering, Vol. SE-7, No.3, May 1981, 299-320.

Basili, Victor R., Selby, Richard W. and Hutchens, David H. (1986) "Experimentation in Software Engineering." IEEE Transactions on Software Engineering, Vol. SE-12. No. 7, July 1986, 733-743.

Beath, Cynthia Mathis (1988) "Some Problems in Generalizing from Information Systems Research." Presented at the TIMS/ORSA Joint National Meeting, April 25, 1988, Washington, D.C.

Behrens, Charles A. (1983) "Measuring the Productivity of Computer Systems Development Activities with Function Points." IEEE Transactions on Software Engineering, Vol. SE-9. No. 6, November 1983, 648-652.

Boar, Bernard H. (1985) Application Prototyping: A Project Management Perspective. American Management Association, New York.

Boehm, Barry W. (1973) "Software and Its Impact: A Quantitative Assessment," Datamation, Vol. 19, No. 5, May 1973, 48-59.

Boehm, Barry W., (1981) Software Engineering Economics, Prentice-Hall, Inc., Englewood Cliffs, New Jersey.

Boehm, Barry W. (1987) "Improving Software Productivity." IEEE Computer, September 1987, 43-57.

Boehm, Barry W., Gray, Terrance E. and Seewaldt, Thomas (1984a) "Prototyping Versus Specifying: A Mulitproject Experiment." IEEE Transactions on Software Engineering, Vol. SE-10, No. 5, May 1984, 290-303.

Boehm, B. W., Penedo, M. H., Stuckle, E. D., Williams, R. D., and Pyster, A. B. (1984b) "A Software Development Environment for Improving Productivity." Computer, Vol. 17, No. 6, June 1984, 30-44.

Burkhard, Donald L. (1989) "Implementing CASE Tools." Journal of Systems Management, May 1989, Vol. 40, No. 5, 20-25.

Cameron, Robert D. (1988) "An Abstract Pretty Printer." IEEE Software, Vol. 5, No. 6, November 1988, 61-67.

Card, David (1988) "Major Obstacles Hinder Successful Measurement." IEEE Software, Vol. 5, No. 6, November 1988, 82-86.

Card, David N., McGarry, Frank E. and Page, Gerald T. (1987) "Evaluating Software Engineering Technologies." IEEE Transactions on Software Engineering, Vol. SE-13, No. 7, July 1987, 845-851.

Carey, Jane M. and McLeod, Raymond (1988) "Use of System Development Methodology and Tools." Journal of Systems Management, March 1988, Vol. 39, No. 3, 30-35.

Chen, Minder, Numaker, Jay F. and Weber, E. Sue (1989) "Computer-Aided Software Engineering: Present Status and Future Directions," Database, Vol. 20, No. 1, Spring 1989, 7-13.

Chen, Peter P. (1976) "Entity-Relationship Model: Toward a Unified View of Data." ACM Transactions on Database Systems, Vol. 1, No.1, March, 1976, 9-36.

Chikofsky, Elliot J. (1988) "Software Technology People Can Really Use." IEEE Software, Vol. 5, No. 2, March 1988, 8-10.

Chikofsky, Elliot J. (1989) "Making CASE Pay Off." CIO, February 1989, Vol. 2, No. 5, 12-16.

Chikofsky, Elliot J. and Rubenstein, Burt L. (1988) "CASE: Reliability Engineering for Information Systems." IEEE Software, Vol. 5, No. 2, March 1988, 11-16.

Curtis, Bill (1983) "Software Metrics: Guest Editor's Introduction." IEEE Transactions on Software Engineering, Vol. SE-9, No. 6, November 1983, 637-638.

de la Torre, Jose. (1988) "Quality-assured Software in 4GL/CASE." Business Software Review, March 1988, Vol. 7, No. 3, 30-33.

DeMarco, Thomas (1979) Structured Analysis and System Specification. Prentice-Hall, New Jersey.

DeMarco, Thomas (1982) Controlling Software Projects, Yourdon Press, New York.

Eliot, Lance B. and Scacchi Walt (1986) "Towards a Knowledge-Based System Factory: Issues and Implementations." IEEE Expert, Vol. 1, No. 4, Winter 1986, 51-58.

Elshoff, James L. (1984) "Characteristic Program Complexity Measures." Proceedings of the 7th International Conference on Software Engineering, March 1984, Orlando, Florida, 288-293 IEEE Computer Society Press, Los Angeles, California.

Fairley, Richard E. (1985) Software Engineering Concepts. McGraw-Hill Company, New York.

Fersko-Weiss, Henry (1990) "CASE Tools for Designing Your Applications," PC Magazine, Vol. 9, No. 2, January 30, 1990, 213-251.

Fitzsimmons, Ann and Love, Tom (1978) "A Review And Evaluation of Software Science." Computing Surveys, Vol. 10, No. 1, March 1978, 3-18.

Flak, Howard (1989) "Software Vendors Serve Up Varied Palette for CASE Users." Computer Design, Vol. 28, No. 1, January 1, 1989, 70-80.

Frank, Werner (1988) "The Myth is Reborn." Software Magazine, Vo. 8, No. 8, August 1988, 8-10.

Frenkel, Karen A. (1985) "Toward Automating the Software-Development Cycle." Communications of the ACM, Vol. 28, No. 6, June 1985, 578-589.

Freeman, Peter (1983) "Fundamentals of Design." in Tutorial on Software Design Techniques, Fourth Edition, edited by Peter Freeman, IEEE Computer Society Press, Silver Spring, Maryland, 2-22.

Gannon, J. D. (1977) "An Experimental Evaluation of Data Type Conventions." Communications of ACM, Vol, 20, No. 8, August 1977, 584-595.

Gibson, Michael Lucas (1989) "The CASE Philosophy." BYTE. April 1989, 209-218.

Gilb, Thomas (1977) Software Metrics. Winthrop Publishers, Inc. Cambridge, Massachusetts.

Glass, R. L. (1982) "Modern Programming Practices: A Report from Industry", Prentice-Hall, Englewood Cliffs, New Jersey, 1982 as cited in Abdel-Hamid, Tarek K. (1988) "Understanding the "90% Syndrome" in Software Project Management: A Simulation-Based Case Study", The Journal of Systems and Software, Vol. 8, No. 4, August 1988, 319-330.

Gordon, Ronald D. (1979a) "Measuring Improvements in Program Clarity." IEEE Transactions on Software Engineering, SE-5, No.2, March 1979, 79-90.

Gordon, Ronald D. (1979b) "A Qualitative Justification for a Measure of Program Clarity." IEEE Transactions on Software Engineering, SE-5, No.2, March 1979, 121-128.

Grady, Robert B. (1987) "Measuring and Managing Software Maintenance." IEEE Software, Vol. 4, No. 5, September 1987, 35-45.

Gurbaxani, Vijay and Mendelson, Haim (1987) "Software and Harware in Data Processing Budgets." IEEE Transactions on Software Engineering, Vol. SE-13, No. 9, September 1987, 1010-1017.

Haas, David F. and Waguespack, Leslie J. (1989) "Sizing Assignments: A Contribution From Software Engineering to Computer Science Education." Proceedings of the Twentieth SIGCSE Technical Symposium on Computer Science Education, Louisville, Kentucky, February 23-25, 1989, eds. Barrett, Robert A. and Mansfield, Maynard J., 190-194, SIGCSE Bulletin, Vol. 21, No. 1, The Association for Computing Machinery, New York, New York.

Hall, William E. III and Zweben, Stuart H. (1986) "The Cloze Procedure and Software Comprehensibility Measurement," IEEE Transactions on Software Engineering, Vol. SE-12, No. 5, May 1986, 608-623.

Halstead, Maurice H. (1977) Elements of Software Science. Elsevier, New York.

Hanna, Mary Alice (1990) "Move Is On To Tie Vision To Information Systems," Software Magazine, Vol. 10, No. 1, January 1990, 39-45.

Hartog, Curt and Herbert, Martin (1985) Opinion Survey of MIS Managers: Key Issues," MIS Quarterly, Vol. 10, No. 4, December 1986, 351-361.

Hausen, Hans-Ludwig and Mullerburg, Monika (1981) "Conspectus of Software Engineering Environments." Proceedings of 5th International Conference on Software Engineering, 34-43.

Humphrey, Watts S. (1988) "Characterizing the Software Process: A Maturity Framework." IEEE Software, Vol. 5, No. 2, March 1988, 73-79.

Humphrey, Watts S. (1989) Managing The Software Process. Addison-Wesley Publishing Company, Reading, Massachusetts.

Index (1987). Excelerator. Cambridge, Mass: Index Technology Corporation.

Jackson, Ken (1988) "Providing For The Missing Steps." UNIX Review, Vol.6, No. 11, 55-63.

Jackson, Michael A. (1983) Systems Development. Prentice Hall, Inc. Englewood Cliffs, New Jersey.

Jones, Capers (1986) Programming Productivity. McGraw-Hill Book Company, New York.

Joyce, Daniel (1987) "An Identification and Investigation of Software Design Guidelines for Encapsulation Units," Doctoral Dissertation, Temple University 1987.

Knight, John C. and Leveson, Nancy G. (1986) "An Experimental Evaluation of The Assumption of Independence in Multiversion Programming." IEEE Transactions on Software Engineering, Vol. SE-12, No. 1, January 1986, 96-109.

Kwong, Arnold W. (1988) "A CASE of Culture Shock." Business Software Review, Vol. 7, No. 5, April 1988, 26-27.

Lee, Wayne (1975) Experimental Design and Analysis. W.H. Freeman and Company, San Francisico, California.

Levine, Harvey A. (1989) "Two Separate Worlds Moving Slowly Closer." Software Magazine, Vol. 9, No. 3, March 1989, 32-40.

Lewis, T. G. (1988) "Software and The Single Programmer." Dr. Dobbs Software Engineering Sourcebook, Winter 1988, 18-27.

Lientz, B. P. and Swanson, E. B. (1980) Software Maintenance Management, Addison-Wesley, Reading, Massachusetts.

Linger, Richard C., Mills, Harlan D. and Witt, Bernard I. (1979) Structured Programming: Theory and Practice. Addison-Wesley Publishing Company, Reading, Massachusetts.

Loh, Marcus and Nelson, R. Ryan, (1989) "Reaping CASE Harvests," Datamation, Vol. 35, No. 13, July 1, 1989, 31-36.

Mahmood, Mo A., (1987) "System Development Methods-A Comparative Investigation," MIS Quarterly, Vol. 11, No. 3, September 1987, 293-311.

Martin, Charles F. (1988a) "Getting CASE in Place." Business Software Review, Vol. 7, No. 5, April 1988, 20-25.

Martin, Charles F. (1988b) "Second-Generation CASE Tools: A Challenge to Vendors." IEEE Software, Vol. 5, No. 2, March 1988, 46-49.

Martin, James (1982) Application Development Without Programmers, Prentice Hall, Englewood Cliffs, New Jersey.

Martin, James and McClure, Carma (1988) Structured Techniques: The Basis for CASE. Prentice Hall, Englewood Cliffs, New Jersey.

McClure, Carma (1989) CASE is Software Automation. Prentice-Hall, Englewood Cliffs, New Jersey.

McWilliams, Gary (1989) "Integrated Computing Environments." Datamation, Vol. 35, No. 9, May 1, 1989, 18-21.

Messenheimer, Susan and Weiszmann, Carol (1988) "Quality Software Quest." Software Magazine, Vol. 8, No. 2, February 1988, 29-36.

Misra, Santosh K. and Subramanian, Venkat (1988) "An Assessemnt of CASE Technology for Software Design." Information and Management, Vol. 15, No. 4, November 1988, 213-228.

Myers, Glenford J. (1978) "A Controlled Experiment in Program Testing and Code Walkthroughs/Inspections", Communications of the ACM, Vol. 21, No. 9, September 1978, 760-768.

Mynatt, Barbee T. and Leventhall, Laura Marie (1989) "A CASE Primer for Computer Science Educators." Proceedings of the Twentieth SIGCSE Technical Symposium on Computer Science Education, Louisville, Kentucky, February 23-25, 1989, eds. Barrett, Robert A. and Mansfield, Maynard J., 122-126.

Necco, Charles R., Gordon, Carol L. and Tsai, Nancy W. (1987) "Systems Analysis and Design: Current Practices," MIS Quarterly, Vol. 11, No. 4, December 1987, 461-475.

Necco, Charles R., Tsai, Nancy W. and Holgeson Kreg W. (1989) "Current Usage of CASE Software." Journal of Systems Management, Vol. 40, No. 5, May 1989, 6-11.

Nejmeh, Brian A. (1988) "Designs on Case." UNIX Review, Vol. 6, No. 11, November 1988, 45-50.

Norman, Ronald, J. and Nunamaker, Jay F. (1988) "An Empirical Study of Information Systems Professionals' Productivity Perceptions of CASE Technology." Proceedings of the Ninth International Conference on Information Systems, Minneapolis, Minnesota, November 30-December 3, 1988, eds. DeGross, Janice I. and Olson, Margrethe H., 111-118.

Norman, Ronald, J. and Nunamaker, Jay F. (1989a) "CASE Productivity Perceptions of Software Engineering Professionals." Communications of the ACM, Vol. 32, no. 9, September 1989, 1102-1108.

Norman, Ronald J. and Nunamker, Jay F. (1989b) "Integrated Development Environment: Technological and Behavioral Productivity Perceptions," The 22nd Hawaii International Conference on System Sciences, Vol. II, ed. Shriver, Bruce D., January 3-6, 1989, 996-1003.

Oppen, Derek C. (1980) "Prettyprinting." ACM Transactions on Programming Languages and Systems, Vol. 2, No. 4, October 1980, 465-483.

Page-Jones, Meilir (1988) The Practical Guide to Structured Systems Design. Yourdon Press, Englewood Cliffs, New Jersey.

Patton (1986). Flow Charting II+ [Computer Progam]. (Version 2.40B). Patton & Patton Software Corp, San Jose, California.

Percy, Tony (1988) "What CASE can't do yet." Computerworld, Vol. XXII, No.25, June 20, 1988, 59-60.

Pressman, Roger S. (1982) Software Engineering: A Practitioner's Approach. McGraw-Hill Company, New York.

Pritsker, A. Alan B. (1984) Introduction to Simulation and Slam II. John Wiley & Sons, New York.

Ramamoorthy, C.V, Prakash, Atul, Tsai, Wei-Tek and Usunda, Yutaka (1984) "Software Engineering Problems and Perspectives," Computer, Vol. 17, No. 10, October 1984, 191-209.

Ramanathan, Jayashree and Sarkar, Soumitra (1988) "Providing Customized Assistance for Software Lifecycle Approaches." IEEE Transactions on Software Engineering, Vol. 14, No. 6, June 1988, 749-757.

Rinaldi, Damian (1989) "The CASE Way of Life; To Each His Own Method." Software Magazine, Vol. 9, No. 5, April 1, 1989, 33-42.

Rochester, Jack B. (1989) "Building More Flexible Systems." I/S Analyzer, Vol. 27, No. 10, October 1989, 1-12.

Rombach, H. Dieter (1987) "A Controlled Experiment on the Impact of Software Structure on Maintainability." IEEE Transactions on Software Engineering, Vol. SE-13, No. 3, March 1987, 344-354.

Rubin, Lisa F. (1983) "Syntax-Directed Pretty Printing - A First Step Towards a Syntax-Directed Editor." IEEE Transactions on Software Engineering, Vol. SE-9, No. 2, March 1983, 119-127.

Schindler, M. (1981) "1981 Technology Forecast-Software," Electronic Design, Vol 29, No. 1, January 1981, 190-199 as cited in Shemer, Itzhak (1987) "Systems Analysis: A Systemic Analysis of a Conceptual Model," Communications of the ACM, Vol. 30, No. 6, June 1987, 506-512.

Selby, Richard W., Basili, Victor R. and Baker, F. Terry (1987) "Cleanroom Software Development: An Empirical Evaluation," IEEE Transactions on Software Engineering, Vol. SE-13, No. 9, September 1987, 1027-1037.

Shemer, Itzhak (1987) "Systems Analysis: A Systemic Analysis of a Conceptual Model," Communications of the ACM, Vol. 30, No. 6, June 1987, 506-512.

Smith, David J. and Wood, Kenneth B (1987) Engineering Quality Software: A Review of Current Practices, Standards and Guidelines Including New Methods and Development Tools, Elsevier Applied Science, New York.

Stevens, W. P., Constantine, L. L., and Myers, G. J. (1974) "Structured Design." IBM Systems Journal, Vol. 13, No. 2, 115-139.

Stratland, Norman (1989) "Payoffs Down the Pike: A CASE Study." Datamation, Vol. 35, No. 7, April 1, 1989, 32-33,52.

Teichroew, Daniel and Hershey, Ernest A. (1977) "PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Infromation Processing Systems." IEEE Transactions on Software Engineering, Vol. SE-3, No. 1, January 1977, 41-48.

Topper, Andrew (1990), "Excelerator," PC Magazine, January 30, 1990, Vol. 9, No. 2, 224-230.

Turner, Ray (1984) Software Engineering Methodology. Reston Publishing Company, Inc., Reston, Virginia.

Voelcher, John (1988) "Automating Software: Proceed with Caution." IEEE Spectrum, Vol. 25, No. 7, July 1988, 25-27.

Wallace, Steve (1988) "Methodology: CASE's Critical Cornerstone." Business Software Review, Vol. 7, No. 5, April 1988, 17-20.

Warnier, Jean-Dominique (1981) Logical Construction of Systems. Van Nostrand Reinhold, New York, 11-38.

Wasserman, Anthony J. and Gutz, Steven (1982) "The Future of Programming," Communications of the ACM, Vol. 25, No. 3, March 1982, 196-206.

Weber, Herbert (1989) "From CASE to Software Factories." Datamation, Vol. 35, No. 7, April 1, 1989, 34-36,52.

Weyuker, Elaine J. (1988) "Evaluating Software Complexity Measures." IEEE Transactions on Software Engineering, Vol. SE-14, No. 9, September 1988, 1357-1365

Whitten, Jeffrey L. and Bentley, Lonnie D. (1987) Using Excelerator for Systems Analysis and Design, Times Mirror/Mosby College Publishing, St. Louis, Misssouri.

Yourdon, Edward (1988) "CASE Competition is All Over the World," Software Magazine, (International Edition) Vol. 8, No. 14, November 1988, 53-60.

Yourdon, Edward N. (1989a) "Software METRICS You can't control what you can't measure," American Programmer, Vol. 2, No 2, February 1989, 3-11.

Yourdon, Edward (1989b) Modern Structured Analysis, Yourdon Press, Englewood Cliffs, New Jersey.

Yourdon, Edward N. and Constantine, Larry L. (1979) Structured Design. Prentice-Hall, Englewood Cliffs, New Jersey.

APPENDIX A

PRETTY PRINTER SPECIFICATIONS

Software Engineering

22-485-322-001

Project Specifications

This project involves designing, coding, implementing, and testing a Pascal "pretty printer". A pretty printer is a software tool which formats programs (source code) without syntax errors into a format which is easy to read, understand, and maintain. You can assume that the programs which are run through the pretty printer have been compiled and contain no syntax errors.

A pretty printer can be used in many different ways. One possible use is in a large data processing/programming shop. Since each programmer has his/her own style of programming, the pretty printer can be used to standardize all of the code produced in the shop. In this manner, all of the code is easily understood and follows the same style guidelines to allow for easy maintenance in the future.

The pretty printer package should be user friendly. A user interface should be present. However an elaborate one is not necessary for the package.

The pretty printer should be able to perform the following tasks for each program run through the package:

1. Capitalize all reserved words. Consult a Pascal Reference Manual for a list of reserved words.

2. Alphabetize all declarations in the CONST, TYPE, and VAR sections of the program. All of the corresponding comment lines should also be moved correctly.

3. Alphabetize procedures and functions in the code by the procedure/function name. If the procedure/functions are physically alphabetized, the Pascal FORWARD command must be used in order to compile the resulting program correctly.

If an index table is built to perform the alphabetizing, the FORWARD command is unnecessary.

4. Allow for output of the pretty printer to be directed to a file, the screen, and/or a printer.

5. The keywords CONST, TYPE, and VAR should be on a line by themselves.

6. Only one declaration per line is allowed in the CONST, TYPE, and VAR sections of the program.

7. The BEGIN and/or END of each section, declaration, or construct should indicate the construct. Examples are as follows:

```
END (*record*)

BEGIN (*case*)
   (statements)
END (*case*)

etc.
```

8. The BEGIN and END of each procedure or function should contain a comment indicating the name of the procedure or function.

9. Each line should be less than or equal to 120 characters in length. However, if you are printing on 8.5" X 11" paper, each line should not exceed 80 characters in length.

10. The keywords REPEAT, BEGIN, and RECORD should have no program text (other than comments) following them on the line in which they appear.

11. All matching ENDs, UNTILs, etc. should be on lines by themselves and aligned with their corresponding previous keyword. An exception is with items similar to the RECORD construct. In this situation, the matching END should be aligned with the name of the record. For example,

```
RECNAME = RECORD
    (statements)
END (*record*)
```

12. Two blank lines should appear before and after each procedure and function.

13. At least one space should appear before and after each ":=", ":", and "=".

14. Only one executable statement is allowed per line.

15. The number of spaces for indentation should be between 3 and 10 (inclusive). The exact number is left to your discretion. However, the indentation must be consistent throughout the program.

16. The statements or declarations within an indented body should be aligned. For example, line up all variable declarations indented under a VAR statement. Also line up all the statements indented under an IF-THEN-ELSE statement.

17. The PROGRAM statement, CONST, TYPE, VAR keywords, BEGIN, and END of the main program should be aligned at the left margin.

18. Procedure and function headings should be aligned with the keywords of the surrounding procedure, function, or program.

19. The declaration keywords (CONST, TYPE, VAR), and BEGIN-END blocks of procedure and functions should be aligned with the procedure headings. Procedures/functions that are physically within another procedure/function (not the main program) should be indented and any declaration keywords aligned with the appropriate headings.

20. All declarations in the CONST, TYPE, and VAR sections should be indented from these keywords.

21. The bodies of FOR, IF-THEN, IF-THEN-ELSE, WHILE, WITH, and CASE statements along with RECORD declarations should be indented from their corresponding keywords.

22. If a body of a FOR, IF-THEN, IF-THEN-ELSE, WHILE, or WITH statement is a compound statement (more than one command), then the BEGIN should follow the keyword on then next line and the END should be on a line by itself aligned with the corresponding BEGIN. When a REPEAT loop appears on more then one line, the UNTIL is aligned with the REPEAT.

Comments

23. It is at your discretion to choose left and right column delimiters for comments. These delimiters can be aligned with the Pascal statements that are being documented or they can be to the right of the Pascal statements. The key with comments will be in "keeping" them with the Pascal statement they describe.

24. Each CONST, TYPE, and VAR declaration must have a descriptive comment appended to the right of the line.

The following shows some acceptable formats for IF-THEN-ELSE statements. The specific format chosen by your pretty printer package is at your discretion and does not have to necessarily match what is shown below. However, the format you choose should be thoroughly documented and consistent throughout the program.

1. A compound IF-THEN-ELSE statement may be formatted as follows:

```
IF (expression) THEN
   BEGIN
      (statements)
   END
ELSE
   BEGIN
      (statements)
   END
```

2. A non-compound IF-THEN-ELSE statement may be formatted as follows:

```
IF (expression) THEN
   statement
ELSE
   statement
```

3. If nested IF-THEN-ELSE statements exist, they may be formatted as follows:

```
IF (expression) THEN
   BEGIN
      (statements)
   END
ELSE
   BEGIN
      IF (expression) THEN
         BEGIN
            (statements)
         END
      ELSE
         BEGIN
            (statements)
         END
      (statements)
   END
```

Limitations

1. You may assume that all of the source code is in 132-column format.

2. You may also assume that no syntax errors exist in the programs which shall be run through the pretty printer.

APPENDIX B

SIX ASSUMPTIONS

MODIFICATIONS TO PRETTY PRINTER SPECIFICATIONS

Assumptions made during the Autumn 1989 Software Engineering Class about the Pascal source code that is the input for the Pretty Printer.

1. Pascal source code must be able to be compiled, therefore having no syntax errors.

2. PROCEDURES must not be nested.

3. There must be at least one blank line between logical sections of code (such as: VAR, CONST, TYPE, PROGRAM, PROCEDURES and FUNCTIONS).

4. All comments must be closed on the same line that they are opened.

5. There can not be any statements after, or in-between comments

6. All PROCEDURES must have a forward command (if you are physically sorting the PROCEDURES).

7. Output should cover 120 columns, not 120 columns or 80 columns.

APPENDIX C

INITIAL QUESTIONNAIRE
USED TO DETERMINE LEVEL OF EXPERIENCE, DEMOGRAPHICS, GPAS AND TEAM MEMBER PREFERENCES

IS 322 (SOFTWARE ENGINEERING)

NAME: _____

AGE: _____

PREVIOUS COMPUTER COURSES:
(please include any computer science, computer engineering, high school courses, special work shops, etc)

PREVIOUS WORK EXPERIENCE - WITH COMPUTERS:
(please list all co-op and other work experiences that involve computers - including the company, all your responsibilities and the amount of time)

(MICROS, Main Frames, VAX, Packages, CASE Tools, Other)

OTHER COURSES YOU ARE TAKING THIS QUARTER:

PEOPLE YOU WOULD LIKE TO WORK WITH:

PEOPLE YOU WOULD NOT LIKE TO WORK WITH:

GPA:
OVERALL _____
IS _____

What are your strengths (academically and/or computer related)?

What are your weaknesses (academically and/or computer related)?

Have you ever worked in a design or programming team? Where? Type of project?

What are your feelings about working in a team environment? (better situation, worse?) And why?

APPENDIX D

COURSE SYLLABUS AND GRADING POLICY

REQUIREMENTS FOR PROGRAMMERS MANUAL, USERS MANUAL AND PROGRAMMERS LOGS

SOFTWARE ENGINEERING (IS 322)

Course prerequisites: IS 280 and IS 321.

Text Books:

Required: Structured Systems Design
Page-Jones
Yourdon Press (1988) 2nd Edition

Optional: Software Engineering Concepts
Richard Fairley
McGraw-Hill (1985)

This course is designed to further develop your knowledge of structured programming techniques and methods, particularly as they relate to larger, multi-programmer projects. You will be working in teams with 3 or 4 classmates (depending upon the size of the class). A major project will be completed in three separate phases: design, implementation and testing. You will be implementing another team's design and then testing another team's implementation. The members of the teams may change for each phase: you may be working with different team members as the project stages change.

Student Evaluation:

Final grades will be determined as follows:

Project	
Design	25%
Implementation	25%
Testing	10%
Homework	10%
Log	5%
Final Test	25%

+/- 5% Instructor's discretion

Homework will consist of project progress reports and two very short (2 -3 page) papers: due dates are attached. Each team member will submit his/her own evaluation of the progress that the team is making during the particular project phase. In addition to the progress reports, each student should maintain a log of time spent on ALL course activities: readings, meetings, writing code, debugging, testing, etc. Each entry in the log should be annotated with comments: these logs will provide an overview of your activities during the quarter and possibly help me change/improve the course. Logs will be handed in at the final exam, but will be date stamped during the quarter.

SOFTWARE ENGINEERING (IS 322)

WEEK	TOPICS
1	Chapter 1 Introduction to Software Engineering Chapter 2 Planning a Software Project
2 + 3	Chapters 3,8,9 Software Design
4 + 5	Chapters 4,5,7,11 Implementation
6	Walkthroughs + Inspections
7 + 8	Handouts Verification and Validation 13.4 (PJ) Software Maintenance Chapter 9 (Fairley)
9 + 10	Chapter 3 (Fairley) Cost Estimation 13.5 (PJ) Chapter 10 (Fairley) Summary

**You are responsible for material in the readings and handouts, whether it is covered or not covered in class. Attendance is not required, but 'I missed class' is not an acceptable excuse for not being aware of any changes of due dates, project requirements, test dates or meetings.

**It is expected that all members of the class will act in an honest, ethical and moral manner.

****
No drops after the third week of class
No makeup tests or homework
No final grades of incomplete
****

The project will be evaluated as follows:
The completeness of the project and how well it performs.
(100%, 75% ....)
Documentation
Programmer's manual
User's manual
Internal documentation
The actual code for the project.
Your programs must be structured.
One function/task - one module
All variables must be meaningful
No global variables; all values must be passed through parameter lists.
Indenting, labeling and other standards - covered in previous classes - should be followed.
Group Dynamics/Interaction/Cohesiveness
Progress at Milestones:
walkthroughs
inspections
There will be a complete evaluation at the end of each of the two phases based on the work submitted, instructors evaluation and each team member's independent evaluation of each of the other member of the team.

Programmer's manual:
For the technically trained person - install, implement and modify the project.
Table of contents
Index
Built in the design phase:
verbal technical description
structure charts
data flow diagrams
data dictionary
module function specifications
module interfaces and integration
data structures and/or record layouts
Built in the implementation phase:
the actual code
implementation restrictions
any changes (and reasons) to the original design - and effects of the changes
compiling and linking instructions
any changes to the data structures and/or records and why internal documentation

Built in the testing phase
copy of test data
suggestions for improvement
better performance
better interface
better error handling

User's manual:
For the untrained/naive user - assume they know how to turn on the computer and do not understand anything else about computers.
Table of contents
Index
Describe the entire package and its usage
A tutorial?
Error handling
Started during the design phase, but fully developed and then modified (if necessary) during the testing phase.

Programmer's log:
As stated above, a complete record of your activities. If you were to be replaced on the project (and that will not happen after the third week of the quarter) someone could read your activity log and be able to replace you on the team. This should also include any structure charts, data flow diagrams, notes, specifications, decision rational, comments - any relevant information.

**All modules will be compiled separately. This will allow you to test them separately and then link them for the final project.

***In addition to the weekly progress reports..for every meeting that your group has someone (designated by the group and may be someone different each time) will take minutes of the meeting and give me a copy. Minutes will include:
Time and duration of meeting
Names of attendees
Subject matter
Decisions and why
Status of project
what will be accomplished during the following week
Time and place of next meeting

**Warning: This course could be dangerous to your health, your social life and your performance in other courses. As usual, get started early, work steadily and try to get some sleep.

APPENDIX E

PROGRAMS USED TO TEST THE PRETTY PRINTER

```
{****  Test for Capitalizing of all reserved words  ****}

program sotypical (input,output);                                {PROGRAm}

const  limit = 10;                                               {CONSt}
       poundsign = '#';
       amorcita = 'ilana';

type   hues = (red,blue,green,orange,violet);                    {TYPe}
       shades = blue..orange;
       smallnumbers = 1..10;
       string = packed array[1..limit] of char;                  {PACKEd}
       class = record
                  name : string;
                  units : integer;
                  grade : char;
                end;
       grades = array[smallnumbers] of class;
       colorcount=array[1..10,'A'..'Z'] of hues;                 {ARRAy}
       classfile=file of class;                                  {FILe}{Of}
       pastels= set of shades;                                   {SEt}
       nextword=^sentence;
       sentence = record                                         {RECORd}
                     currentword : string;
                     comingword  : nextword;
                  end;

var  high, low, counter : integer;                                 {VAr}
     first, last: char;
     height, weight:real;
     testing, debugging: boolean;
     colors : hues;
     shorts: smallnumbers;
     name : string;
     onecourse : class;
     curriculum: grades;
     colorsquares: colorcount;
     schedule : classfile;
     source, results : text;
     crayons : pastels;
     list, pointer : nextword;
     count : integer;

label 1;

procedure verybusy (    incoming:integer;
                    var outgoing:integer); forward; {PROCEDURe}

function capital(parameter : char):boolean; forward;  {FUNCTIOn}{FORWARd}

procedure verybusy;

var local : integer;

begin
   readln(local);
   outgoing := incoming * local;
end;

function capital;

begin
   capital := parameter in ['A'..'Z'];                  {In}
end;

begin
   writeln('Let''s start demonstrating things.');
   readln(first,last);
```

```
   if first <= last then
      begin
         write(first,' and ',last, ' are');
         writeln(' in alphabetical order.');
      end;

   if first = poundsign then                          {If}{THEn}
      high := (100 mod 90)                            {MOd}
   else                                               {ELSe}
      high := 20;

   for counter := 1 to limit do                       {FOr}{To}
      read(name[counter]);

   case limit div 2 of                                {CASe}{Of}{DIv}
      0, 1, 2, 3, 4, 5 : ;
      6, 7, 8, 9: writeln('within range.');
   end;

   repeat                                             {REPEAt}
      read(shorts);
   until (shorts=1) or (shorts=10);                   {Or}{UNTIl}

   while not eoln do                                  {NOt}{WHILe}
      begin                                           {BEGIn}
         read(first);
         writeln(first);
      end;

   with onecourse do                                  {WITh}{Do}
      begin
         name := 'study hall';
         units := 5;
         grade := 'p'
      end;

   for count := 5 downto 1 do                         {DOWNTo}
      write(' ');
   testing := capital(first);
   verybusy(high,low);
   reset(source);
   read(source,last);
   rewrite(results);
   write(results,last);
   new(list);
   list^.comingword := nil;                        {NIL}
   pointer := list;
   goto 1;
1: end.                                            {END.}
```

```
{*****  Test Alphabetizing of all delarations  *****}

program sotypical (input,output);

const  limit = 10;                                          {comment 3}
       poundsign = '#';                                     {comment 2}
       amorcita = 'ilana';                                  {comment 1}

type   hues = (red,blue,green,orange,violet);               {comment 5}
       shades = blue..orange;                               {comment 9}
       smallnumbers = 1..10;                                {comment 10}
       string = packed array[1..limit] of char;             {comment 11}
       class = record                                       {comment 1}
                  name : string;                            {comment 1b}
                  units : integer;                          {comment 1c}
                  grade : char;                             {comment 1a}
                end;                                        {comment 1d}
       grades = array[smallnumbers] of class;               {comment 4}
       colorcount=array[1..10,'A'..'Z'] of hues;            {comment 3}
       classfile=file of class;                             {comment 2}
       pastels= set of shades;                              {comment 7}
       nextword=^sentence;                                  {comment 6}
       sentence = record                                    {comment 8}
                     currentword : string;                  {comment 8b}
                     comingword  : nextword;                {comment 8a}
                  end;                                      {comment 8c}

var  high, low, counter : integer;                          {comment 8}
     first, last: char;                                     {comment 6}
     height, weight:real;                                   {comment 7}
     testing, debugging: boolean;                           {comment 15}
     colors : hues;                                         {comment 1}
     shorts: smallnumbers;                                  {comment 13}
     name : string;                                         {comment 10}
     onecourse : class;                                     {comment 11}
     curriculum: grades;                                    {comment 5}
     colorsquares: colorcount;                              {comment 2}
     schedule : classfile;                                  {comment 12}
     source, results : text;                                {comment 14}
     crayons : pastels;                                     {comment 4}
     list, pointer : nextword;                              {comment 9}
     count : integer;                                       {comment 3}

label 1;

procedure verybusy (    incoming:integer;
                    var outgoing:integer); forward;

function capital(parameter : char):boolean; forward;


procedure verybusy;

var local : integer;

begin
   readln(local);
   outgoing := incoming * local;
end;

function capital;

begin
   capital := parameter in ['A'..'Z'];
end;

begin
   writeln('Let''s start demonstrating things.');
   readln(first,last);
```

```
   if first <= last then
      begin
         write(first,' and ',last, ' are');
         writeln(' in alphabetical order.');
      end;

   if first = poundsign then
      high := (100 mod 90)
   else
      high := 20;

   for counter := 1 to limit do
      read(name[counter]);

   case limit div 2 of
      0, 1, 2, 3, 4, 5 : ;
      6, 7, 8, 9: writeln('within range.');
   end;

   repeat
      read(shorts);
   until (shorts=1) or (shorts=10);

   while not eoln do
      begin
         read(first);
         writeln(first);
      end;

   with onecourse do
      begin
         name := 'study hall';
         units := 5;
         grade := 'p'
      end;

   for count := 5 downto 1 do
      write(' ');
   testing := capital(first);
   verybusy(high,low);
   reset(source);
   read(source,last);
   rewrite(results);
   write(results,last);
   new(list);
   list^.comingword := nil;
   pointer := list;
   goto 1;
1: end.
```

```
{*****  Test Alphabetizing of Procedures and Functions  *****}


program sotypical (input,output);

const  limit = 10;
       poundsign = '#';
       amorcita = 'ilana';

type   hues = (red,blue,green,orange,violet);
       shades = blue..orange;
       smallnumbers = 1..10;
       string = packed array[1..limit] of char;
       class = record
                  name : string;
                  units : integer;
                  grade : char;
                end;
       grades = array[smallnumbers] of class;
       colorcount=array[1..10,'A'..'Z'] of hues;
       classfile=file of class;
       pastels= set of shades;
       nextword=^sentence;
       sentence = record
                     currentword : string;
                     comingword  : nextword;
                  end;

var  high, low, counter : integer;
     first, last: char;
     height, weight:real;
     testing, debugging: boolean;
     colors : hues;
     shorts: smallnumbers;
     name : string;
     onecourse : class;
     curriculum: grades;
     colorsquares: colorcount;
     schedule : classfile;
     source, results : text;
     crayons : pastels;
     list, pointer : nextword;
     count : integer;

label 1;

procedure verybusy (    incoming:integer;
                     var outgoing:integer); forward;

function capital3(parameter : char): boolean; forward;

function capital(parameter : char) : boolean; forward;

function capital2(parameter : char):boolean; forward;


function capital3;       {--- Should be third function }

begin
   if parameter = 'a' then
      capital3 := true;
end;


function capital2;       {--- Should be second function }

begin
   if parameter = 'a' then
```

```
      capital2 := true;
end;


procedure verybusy;

var local : integer;

begin
   readln(local);
   outgoing := incoming * local;
end;

function capital;                          {--- Should be first function  }

begin
   capital := parameter in ['A'..'Z'];
end;



begin
   writeln('Let''s start demonstrating things.');
   readln(first,last);
   if first <= last then
      begin
         write(first,' and ',last, ' are');
         writeln(' in alphabetical order.');
      end;

   if first = poundsign then
      high := (100 mod 90)
   else
      high := 20;

   for counter := 1 to limit do
      read(name[counter]);

   case limit div 2 of
      0, 1, 2, 3, 4, 5 : ;
      6, 7, 8, 9: writeln('within range.');
   end;

   repeat
      read(shorts);
   until (shorts=1) or (shorts=10);

   while not eoln do
      begin
         read(first);
         writeln(first);
      end;

   with onecourse do
      begin
         name := 'study hall';
         units := 5;
         grade := 'p'
      end;


   for count := 5 downto 1 do
      write(' ');
   testing := capital(first);
   verybusy(high,low);
   reset(source);
   read(source,last);
   rewrite(results);
   write(results,last);
```

```
    new(list);
    list^.comingword := nil;
    pointer := list;
    goto 1;
1: end.
```

```
{*****  Test Output to a File  *****}

program sotypical (input,output);                              {PROGRAm}

const  limit = 10;                                             {CONSt}
       poundsign = '#';
       amorcita = 'ilana';

type   hues = (red,blue,green,orange,violet);                  {TYPe}
       shades = blue..orange;
       smallnumbers = 1..10;
       string = packed array[1..limit] of char;                {PACKEd}
       class = record
                  name : string;
                  units : integer;
                  grade : char;
                end;
       grades = array[smallnumbers] of class;
       colorcount=array[1..10,'A'..'Z'] of hues;               {ARRAy}
       classfile=file of class;                                {FILe}{OF}
       pastels= set of shades;                                 {SEt}
       nextword=^sentence;
       sentence = record                                       {RECORd}
                     currentword : string;
                     comingword  : nextword;
                  end;

var  high, low, counter : integer;                              {VAr}
     first, last: char;
     height, weight:real;
     testing, debugging: boolean;
     colors : hues;
     shorts: smallnumbers;
     name : string;
     onecourse : class;
     curriculum: grades;
     colorsquares: colorcount;
     schedule : classfile;
     source, results : text;
     crayons : pastels;
     list, pointer : nextword;
     count : integer;

label 1;

procedure verybusy (    incoming:integer;
                    var outgoing:integer); forward; {PROCEDURe}

function capital(parameter : char):boolean; forward;  {FUNCTIOn}{FORWARd}


procedure verybusy;

var local : integer;

begin
   readln(local);
   outgoing := incoming * local;
end;

function capital;

begin
   capital := parameter in ['A'..'Z'];                          {In}
end;

begin
   writeln('Let''s start demonstrating things.');
   readln(first,last);
```

```
if first <= last then
   begin
      write(first,' and ',last, ' are');
      writeln(' in alphabetical order.');
   end;

if first = poundsign then                          {If}{THEn}
   high := (100 mod 90)                            {MOd}
else                                               {ELSe}
   high := 20;

for counter := 1 to limit do                       {FOr}{To}
   read(name[counter]);

case limit div 2 of                                {CASe}{Of}{DIv}
   0, 1, 2, 3, 4, 5 : ;
   6, 7, 8, 9: writeln('within range.');
end;

repeat                                             {REPEAt}
   read(shorts);
until (shorts=1) or (shorts=10);                   {Or}{UNTIl}

while not eoln do                                  {NOt}{WHILe}
   begin                                           {BEGIn}
      read(first);
      writeln(first);
   end;

with onecourse do                                  {WITh}{Do}
   begin
      name := 'study hall';
      units := 5;
      grade := 'p'
   end;

for count := 5 downto 1 do                         {DOWNTo}
   write(' ');
testing := capital(first);
verybusy(high,low);
reset(source);
read(source,last);
rewrite(results);
write(results,last);
new(list);
list^.comingword := nil;                           {NIL}
pointer := list;
goto 1;
1: end.                                            {END.}
```

```
{*****  Test CONST, TYPE and VAR reserved words are on 1 line alone  ****}


program sotypical (input,output);

const  limit = 10;
       poundsign = '#'; amorcita = 'ilana';          {--- poundsign = '#'; amorcita = 'ilana';}

type   hues = (red,blue,green,orange,violet);
       shades = blue..orange;
       smallnumbers = 1..10;
       string = packed array[1..limit] of char;
       class = record
                 name : string;
                 units : integer;
                 grade, junk : char;                 {--- Grade, Junk  }
               end;
       grades = array[smallnumbers] of class;
       colorcount=array[1..10,'A'..'Z'] of hues;
       classfile=file of class;
       pastels= set of shades;
       nextword=^sentence;
       sentence = record
                    currentword : string;
                    comingword  : nextword;
                  end;

var  high, low, counter : integer;                 {--- High Low Counter }
     first, last: char;                            {--- First, Last }
     height, weight:real;                          {--- Hight, Weight  }
     testing, debugging: boolean;                  {--- Testing, Debugging  }
     colors : hues;
     shorts: smallnumbers;
     name : string;
     onecourse : class;
     curriculum: grades;
     colorsquares: colorcount;
     schedule : classfile;
     source, results : text;
     crayons : pastels;
     list, pointer : nextword;
     count,count2,count3,count4,count5,count6 : integer;
                           {--- Count,Count2,Count3,Count4,Count5,Count6 }

label 1;

procedure verybusy (    incoming:integer;
                    var outgoing:integer); forward;

function capital(parameter : char):boolean; forward;


procedure verybusy;

var local : integer;

begin
   readln(local);
   outgoing := incoming * local;
end;

function capital;

begin
   capital := parameter in ['A'..'Z'];
end;

begin
```

```
   writeln('Let''s start demonstrating things.');
   readln(first,last);
   if first <= last then
      begin
         write(first,' and ',last, ' are');
         writeln(' in alphabetical order.');
      end;

   if first = poundsign then
      high := (100 mod 90)
   else
      high := 20;

   for counter := 1 to limit do
      read(name[counter]);

   case limit div 2 of
      0, 1, 2, 3, 4, 5 : ;
      6, 7, 8, 9: writeln('within range.');
   end;

   repeat
      read(shorts);
   until (shorts=1) or (shorts=10);

   while not eoln do
      begin
         read(first);
         writeln(first);
      end;

   with onecourse do
      begin
         name := 'study hall';
         units := 5;
         grade := 'p'
      end;

   for count := 5 downto 1 do
      write(' ');
   testing := capital(first);
   verybusy(high,low);
   reset(source);
   read(source,last);
   rewrite(results);
   write(results,last);
   new(list);
   list^.comingword := nil;
   pointer := list;
   goto 1;
1: end.
```

```
{*****  Test Only one declaration per line in CONST, TYPE and VAR Section  ****}


program sotypical (input,output);

const  limit = 10;
       poundsign = '#'; amorcita = 'ilana';          {--- poundsign = '#'; amorcita = 'ilana';}

type   hues = (red,blue,green,orange,violet);
       shades = blue..orange;
       smallnumbers = 1..10;
       string = packed array[1..limit] of char;
       class = record
                  name : string;
                  units : integer;
                  grade, junk : char;                {--- Grade, Junk  }
                end;
       grades = array[smallnumbers] of class;
       colorcount=array[1..10,'A'..'Z'] of hues;
       classfile=file of class;
       pastels= set of shades;
       nextword=^sentence;
       sentence = record
                     currentword : string;
                     comingword  : nextword;
                  end;

var  high, low, counter : integer;                   {--- High Low Counter }
     first, last: char;                              {--- First, Last }
     height, weight:real;                            {--- Hight, Weight  }
     testing, debugging: boolean;                    {--- Testing, Debugging  }
     colors : hues;
     shorts: smallnumbers;
     name : string;
     onecourse : class;
     curriculum: grades;
     colorsquares: colorcount;
     schedule : classfile;
     source, results : text;
     crayons : pastels;
     list, pointer : nextword;
     count,count2,count3,count4,count5,count6 : integer;
                 {--- Count,Count2,Count3,Count4,Count5,Count6 }

label 1;

procedure verybusy (    incoming:integer;
                    var outgoing:integer); forward;

function capital(parameter : char):boolean; forward;


procedure verybusy;

var local : integer;

begin
   readln(local);
   outgoing := incoming * local;
end;

function capital;

begin
   capital := parameter in ['A'..'Z'];
end;

begin
```

```
   writeln('Let''s start demonstrating things.');
   readln(first,last);
   if first <= last then
      begin
         write(first,' and ',last, ' are');
         writeln(' in alphabetical order.');
      end;

   if first = poundsign then
      high := (100 mod 90)
   else
      high := 20;

   for counter := 1 to limit do
      read(name[counter]);

   case limit div 2 of
      0, 1, 2, 3, 4, 5 : ;
      6, 7, 8, 9: writeln('within range.');
   end;

   repeat
      read(shorts);
   until (shorts=1) or (shorts=10);

   while not eoln do
      begin
         read(first);
         writeln(first);
      end;

   with onecourse do
      begin
         name := 'study hall';
         units := 5;
         grade := 'p'
      end;

   for count := 5 downto 1 do
      write(' ');
   testing := capital(first);
   verybusy(high,low);
   reset(source);
   read(source,last);
   rewrite(results);
   write(results,last);
   new(list);
   list^.comingword := nil;
   pointer := list;
   goto 1;
1: end.
```

```
{****  Test Begin and End construct.  Indecation of Construct Test  ****}

program sotypical (input,output);

const  limit = 10;
       poundsign = '#';
       amorcita = 'ilana';

type   hues = (red,blue,green,orange,violet);
       shades = blue..orange;
       smallnumbers = 1..10;
       string = packed array[1..limit] of char;
       class = record
                 name : string;
                 units : integer;
                 grade, junk : char;
               end;                                          {Record}
       grades = array[smallnumbers] of class;
       colorcount=array[1..10,'A'..'Z'] of hues;
       classfile=file of class;
       pastels= set of shades;
       nextword=^sentence;
       sentence = record
                    currentword : string;
                    comingword  : nextword;
                  end;                                       {Record}

var  high, low, counter : integer;
     first, last: char;
     height, weight:real;
     testing, debugging: boolean;
     colors : hues;
     shorts: smallnumbers;
     name : string;
     onecourse : class;
     curriculum: grades;
     colorsquares: colorcount;
     schedule : classfile;
     source, results : text;
     crayons : pastels;
     list, pointer : nextword;
     count,count2,count3,count4,count5,count6 : integer;

label 1;

procedure verybusy (    incoming:integer;
                    var outgoing:integer); forward;

function capital(parameter : char):boolean; forward;


procedure verybusy;

var local : integer;

begin                                              {Procedure}
   readln(local);
   outgoing := incoming * local;
end;                                               {Procedure}

function capital;

begin                                               {Function}
   capital := parameter in ['A'..'Z'];
end;                                                {Function}

begin                                                 {Program}
   writeln('Let''s start demonstrating things.');
   readln(first,last);
```

```
   if first <= last then
      begin                                          {If}
         write(first,' and ',last, ' are');
         writeln(' in alphabetical order.');
      end;                                           {If}

   if first = poundsign then
      high := (100 mod 90)
   else
      high := 20;

   for counter := 1 to limit do
      read(name[counter]);

   case limit div 2 of
      0, 1, 2, 3, 4, 5 : ;
      6, 7, 8, 9: writeln('within range.');
   end;                                              {Case}

   repeat
      read(shorts);
   until (shorts=1) or (shorts=10);

   while not eoln do
      begin                                          {While}
         read(first);
         writeln(first);
      end;                                           {While}

   with onecourse do
      begin                                          {With}
         name := 'study hall';
         units := 5;
         grade := 'p'
      end;                                           {With}

   for count := 5 downto 1 do
      write(' ');
   testing := capital(first);
   verybusy(high,low);
   reset(source);
   read(source,last);
   rewrite(results);
   write(results,last);
   new(list);
   list^.comingword := nil;
   pointer := list;
   goto 1;
1: end.                                              {Program}
```

```
{****  Test Begin and End construct. Name of Procedure/Function indecated ****}

program sotypical (input,output);

const  limit = 10;
       poundsign = '#';
       amorcita = 'ilana';

type   hues = (red,blue,green,orange,violet);
       shades = blue..orange;
       smallnumbers = 1..10;
       string = packed array[1..limit] of char;
       class = record
                  name : string;
                  units : integer;
                  grade, junk : char;
                end;
       grades = array[smallnumbers] of class;
       colorcount=array[1..10,'A'..'Z'] of hues;
       classfile=file of class;
       pastels= set of shades;
       nextword=^sentence;
       sentence = record
                     currentword : string;
                     comingword  : nextword;
                  end;

var  high, low, counter : integer;
     first, last: char;
     height, weight:real;
     testing, debugging: boolean;
     colors : hues;
     shorts: smallnumbers;
     name : string;
     onecourse : class;
     curriculum: grades;
     colorsquares: colorcount;
     schedule : classfile;
     source, results : text;
     crayons : pastels;
     list, pointer : nextword;
     count,count2,count3,count4,count5,count6 : integer;

label 1;

procedure verybusy (    incoming:integer;
                    var outgoing:integer); forward;

function capital(parameter : char):boolean; forward;


procedure verybusy;

var local : integer;

begin                                          {VeryBusy}
   readln(local);
   outgoing := incoming * local;
end;                                           {VeryBusy}

function capital;

begin                                           {Capital}
   capital := parameter in ['A'..'Z'];
end;                                            {Capital}

begin
   writeln('Let''s start demonstrating things.');
   readln(first,last);
```

```
   if first <= last then
      begin
         write(first,' and ',last, ' are');
         writeln(' in alphabetical order.');
      end;

   if first = poundsign then
      high := (100 mod 90)
   else
      high := 20;

   for counter := 1 to limit do
      read(name[counter]);

   case limit div 2 of
      0, 1, 2, 3, 4, 5 : ;
      6, 7, 8, 9: writeln('within range.');
   end;

   repeat
      read(shorts);
   until (shorts=1) or (shorts=10);

   while not eoln do
      begin
         read(first);
         writeln(first);
      end;

   with onecourse do
      begin
         name := 'study hall';
         units := 5;
         grade := 'p'
      end;

   for count := 5 downto 1 do
      write(' ');
   testing := capital(first);
   verybusy(high,low);
   reset(source);
   read(source,last);
   rewrite(results);
   write(results,last);
   new(list);
   list^.comingword := nil;
   pointer := list;
   goto 1;
1: end.
```

```
{****  Test Line Size no more than 120 Characters Long ****}

       {****  The Line Below Should be broken up and shorter than 120 Charactes ****}
program sotypical (input,output);  const  limit = 10;      poundsign = '#';      amorcita = 'ilana';
type   hues = (red,blue,green,orange,violet);        shades = blue..orange;        smallnumbers =
1..10;       string = packed array[1..limit] of char;
      class = record
                 name : string;
                 units : integer;
                 grade, junk : char;
               end;
      grades = array[smallnumbers] of class;
      colorcount=array[1..10,'A'..'Z'] of hues;
      classfile=file of class;
      pastels= set of shades;
      nextword=^sentence;
      sentence = record
                    currentword : string;
                    comingword  : nextword;
                 end;

var  high, low, counter : integer;
     first, last: char;
     height, weight:real;
     testing, debugging: boolean;
     colors : hues;
     shorts: smallnumbers;
     name : string;
     onecourse : class;
     curriculum: grades;
     colorsquares: colorcount;
     schedule : classfile;
     source, results : text;
     crayons : pastels;
     list, pointer : nextword;
     {*****  The Line below should be broken up and  be less than 120 characters  *****}
     count,                    count2,          count3,          count4,          count5,
    count6,          count7,        count8,       count9,    count10 : integer;

label 1;

procedure verybusy (    incoming:integer;
                    var outgoing:integer); forward;

function capital(parameter : char):boolean; forward;


procedure verybusy;

var local : integer;

begin                                          {VeryBusy}
   readln(local);
   outgoing := incoming * local;
end;                                           {VeryBusy}

function capital;

begin                                           {Capital}
   capital := parameter in ['A'..'Z'];
end;                                            {Capital}

begin
   writeln('Let''s start demonstrating things.');
   readln(first,last);

   {*****  The following line should be broken up and be less than 120 characters in length  *****}
   if (first <= last) and (first < last) or (first > last) or (last > first) or (last <> first) and
(last > first) and (last > first) and (first > last) then
```

```
      begin
      end;

   if first <= last then
      begin
         write(first,' and ',last, ' are');
         writeln(' in alphabetical order.');
      end;

   if first = poundsign then
      high := (100 mod 90)
   else
      high := 20;

   for counter := 1 to limit do
      read(name[counter]);

   case limit div 2 of
      0, 1, 2, 3, 4, 5 : ;
      {*****  The following line should be broken up and be less than 120 characters long  *****}
      6,                                7,                                        8,
      9:         writeln('within range.');
   end;

   repeat
      read(shorts);
      {*****  The following line should be broken up and be less than 120 characters long  *****}
   until (shorts=1) or (shorts=10) or (shorts=10) or (shorts=1) or (shorts=10) or (shorts=10) or
(shorts=1) or (shorts=10) or (shorts=10);

{**************         This comment should be broken up into more than one piece and be under 120
characters long.   It should also be compilable after it has been broken up  *****}

   while not eoln do
      begin
         read(first);
         writeln(first);
      end;

   with onecourse do
      begin
         name := 'study hall';
         units := 5;
         grade := 'p'
      end;

   for count := 5 downto 1 do
      write(' ');
   testing := capital(first);
   verybusy(high,low);
   reset(source);
   read(source,last);
   rewrite(results);
   write(results,last);
   new(list);
   list^.comingword := nil;
   pointer := list;
   goto 1;
1: end.
```

```
{****  Test that the reserved words RECORD, BEGIN, and REPEAT are on lines by themselfs (excluding
comments)  ****}


program sotypical (input,output);

const  limit = 10;
       poundsign = '#';
       amorcita = 'ilana';

type   hues = (red,blue,green,orange,violet);
       shades = blue..orange;
       smallnumbers = 1..10;
       string = packed array[1..limit] of char;
       class = record  name : string;                    {Record Only}
                  units : integer;
                  grade, junk : char;
               end;
       grades = array[smallnumbers] of class;
       colorcount=array[1..10,'A'..'Z'] of hues;
       classfile=file of class;
       pastels= set of shades;
       nextword=^sentence;
       sentence = record    currentword : string;        {Record Only}
                     comingword  : nextword;
                  end;

var  high, low, counter : integer;
     first, last: char;
     height, weight:real;
     testing, debugging: boolean;
     colors : hues;
     shorts: smallnumbers;
     name : string;
     onecourse : class;
     curriculum: grades;
     colorsquares: colorcount;
     schedule : classfile;
     source, results : text;
     crayons : pastels;
     list, pointer : nextword;
     count,count2,count3,count4,count5,count6 : integer;

label 1;

procedure verybusy (    incoming:integer;
                    var outgoing:integer); forward;

function capital(parameter : char):boolean; forward;


procedure verybusy;

var local : integer;

begin   readln(local);                                {Begin Only}
   outgoing := incoming * local;
end;

function capital;

begin    capital := parameter in ['A'..'Z'];          {Begin Only}
end;

begin    writeln('Let''s start demonstrating things.');  {Begin Only}
   readln(first,last);
   if first <= last then
     begin        write(first,' and ',last, ' are');    {Begin Only}
        writeln(' in alphabetical order.');
```

```
      end;

   if first = poundsign then
      high := (100 mod 90)
   else
      high := 20;

   for counter := 1 to limit do
      read(name[counter]);

   case limit div 2 of
      0, 1, 2, 3, 4, 5 : ;
      6, 7, 8, 9: writeln('within range.');
   end;

   repeat      read(shorts);                    {Repeat Only}
   until (shorts=1) or (shorts=10);

   while not eoln do
      begin         read(first);                {Begin Only}
         writeln(first);
      end;

   with onecourse do
      begin                                     {Begin Only - Leave Comment}
         name := 'study hall';
         units := 5;
         grade := 'p'
      end;

   for count := 5 downto 1 do
      write(' ');
   testing := capital(first);
   verybusy(high,low);
   reset(source);
   read(source,last);
   rewrite(results);
   write(results,last);
   new(list);
   list^.comingword := nil;
   pointer := list;
   goto 1;
1: end.
```

```
{****  Test that the Ends, Untils, and Records are on lines by themselves and ****}
{****  matched up in the same column as the begins, repeats and record name   ****}
{****  in the record.                                                         ****}


program sotypical (input,output);

const  limit = 10;
       poundsign = '#';
       amorcita = 'ilana';

type   hues = (red,blue,green,orange,violet);
       shades = blue..orange;
       smallnumbers = 1..10;
       string = packed array[1..limit] of char;
       class = record  name : string;
                 units : integer;
                 grade, junk : char;
               end;                                    {end should line up with class}
       grades = array[smallnumbers] of class;
       colorcount=array[1..10,'A'..'Z'] of hues;
       classfile=file of class;
       pastels= set of shades;
       nextword=^sentence;
       sentence = record    currentword : string;      {Record Only}
                    comingword  : nextword;
                  end;

var  high, low, counter : integer;
     first, last: char;
     height, weight:real;
     testing, debugging: boolean;
     colors : hues;
     shorts: smallnumbers;
     name : string;
     onecourse : class;
     curriculum: grades;
     colorsquares: colorcount;
     schedule : classfile;
     source, results : text;
     crayons : pastels;
     list, pointer : nextword;
     count,count2,count3,count4,count5,count6 : integer;

label 1;

procedure verybusy (    incoming:integer;
                    var outgoing:integer); forward;

function capital(parameter : char):boolean; forward;


procedure verybusy;

var local : integer;

begin   readln(local);
   outgoing := incoming * local;
        end;                                           {end should be lined up with begin}

function capital;

begin    capital := parameter in ['A'..'Z'];
     end;                                              {end should be lined up with begin}

begin    writeln('Let''s start demonstrating things.');
   readln(first,last);
   if first <= last then
     begin         write(first,' and ',last, ' are');
```

```
      writeln(' in alphabetical order.');
  end;                                          {end should be lined up with begin}

  if first = poundsign then
     high := (100 mod 90)
  else
     high := 20;

  for counter := 1 to limit do
     read(name[counter]);

  case limit div 2 of
     0, 1, 2, 3, 4, 5 : ;
     6, 7, 8, 9: writeln('within range.');
         end;                                   {end should be lined up with case}

  repeat      read(shorts);
until (shorts=1) or (shorts=10);                {end should be lined up with repeat}

  while not eoln do
     begin         read(first);
        writeln(first);
          end;                                  {end should be lined up with begin}

  with onecourse do
     begin
        name := 'study hall';
        units := 5;
        grade := 'p';  end;                     {end should be lined up with begin}

  for count := 5 downto 1 do
     write(' ');
  testing := capital(first);
  verybusy(high,low);
  reset(source);
  read(source,last);
  rewrite(results);
  write(results,last);
  new(list);
  list^.comingword := nil;
  pointer := list;
  goto 1;
1: end.
```

```
{****  Test that there are 2 blank lines between procedures  *****}



{****  This should only be two lines  ****}



program sotypical (input,output);

const  limit = 10;
       poundsign = '#';
       amorcita = 'ilana';

type   hues = (red,blue,green,orange,violet);
       shades = blue..orange;
       smallnumbers = 1..10;
       string = packed array[1..limit] of char;
       class = record  name : string;
                 units : integer;
                 grade, junk : char;
               end;
       grades = array[smallnumbers] of class;
       colorcount=array[1..10,'A'..'Z'] of hues;
       classfile=file of class;
       pastels= set of shades;
       nextword=^sentence;
       sentence = record    currentword : string;
                    comingword  : nextword;
                  end;

var  high, low, counter : integer;
     first, last: char;
     height, weight:real;
     testing, debugging: boolean;
     colors : hues;
     shorts: smallnumbers;
     name : string;
     onecourse : class;
     curriculum: grades;
     colorsquares: colorcount;
     schedule : classfile;
     source, results : text;
     crayons : pastels;
     list, pointer : nextword;
     count,count2,count3,count4,count5,count6 : integer;

label 1;

procedure verybusy (    incoming:integer;
                    var outgoing:integer); forward;

function capital(parameter : char):boolean; forward;




{****  There should only be two lines here  ****}


procedure verybusy;
```

```
var local : integer;

begin   readln(local);
   outgoing := incoming * local;
end;                                        {** There should be two blank lines under this line  **}
function capital;

begin    capital := parameter in ['A'..'Z'];
end;
                                          {** There should be one more blank line under this line **}
begin    writeln('Let''s start demonstrating things.');
   readln(first,last);
   if first <= last then
      begin         write(first,' and ',last, ' are');
         writeln(' in alphabetical order.');
      end;

   if first = poundsign then
      high := (100 mod 90)
   else
      high := 20;

   for counter := 1 to limit do
      read(name[counter]);

   case limit div 2 of
      0, 1, 2, 3, 4, 5 : ;
      6, 7, 8, 9: writeln('within range.');
   end;

   repeat      read(shorts);
   until (shorts=1) or (shorts=10);

   while not eoln do
      begin         read(first);              {Begin Only}
         writeln(first);
      end;

   with onecourse do
      begin                                   {Begin Only - Leave Comment}
         name := 'study hall';
         units := 5;
         grade := 'p'
      end;

   for count := 5 downto 1 do
      write(' ');
   testing := capital(first);
   verybusy(high,low);
   reset(source);
   read(source,last);
   rewrite(results);
   write(results,last);
   new(list);
   list^.comingword := nil;
   pointer := list;
   goto 1;
1: end.
```

```
{****  Test that there is at least one space before and after each ":=", ":", and "="  ****}

program sotypical (input,output);

const  limit=10;
       poundsign='#';
       amorcita='ilana';

type   hues=(red,blue,green,orange,violet);
       shades=blue..orange;
       smallnumbers=1..10;
       string=packed array[1..limit] of char;
       class=record  name:string;                                  {:}
                     units:integer;                                {:}
                     grade, junk:char;                             {:}
                end;
       grades=array[smallnumbers] of class;
       colorcount=array[1..10,'A'..'Z'] of hues;
       classfile=file of class;
       pastels= set of shades;
       nextword=^sentence;
       sentence=record    currentword:string;
                     comingword :nextword;
                  end;

var  high, low, counter:integer;                                   {:}
     first, last:char;                                             {:}
     height, weight:real;                                          {:}
     testing, debugging:boolean;                                   {:}
     colors:hues;                                                  {:}
     shorts:smallnumbers;                                          {:}
     name:string;                                                  {:}
     onecourse:class;                                              {:}
     curriculum:grades;                                            {:}
     colorsquares:colorcount;                                      {:}
     schedule:classfile;                                           {:}
     source, results:text;                                         {:}
     crayons:pastels;                                              {:}
     list, pointer:nextword;                                       {:}
     count,count2,count3,count4,count5,count6:integer;             {:}

label 1;

procedure verybusy (    incoming:integer;                          {:}
                    var outgoing:integer); forward;                {:}

function capital(parameter:char):boolean; forward;                 {:}


procedure verybusy;

var local:integer;                                                 {:}

begin   readln(local);
   outgoing:=incoming * local;                                     {:=}
end;

function capital;

begin    capital:=parameter in ['A'..'Z'];                         {:=}
end;

begin    writeln('Let''s start demonstrating things.');
   readln(first,last);
   if first <= last then
      begin        write(first,' and ',last, ' are');
         writeln(' in alphabetical order.');
      end;
```

```
   if first=poundsign then
      high:=(100 mod 90)                                    {:=}
   else
      high:=20;                                             {:=}
    {:=}

   for counter:=1 to limit do                               {:=}
      read(name[counter]);

   case limit div 2 of
      0, 1, 2, 3, 4, 5:;
      6, 7, 8, 9:writeln('within range.');
   end;

   repeat      read(shorts);
   until (shorts=1) or (shorts=10);

   while not eoln do
      begin         read(first);
         writeln(first);
      end;

   with onecourse do
      begin
         name:='study hall';                                {:=}
         units:=5;                                          {:=}
         grade:='p'                                         {:=}
      end;

   for count:=5 downto 1 do                                 {:=}
      write(' ');
   testing:=capital(first);                                 {:=}
   verybusy(high,low);
   reset(source);
   read(source,last);
   rewrite(results);
   write(results,last);
   new(list);
   list^.comingword:=nil;
   pointer:=list;                                           {:=}
   goto 1;
1:end.
```

```
{****  Test Only One Executable Statement per line  ****}

program sotypical (input,output);

const  limit = 10;
       poundsign = '#';
       amorcita = 'ilana';

type   hues = (red,blue,green,orange,violet);
       shades = blue..orange;
       smallnumbers = 1..10;
       string = packed array[1..limit] of char;
       class = record  name : string;
                 units : integer;
                 grade, junk : char;
               end;
       grades = array[smallnumbers] of class;
       colorcount=array[1..10,'A'..'Z'] of hues;
       classfile=file of class;
       pastels= set of shades;
       nextword=^sentence;
       sentence = record    currentword : string;
                    comingword  : nextword;
                  end;

var  high, low, counter : integer;
     first, last: char;
     height, weight:real;
     testing, debugging: boolean;
     colors : hues;
     shorts: smallnumbers;
     name : string;
     onecourse : class;
     curriculum: grades;
     colorsquares: colorcount;
     schedule : classfile;
     source, results : text;
     crayons : pastels;
     list, pointer : nextword;
     count,count2,count3,count4,count5,count6 : integer;

label 1;

procedure verybusy (    incoming:integer;
                    var outgoing:integer); forward;

function capital(parameter : char):boolean; forward;


procedure verybusy;

var local : integer;

begin   readln(local);   outgoing := incoming * local;      {Line Should Be broke up}
end;

function capital;

begin
   capital := parameter in ['A'..'Z'];
end;



{***  The line below should be broke up into multiple lines  ****}
begin    writeln('Let''s start demonstrating things.');   readln(first,last);   if first <= last then
    begin        write(first,' and ',last, ' are');         writeln(' in alphabetical order.');
end;
```

```
   if first = poundsign then
      high := (100 mod 90)
   else
      high := 20;

{*** The line below should be broke up into multiple lines ***}
   for counter := 1 to limit do      read(name[counter]);

   case limit div 2 of
      0, 1, 2, 3, 4, 5 : ;
      6, 7, 8, 9: writeln('within range.');
   end;

   repeat      read(shorts);
   until (shorts=1) or (shorts=10);

   while not eoln do
   {****  The following line should be broken up  ****}
      begin         read(first);         writeln(first);
      end;

   with onecourse do
      begin
         { The follwing line should be broken up }
         name := 'study hall';         units := 5;         grade := 'p'
      end;

   for count := 5 downto 1 do
{*** The following line should be broke up ***}
      write('  ');      testing  :=  capital(first);      verybusy(high,low);      reset(source);
read(source,last);
   rewrite(results); write(results,last); new(list); list^.comingword := nil; pointer := list;
                                                    {**** This line should have been broke up ***}
   goto 1;
1: end.
```

```
{***** Test indentation *****}

program sotypical (input,output);

const  limit = 10;
poundsign = '#';
amorcita = 'ilana';

         type   hues = (red,blue,green,orange,violet);
         shades = blue..orange;
         smallnumbers = 1..10;
         string = packed array[1..limit] of char;
         class = record
         name : string;
         units : integer;
         grade : char;
         end;
         grades = array[smallnumbers] of class;
         colorcount=array[1..10,'A'..'Z'] of hues;
         classfile=file of class;
         pastels= set of shades;
         nextword=^sentence;
         sentence = record
         currentword : string;
         comingword  : nextword;
         end;

         var  high, low, counter : integer;
         first, last: char;
   height, weight:real;
   testing, debugging: boolean;
   colors : hues;
   shorts: smallnumbers;
   name : string;
   onecourse : class;
   curriculum: grades;
   colorsquares: colorcount;
   schedule : classfile;
   source, results : text;
   crayons : pastels;
   list, pointer : nextword;
   count : integer;

   label 1;

   procedure verybusy (    incoming:integer;
   var outgoing:integer); forward;

   function capital(parameter : char):boolean; forward;


   procedure verybusy;

   var local : integer;

   begin
        readln(local);
outgoing := incoming * local;
end;

function capital;

begin
capital := parameter in ['A'..'Z'];
end;

begin
writeln('Let''s start demonstrating things.');
readln(first,last);
```

```
if first <= last then
begin
write(first,' and ',last, ' are');
writeln(' in alphabetical order.');
end;

if first = poundsign then
high := (100 mod 90)
else
high := 20;

for counter := 1 to limit do
read(name[counter]);

case limit div 2 of
0, 1, 2, 3, 4, 5 : ;
6, 7, 8, 9: writeln('within range.');
end;

repeat
read(shorts);
until (shorts=1) or (shorts=10);

while not eoln do
begin
read(first);
writeln(first);
end;

with onecourse do
begin
name := 'study hall';
units := 5;
grade := 'p'
end;

for count := 5 downto 1 do
write(' ');
testing := capital(first);
verybusy(high,low);
reset(source);
read(source,last);
   rewrite(results);
   write(results,last);
   new(list);
   list^.comingword := nil;
   pointer := list;
   goto 1;
   1: end.
```

```
{*****  Test that declarations within an indented body should be aligned.  *****}
{*****  For example, line up all variable declarations indented under a    *****}
{*****  VAR statement.  Also line up all the statements indented under an  *****}
{*****  IF-THEN-ELSE statement.                                            *****}


program sotypical (input,output);

const  limit = 10;
poundsign = '#';
amorcita = 'ilana';

type   hues = (red,blue,green,orange,violet);
shades = blue..orange;
smallnumbers = 1..10;
string = packed array[1..limit] of char;
class = record
name : string;
units : integer;
grade : char;
end;
grades = array[smallnumbers] of class;
colorcount=array[1..10,'A'..'Z'] of hues;
classfile=file of class;
pastels= set of shades;
nextword=^sentence;
sentence = record
currentword : string;
comingword  : nextword;
end;

var  high, low, counter : integer;
first, last: char;
height, weight:real;
testing, debugging: boolean;
colors : hues;
shorts: smallnumbers;
name : string;
onecourse : class;
curriculum: grades;
colorsquares: colorcount;
schedule : classfile;
source, results : text;
crayons : pastels;
list, pointer : nextword;
count : integer;

label 1;

procedure verybusy (    incoming:integer;
var outgoing:integer); forward;

function capital(parameter : char):boolean; forward;


procedure verybusy;

var local : integer;

begin
readln(local);
outgoing := incoming * local;
end;

function capital;

begin
capital := parameter in ['A'..'Z'];
end;
```

```
begin
writeln('Let''s start demonstrating things.');
readln(first,last);
if first <= last then
begin
write(first,' and ',last, ' are');
writeln(' in alphabetical order.');
end;

if first = poundsign then
high := (100 mod 90)
else
high := 20;

for counter := 1 to limit do
read(name[counter]);

case limit div 2 of
0, 1, 2, 3, 4, 5 : ;
6, 7, 8, 9: writeln('within range.');
end;

repeat
read(shorts);
until (shorts=1) or (shorts=10);

while not eoln do
begin
read(first);
writeln(first);
end;

with onecourse do
begin
name := 'study hall';
units := 5;
grade := 'p'
end;

for count := 5 downto 1 do
write(' ');
testing := capital(first);
verybusy(high,low);
reset(source);
read(source,last);
rewrite(results);
write(results,last);
new(list);
list^.comingword := nil;
pointer := list;
goto 1;
1: end.
```

```
{****  The PROGRAM statement, CONST, TYPE, VAR keywords, BEGIN, and END ****}
{****  of the main program should be aligned at the left margin.       ****}

     program sotypical (input,output);                    {PROGRAM}

     const  limit = 10;                                   {CONST}
     poundsign = '#';
     amorcita = 'ilana';

     type   hues = (red,blue,green,orange,violet);        {TYPE}
     shades = blue..orange;
     smallnumbers = 1..10;
     string = packed array[1..limit] of char;
     class = record
     name : string;
     units : integer;
     grade : char;
     end;
     grades = array[smallnumbers] of class;
     colorcount=array[1..10,'A'..'Z'] of hues;
     classfile=file of class;
     pastels= set of shades;
     nextword=^sentence;
     sentence = record
     currentword : string;
     comingword  : nextword;
     end;

     var  high, low, counter : integer;                   {VAR}
     first, last: char;
     height, weight:real;
     testing, debugging: boolean;
     colors : hues;
     shorts: smallnumbers;
     name : string;
     onecourse : class;
     curriculum: grades;
     colorsquares: colorcount;
     schedule : classfile;
     source, results : text;
     crayons : pastels;
     list, pointer : nextword;
     count : integer;

     label 1;

     procedure verybusy (    incoming:integer;
     var outgoing:integer); forward;

     function capital(parameter : char):boolean; forward;


     procedure verybusy;

     var local : integer;

     begin
     readln(local);
     outgoing := incoming * local;
     end;

     function capital;

     begin
     capital := parameter in ['A'..'Z'];
     end;

     begin                                                {BEGIN}
     writeln('Let''s start demonstrating things.');
```

```
readln(first,last);
if first <= last then
begin
write(first,' and ',last, ' are');
writeln(' in alphabetical order.');
end;

if first = poundsign then
high := (100 mod 90)
else
high := 20;

for counter := 1 to limit do
read(name[counter]);

case limit div 2 of
0, 1, 2, 3, 4, 5 : ;
6, 7, 8, 9: writeln('within range.');
end;

repeat
read(shorts);
until (shorts=1) or (shorts=10);

while not eoln do
begin
read(first);
writeln(first);
end;

with onecourse do
begin
name := 'study hall';
units := 5;
grade := 'p'
end;

for count := 5 downto 1 do
write(' ');
testing := capital(first);
verybusy(high,low);
reset(source);
read(source,last);
rewrite(results);
write(results,last);
new(list);
list^.comingword := nil;
pointer := list;
goto 1;
1: end.                                          {END}
```

```
{*****  The decleatation keywords (CONST,TYPE,VAR), and BEGIN-END  *****}
{*****  blocks of procedure and functions should be aligned with   *****}
{*****  the procedure headings.  Procedures/functions that are     *****}
{*****  physically within another procedure/function (not the main *****}
{*****  program) should be indented and any declaration keywords   *****}
{*****  aligned with the appropriate headings.                     *****}

    program sotypical (input,output);

    const  limit = 10;
    poundsign = '#';
    amorcita = 'ilana';

    type   hues = (red,blue,green,orange,violet);
    shades = blue..orange;
    smallnumbers = 1..10;
    string = packed array[1..limit] of char;
    class = record
    name : string;
    units : integer;
    grade : char;
    end;
    grades = array[smallnumbers] of class;
    colorcount=array[1..10,'A'..'Z'] of hues;
    classfile=file of class;
    pastels= set of shades;
    nextword=^sentence;
    sentence = record
    currentword : string;
    comingword  : nextword;
    end;

    var  high, low, counter : integer;
    first, last: char;
    height, weight:real;
    testing, debugging: boolean;
    colors : hues;
    shorts: smallnumbers;
    name : string;
    onecourse : class;
    curriculum: grades;
    colorsquares: colorcount;
    schedule : classfile;
    source, results : text;
    crayons : pastels;
    list, pointer : nextword;
    count : integer;

    label 1;

    procedure verybusy (    incoming:integer;
    var outgoing:integer); forward;

    function capital(parameter : char):boolean; forward;


    procedure verybusy;                 {--- Should be indented ---}

    var local : integer;                {--- Should be aligned with proc }

    begin                               {--- Should be aligned with proc }
    readln(local);                      {--- Should be indented from proc }
    outgoing := incoming * local;
    end;

    function capital;                   {--- Should be indented ---}

    begin                               {--- Should be aligned with proc ---}
    capital := parameter in ['A'..'Z']; {--- Should be indented from proc }
```

```
end;

begin
writeln('Let''s start demonstrating things.');
readln(first,last);
if first <= last then
begin
write(first,' and ',last, ' are');
writeln(' in alphabetical order.');
end;

if first = poundsign then
high := (100 mod 90)
else
high := 20;

for counter := 1 to limit dc
read(name[counter]);

case limit div 2 of
0, 1, 2, 3, 4, 5 : ;
6, 7, 8, 9: writeln('within range.');
end;

repeat
read(shorts);
until (shorts=1) or (shorts=10);

while not eoln do
begin
read(first);
writeln(first);
end;

with onecourse do
begin
name := 'study hall';
units := 5;
grade := 'p'
end;

for count := 5 downto 1 do
write(' ');
testing := capital(first);
verybusy(high,low);
reset(source);
read(source,last);
rewrite(results);
write(results,last);
new(list);
list^.comingword := nil;
pointer := list;
goto 1;
1: end.
```

```
{*****  Test that declarations within an indented body should be aligned.  *****}
{*****  For example, line up all variable declarations indented under a    *****}
{*****  VAR statement.  Also line up all the statements indented under an  *****}
{*****  IF-THEN-ELSE statement.                                            *****}


program sotypical (input,output);

const  limit = 10;
poundsign = '#';
amorcita = 'ilana';

type   hues = (red,blue,green,orange,violet);
shades = blue..orange;
smallnumbers = 1..10;
string = packed array[1..limit] of char;
class = record
name : string;
units : integer;
grade : char;
end;
grades = array[smallnumbers] of class;
colorcount=array[1..10,'A'..'Z'] of hues;
classfile=file of class;
pastels= set of shades;
nextword=^sentence;
sentence = record
currentword : string;
comingword  : nextword;
end;

var  high, low, counter : integer;
first, last: char;
height, weight:real;
testing, debugging: boolean;
colors : hues;
shorts: smallnumbers;
name : string;
onecourse : class;
curriculum: grades;
colorsquares: colorcount;
schedule : classfile;
source, results : text;
crayons : pastels;
list, pointer : nextword;
count : integer;

label 1;

procedure verybusy (    incoming:integer;
var outgoing:integer); forward;

function capital(parameter : char):boolean; forward;


procedure verybusy;

var local : integer;

begin
readln(local);
outgoing := incoming * local;
end;

function capital;

begin
capital := parameter in ['A'..'Z'];
end;
```

```
begin
writeln('Let''s start demonstrating things.');
readln(first,last);
if first <= last then
begin
write(first,' and ',last, ' are');
writeln(' in alphabetical order.');
end;

if first = poundsign then
high := (100 mod 90)
else
high := 20;

for counter := 1 to limit do
read(name[counter]);

case limit div 2 of
0, 1, 2, 3, 4, 5 : ;
6, 7, 8, 9: writeln('within range.');
end;

repeat
read(shorts);
until (shorts=1) or (shorts=10);

while not eoln do
begin
read(first);
writeln(first);
end;

with onecourse do
begin
name := 'study hall';
units := 5;
grade := 'p'
end;

for count := 5 downto 1 do
write(' ');
testing := capital(first);
verybusy(high,low);
reset(source);
read(source,last);
rewrite(results);
write(results,last);
new(list);
list^.comingword := nil;
pointer := list;
goto 1;
1: end.
```

```
{*****  Test that declarations within an indented body should be aligned.      *****}
{*****  Test that the body of all IF-THEN, IF-THEN-ELSE, WHILE, WITH, and      *****}
{*****  CASE statements should be indented from their corresponding keywords.  *****}

program sotypical (input,output);

const  limit = 10;
poundsign = '#';
amorcita = 'ilana';

type   hues = (red,blue,green,orange,violet);
shades = blue..orange;
smallnumbers = 1..10;
string = packed array[1..limit] of char;
class = record                              {--- This should be indented  }
name : string;
units : integer;
grade : char;
end;
grades = array[smallnumbers] of class;
colorcount=array[1..10,'A'..'Z'] of hues;
classfile=file of class;
pastels= set of shades;
nextword=^sentence;
sentence = record
currentword : string;
comingword  : nextword;
end;

var  high, low, counter : integer;
first, last: char;
height, weight:real;
testing, debugging: boolean;
colors : hues;
shorts: smallnumbers;
name : string;
onecourse : class;
curriculum: grades;
colorsquares: colorcount;
schedule : classfile;
source, results : text;
crayons : pastels;
list, pointer : nextword;
count : integer;

label 1;

procedure verybusy (    incoming:integer;
var outgoing:integer); forward;

function capital(parameter : char):boolean; forward;


procedure verybusy;

var local : integer;

begin
readln(local);
outgoing := incoming * local;
end;

function capital;

begin
capital := parameter in ['A'..'Z'];
end;

begin
```

```
writeln('Let''s start demonstrating things.');
readln(first,last);
if first <= last then
begin                                   {--- This should be indented }
write(first,' and ',last, ' are');      {--- This should be indented }
writeln(' in alphabetical order.');
end;

if first = poundsign then
high := (100 mod 90)                    {--- This should be indented }
else
high := 20;                             {--- This should be indented }

for counter := 1 to limit do
read(name[counter]);                    {--- This should be indented }

case limit div 2 of
0, 1, 2, 3, 4, 5 : ;
6, 7, 8, 9: writeln('within range.');
end;

repeat
read(shorts);
until (shorts=1) or (shorts=10);

while not eoln do
begin                                   {--- This should be indented }
read(first);                            {--- This should be indented }
writeln(first);
end;

with onecourse do
begin                                   {--- This should be indented }
name := 'study hall';                   {--- This should be indetned }
units := 5;
grade := 'p'
end;

for count := 5 downto 1 do
write(' ');                             {--- This should be indented }
testing := capital(first);
verybusy(high,low);
reset(source);
read(source,last);
rewrite(results);
write(results,last);
new(list);
list^.comingword := nil;
pointer := list;
goto 1;
1: end.
```

```
{**** If a body of a FOR, IF-THEN, IF-THEN-ELSE, WHILE, or WITH     *****}
{**** statement is a compound statement (more than one command),    *****}
{**** then the BEGIN should follow the keyword on the next line and *****}
{**** the END should be on a line by itself aligned with the        *****}
{**** corresponding BEGIN.  When a REPEAT loop appears on more      *****}
{**** than one line, the UNTIL is aligned with the REPEAT.          *****}


program sotypical (input,output);

const  limit = 10;
poundsign = '#';
amorcita = 'ilana';

type   hues = (red,blue,green,orange,violet);
shades = blue..orange;
smallnumbers = 1..10;
string = packed array[1..limit] of char;
class = record
name : string;
units : integer;
grade : char;
end;
grades = array[smallnumbers] of class;
colorcount=array[1..10,'A'..'Z'] of hues;
classfile=file of class;
pastels= set of shades;
nextword=^sentence;
sentence = record
currentword : string;
comingword  : nextword;
end;

var  high, low, counter : integer;
first, last: char;
height, weight:real;
testing, debugging: boolean;
colors : hues;
shorts: smallnumbers;
name : string;
onecourse : class;
curriculum: grades;
colorsquares: colorcount;
schedule : classfile;
source, results : text;
crayons : pastels;
list, pointer : nextword;
count : integer;

label 1;

procedure verybusy (    incoming:integer;
var outgoing:integer); forward;

function capital(parameter : char):boolean; forward;


procedure verybusy;

var local : integer;

begin
readln(local);
outgoing := incoming * local;
end;

function capital;

begin
```

```
capital := parameter in ['A'..'Z'];
end;

begin
writeln('Let''s start demonstrating things.');
readln(first,last);
if first <= last then
begin                                   {--- This should be indented  }
write(first,' and ',last, ' are');      {--- This should be indented  }
writeln(' in alphabetical order.');
end;

if first = poundsign then
high := (100 mod 90)
else
high := 20;

for counter := 1 to limit do
read(name[counter]);

case limit div 2 of
0, 1, 2, 3, 4, 5 : ;
6, 7, 8, 9: writeln('within range.');
end;

repeat
read(shorts);                           {--- This should be indented  }
until (shorts=1) or (shorts=10);        {--- This should be lined up with the Repeat  }

while not eoln do
begin                                   {--- This should be indented  }
read(first);                            {--- This should be indented  }
writeln(first);
end;

with onecourse do
begin                                   {--- This should be indented  }
name := 'study hall';                   {--- This should be indetned  }
units := 5;
grade := 'p'
end;

for count := 5 downto 1 do
write(' ');
testing := capital(first);
verybusy(high,low);
reset(source);
read(source,last);
rewrite(results);
write(results,last);
new(list);
list^.comingword := nil;
pointer := list;
goto 1;
1: end.
```

```
{**** If a body of a FOR, IF-THEN, IF-THEN-ELSE, WHILE, or WITH     *****}
{**** statement is a compound statement (more than one command),    *****}
{**** then the BEGIN should follow the keyword on the next line and *****}
{**** the END should be on a line by itself aligned with the        *****}
{**** corresponding BEGIN.  When a REPEAT loop appears on more      *****}
{**** than one line, the UNTIL is aligned with the REPEAT.          *****}

program sotypical (input,output);
                                {*****  These Lines Should all be commented  *****}
const  limit = 10;{--- Comment 1}
poundsign = '#';  {--- Comment 2}
amorcita = 'ilana'; {--- Comment 3}

       {Comment}
type   hues = (red,blue,green,orange,violet);
       {Comment}
shades = blue..orange;
       {Comment}
smallnumbers = 1..10;
       {Comment}
string = packed array[1..limit] of char;
       {Comment}
class = record
       {Comment}
name : string;
       {Comment}
units : integer;
       {Comment}
grade : char;
end;
       {Comment}
grades = array[smallnumbers] of class;   {Comment}
colorcount=array[1..10,'A'..'Z'] of hues;   {Comment}
classfile=file of class;   {Comment}
pastels= set of shades;   {Comment}
nextword=^sentence;   {Comment}
sentence = record   {Comment}
currentword : string;   {Comment}
comingword  : nextword;   {Comment}
end;   {Comment}

var  high, low, counter : integer;   {Comment}
first, last: char;   {Comment}
height, weight:real;   {Comment}
testing, debugging: boolean;   {Comment}
colors : hues;   {Comment}
shorts: smallnumbers;   {Comment}
name : string;   {Comment}
onecourse : class;   {Comment}
curriculum: grades;   {Comment}
colorsquares: colorcount;   {Comment}
schedule : classfile;   {Comment}
source, results : text;   {Comment}
crayons : pastels;   {Comment}
list, pointer : nextword;   {Comment}
count : integer;   {Comment}

label 1;

procedure verybusy (    incoming:integer;
var outgoing:integer); forward;   {Comment}

function capital(parameter : char):boolean; forward;   {Comment}

procedure verybusy;
var local : integer;   {Comment}

begin
readln(local);
```

```
outgoing := incoming * local;
end;

function capital;

begin
capital := parameter in ['A'..'Z'];
end;

begin
writeln('Let''s start demonstrating things.');
readln(first,last);
if first <= last then
begin                                   {--- This should be indented  }
write(first,' and ',last, ' are');      {--- This should be indented  }
writeln(' in alphabetical order.');
end;

if first = poundsign then
high := (100 mod 90)
else
high := 20;

for counter := 1 to limit do
read(name[counter]);

case limit div 2 of
0, 1, 2, 3, 4, 5 : ;
6, 7, 8, 9: writeln('within range.');
end;

repeat
read(shorts);                           {--- This should be indented  }
until (shorts=1) or (shorts=10);        {--- This should be lined up with the Repeat  }

while not eoln do
begin                                   {--- This should be indented  }
read(first);                            {--- This should be indented  }
writeln(first);
end;

with onecourse do
begin                                   {--- This should be indented  }
name := 'study hall';                   {--- This should be indetned  }
units := 5;
grade := 'p'
end;

for count := 5 downto 1 do
write(' ');   {Comment}
testing := capital(first);   {Comment}
verybusy(high,low);   {Comment}
reset(source);   {Comment}
read(source,last);   {Comment}
rewrite(results);   {Comment}
write(results,last);   {Comment}
new(list);   {Comment}
list^.comingword := nil;   {Comment}
pointer := list;   {Comment}
goto 1;   {Comment}
1: end.   {Comment}
```

APPENDIX F

SPSSX DISCRIMINANT ANALYSIS CODE AND RELEVANT RESULTS

ALL TEAMS FOR BOTH GROUPS

ALL VARIABLES

```
19-May-90   SPSS-X RELEASE 3.1 FOR VAX/VMS
              Page   1
16:32:20    STUDENT ACCESS NETWORK SPSS-X  on UCBEH::          VMS V5.3

VAX                    STUDENT ACCESS NETWORK SPSS-X     License Number 19638
This software is functional through June 30, 1990.

Try the new SPSS-X Release 3.0 and 3.1 features:

* Interactive SPSS-X command execution          * The new RANK procedure
* Online,VMS-like Help                          * Improvements in:
* Nonlinear Regression                          *   REPORT and TABLES
* Time Series and Forecasting (TRENDS)          *   Simplified Syntax
* Macro Facility                                *   Matrix I/O

See SPSS-X User's Guide, Third Edition, for more information on these features.

   1  0  DATA LIST FILE='ALLDATA.DAT' RECORDS=3
   2  0   /1 ID 1-2  GROUP 1 REC 4 CLARITY 6-13 EFFORT 15-22 LOOPS 24-28 SELECTS 30-34
   3  0      n1 36-40 n2 42-46 CALLS 48-52 DATADIFF 54-58 DIFFICUL 60-65 BLOCKS 67-71
   4  0   /2 MODULES 12-16 LOC 18-22 CMMNTS 24-28 LENGTHN 30-34 ESTN 36-40
   5  0      IMPLEVEL 42-46 VOLUME 48-52 VOCAB 54-58
   6  0   /3 GLOGON 12-16 GCOMPL 18-22 GLINKS 24-28 GRUNS 30-34 GTIME 36-40 TOTTIME 42-46
   7  0      TOTDES 48-52 TOTCOD 54-58

This command will read 3 records from SYS$STAFF:[FACULTY.GRANGER.STATS]ALLDATA.DAT;

Preceding task required .28 seconds CPU time;  .37 seconds elapsed.

   9  0  DISCRIMINANT   GROUPS=GROUP(1,2)/
  10  0    VARIABLES = CLARITY  TO  TOTCOD /
  11  0    ANALYSIS = CLARITY TO TOTCOD /
  12  0    METHOD = WILKS /  PIN = .05 /
  13  0    CLASSIFY = POOLED /
  14  0    STATISTICS = MEAN STDDEV CORR FPAIR UNIVF BOXM RAW COEFF TABLE /
  15  0    PLOT = ALL

There are 12,008,608 bytes of memory available.

This DISCRIMINANT analysis requires 23408 bytes of memory.
```

EXHIBIT F-1

SPSSX - DISCRIMINANT ANALYSIS COMMANDS

GROUP MEANS

GROUP	CLARITY	EFFORT	LOOPS	SELECTS	N1	N2
CALLS	DATADIFF					
1	2078688.57143	6401098.85714	61.57143	181.85714	34.14286	181.28571
69.14286	10.63429					
2	1413530.00000	3848281.75000	56.50000	112.00000	33.25000	165.50000
45.50000	7.92750					
TOTAL	1836812.72727	5472801.72727	59.72727	156.45455	33.81818	175.54545
60.54545	9.65000					

GROUP	DIFFICUL	BLOCKS	MODULES	LOC	CMNTS	LENGTHN
ESTN	IMPLEVEL					
1	180.47429	311.85714	34.00000	1751.00000	188.14286	4655.71429
1567.71429	0.00607					
2	132.29250	211.75000	25.75000	1437.50000	221.00000	3694.75000
1398.25000	0.00803					
TOTAL	162.95364	275.45455	31.00000	1637.00000	200.09091	4306.27273
1506.09091	0.00678					

GROUP	VOLUME	VOCAB	GLOGON	GCOMPL	GLINKS	GRUNS
GTIME	TOTTIME					
1	36553.71429	215.42857	220.28571	3912.71429	1057.85714	1023.14286
210.85714	204.42857					
2	28439.25000	198.75000	165.00000	1652.50000	559.75000	536.25000
164.00000	73.50000					
TOTAL	33603.00000	209.36364	200.18182	3090.81818	876.72727	846.09091
193.81818	156.81818					

GROUP	TOTDES	TOTCOD
1	81.71429	122.71429
2	50.00000	23.50000
TOTAL	70.18182	86.63636

EXHIBIT F-2

GROUP MEANS - ALL VARIABLES - ALL GROUPS

WILKS' LAMBDA (U-STATISTIC) AND UNIVARIATE F-RATIO
WITH 1 AND 9 DEGREES OF FREEDOM

VARIABLE	WILKS' LAMBDA	F	SIGNIFICANCE
CLARITY	0.92103	0.7717	0.4025
EFFORT	0.83289	1.806	0.2119
LOOPS	0.99582	0.3781E-01	0.8501
SELECTS	0.83251	1.811	0.2113
N1	0.94306	0.5434	0.4798
N2	0.99138	0.7825E-01	0.7860
CALLS	0.90893	0.9017	0.3671
DATADIFF	0.84468	1.655	0.2304
DIFFICUL	0.81052	2.104	0.1809
BLOCKS	0.87728	1.259	0.2909
MODULES	0.85167	1.567	0.2421
LOC	0.89442	1.062	0.3296
CMMNTS	0.98567	0.1308	0.7259
LENGTHN	0.93813	0.5935	0.4608
ESTN	0.98823	0.1072	0.7508
IMPLEVEL	0.81586	2.031	0.1878
VOLUME	0.94836	0.4900	0.5016
VOCAB	0.99060	0.8540E-01	0.7767
GLOGON	0.74897	3.016	0.1164
GCOMPL	0.65827	4.672	0.0589
GLINKS	0.52503	8.142	0.0190
GRUNS	0.51551	8.458	0.0174
GTIME	0.58576	6.365	0.0326
TOTTIME	0.74003	3.162	0.1091
TOTDES	0.78162	2.515	0.1473
TOTCOD	0.78897	2.407	0.1552

EXHIBIT F-3

STATISTICS FOR ALL VARIABLES - ALL GROUPS

Preceding task required 4.56 seconds CPU time; 6.35 seconds elapsed.

```
  16  0  DISCRIMINANT  GROUPS=GROUP(1,2)/
  17  0    VARIABLES = CLARITY TO blocks,loc /
  18  0    ANALYSIS = CLARITY TO blocks,loc /
  19  0    METHOD = WILKS / PIN = .05 /
  20  0    CLASSIFY = POOLED /
  21  0    STATISTICS = MEAN STDDEV CORR FPAIR UNIVF BOXM RAW COEFF TABLE /
  22  0    PLOT = ALL
```

There are 12,008,464 bytes of memory available.

This DISCRIMINANT analysis requires 4688 bytes of memory.

WILKS' LAMBDA (U-STATISTIC) AND UNIVARIATE F-RATIO
WITH 1 AND 9 DEGREES OF FREEDOM

VARIABLE	WILKS' LAMBDA	F	SIGNIFICANCE
CLARITY	0.92103	0.7717	0.4025
EFFORT	0.83289	1.806	0.2119
LOOPS	0.99582	0.3781E-01	0.8501
SELECTS	0.83251	1.811	0.2113
H1	0.94306	0.5434	0.4798
H2	0.99138	0.7825E-01	0.7860
CALLS	0.90893	0.9017	0.3671
DATADIFF	0.84468	1.655	0.2304
DIFFICUL	0.81052	2.104	0.1809
BLOCKS	0.87725	1.259	0.2909
LOC	0.89442	1.062	0.3296

EXHIBIT F-4

SPSSX - DISCRIMINANT ANALYSIS COMMANDS - COMPLEXITY

Preceding task required .54 seconds CPU time; 1.32 seconds elapsed.

```
  23  0  DISCRIMINANT   GROUPS=GROUP(1,2)/
  24  0    VARIABLES = modules to vocab /
  25  0    ANALYSIS = modules TO vocab /
  26  0    METHOD = WILKS /  PIN = .05 /
  27  0    CLASSIFY = POOLED /
  28  0    STATISTICS = MEAN STDDEV CORR FPAIR UNIVF BOXM RAW COEFF TABLE /
  29  0    PLOT = ALL
```

There are 12,008,576 bytes of memory available.

This DISCRIMINANT analysis requires 2200 bytes of memory.

WILKS' LAMBDA (U-STATISTIC) AND UNIVARIATE F-RATIO
WITH 1 AND 9 DEGREES OF FREEDOM

VARIABLE	WILKS' LAMBDA	F	SIGNIFICANCE
MODULES	0.85167	1.567	0.2421
LOC	0.89442	1.062	0.3296
CMMNTS	0.98567	0.1308	0.7259
LENGTHN	0.93813	0.5935	0.4608
ESTN	0.98823	0.1072	0.7508
IMPLEVEL	0.81586	2.031	0.1878
VOLUME	0.94836	0.4900	0.5016
VOCAB	0.99060	0.8540E-01	0.7767

EXHIBIT F-5

SPSSX - DISCRIMINANT ANALYSIS COMMANDS - SIZE

```
Preceding task required .44 seconds CPU time;  .91 seconds elapsed.

  30  0  DISCRIMINANT  GROUPS=GROUP(1,2)/
  31  0    VARIABLES = glogon to TOTCOD /
  32  0    ANALYSIS = glogon to TOTCOD /
  33  0    METHOD = WILKS /  PIN = .05 /
  34  0    CLASSIFY = POOLED /
  35  0    STATISTICS = MEAN STDDEV CORR FPAIR UNIVF BOXM RAW COEFF TABLE /
  36  0    PLOT = ALL

There are 12,008,560 bytes of memory available.

This DISCRIMINANT analysis requires 2688 bytes of memory.
```

WILKS' LAMBDA (U-STATISTIC) AND UNIVARIATE F-RATIO
WITH 1 AND 9 DEGREES OF FREEDOM

VARIABLE	WILKS' LAMBDA	F	SIGNIFICANCE
GLOGON	0.74897	3.016	0.1164
GCOMPL	0.65827	4.672	0.0589
GLINKS	0.52503	8.142	0.0190
GRUNS	0.51551	8.458	0.0174
GTIME	0.58576	6.365	0.0326
TOTTIME	0.74003	3.162	0.1091
TOTDES	0.78162	2.515	0.1473
TOTCOD	0.78897	2.407	0.1552

EXHIBIT F-6

SPSSX - DISCRIMINANT ANALYSIS COMMANDS - TIME

APPENDIX G

SPSSX DISCRIMINANT ANALYSIS CODE AND RELEVANT RESULTS

WITHOUT TEAM 3 FROM THE TREATMENT GROUP (GROUP 2)

ALL VARIABLES

```
19-May-90   SPSS-X RELEASE 3.1 FOR VAX/VMS
              Page   1
16:32:52    STUDENT ACCESS NETWORK SPSS-X  on UCBLH::          VMS V5.3

VAX                    STUDENT ACCESS NETWORK SPSS-X     License Number 19638
This software is functional through June 30, 1990.

Try the new SPSS-X Release 3.0 and 3.1 features:

* Interactive SPSS-X command execution          * The new RANK procedure
* Online,VMS-like Help                          * Improvements in:
* Nonlinear Regression                          *   REPORT and TABLES
* Time Series and Forecasting (TRENDS)          *   Simplified Syntax
* Macro Facility                                *   Matrix I/O

See SPSS-X User's Guide, Third Edition, for more information on these features.

   1  0  DATA LIST FILE='SPRZL.DAT' RECORDS=3
   2  0   /1 ID 1-2  GROUP 1 REC 4 CLARITY 6-13 EFFORT 15-22 LOOPS 24-28 SELECTS 30-34
   3  0      n1 36-40 n2 42-46 CALLS 48-52 DATADIFF 54-58 DIFFICUL 60-65 BLOCKS 67-71
   4  0   /2 MODULES 12-16 LOC 18-22 CMMNTS 24-28 LENGTHN 30-34 ESTN 36-40
   5  0      IMPLEVEL 42-46 VOLUME 48-52 VOCAB 54-58
   6  0   /3 GLOGON 12-16 GCOMPL 18-22 GLINKS 24-28 GRUNS 30-34 GTIME 36-40 TOTTME 42-46
   7  0      TOTDES 48-52 TOTCOD 54-58

This command will read 3 records from SYSSSTAFF:[FACULTY.GRANGER.STATS]SPRZL.DAT;

Preceding task required .25 seconds CPU time;  .32 seconds elapsed.

   9  0  DISCRIMINANT   GROUPS=GROUP(1,2)/
  10  0    VARIABLES = CLARITY  TO  TOTCOD /
  11  0    ANALYSIS = CLARITY TO TOTCOD /
  12  0    METHOD = WILKS /  FIN = .05 /
  13  0    CLASSIFY = POOLED /
  14  0    STATISTICS = MEAN STDDEV CORR FPAIR UNIVF BOXM RAW COEFF TABLE /
  15  0    PLOT = ALL

There are 12,008,608 bytes of memory available.


This DISCRIMINANT analysis requires 21712 bytes of memory.
```

EXHIBIT G-1

SPSSX - DISCRIMINANT ANALYSIS COMMANDS

GROUP MEANS

GROUP	CLARITY	EFFORT	LOOPS	SELECTS	N1	N2
CALLS	DATADIFF					
1	2078688.57143	6401098.85714	61.57143	181.85714	34.14286	181.28571
69.14286	10.63429					
2	1447880.66667	3799994.00000	67.00000	112.33333	34.00000	170.66667
36.33333	7.53333					
TOTAL	1889446.20000	5620767.40000	63.20000	161.00000	34.10000	178.10000
59.30000	9.70400					

GROUP	DIFFICUL	BLOCKS	MODULES	LOC	CMMNTS	LENGTHN
ESTN	IMPLEVEL					
1	180.47429	311.85714	34.00000	1751.00000	188.14286	4655.71429
1567.71429	0.00607					
2	129.34000	212.66667	26.66667	1436.66667	224.66667	3669.00000
1452.00000	0.00833					
TOTAL	165.13400	282.10000	31.80000	1656.70000	199.10000	4359.70000
1533.00000	0.00675					

GROUP	VOLUME	VOCAB	GLOGON	GCOMPL	GLINKS	GRUNS
GTIME	TOTTIME					
1	36553.71429	215.42857	220.28571	3912.71429	1057.85714	1023.14286
210.85714	204.42857					
2	28489.33333	204.66667	167.00000	1528.33333	717.00000	686.33333
170.66667	78.66667					
TOTAL	34134.40000	212.20000	204.30000	3197.40000	955.60000	922.10000
198.80000	166.70000					

GROUP	TOTDES	TOTCOD
1	81.71429	122.71429
2	51.66667	27.00000
TOTAL	72.70000	94.00000

EXHIBIT G-2

GROUP MEANS - ALL VARIABLES

WITHOUT TEAM 3 FROM TREATMENT GROUP (GROUP 2)

WILKS' LAMBDA (U-STATISTIC) AND UNIVARIATE F-RATIO
WITH 1 AND 8 DEGREES OF FREEDOM

VARIABLE	WILKS' LAMBDA	F	SIGNIFICANCE
CLARITY	0.94012	0.5095	0.4956
EFFORT	0.85331	1.375	0.2746
LOOPS	0.99568	0.3471E-01	0.8568
SELECTS	0.85881	1.315	0.2846
N1	0.99841	0.1277E-01	0.9128
N2	0.99675	0.2608E-01	0.8757
CALLS	0.85372	1.371	0.2754
DATADIFF	0.83137	1.623	0.2385
DIFFICUL	0.82093	1.745	0.2230
BLOCKS	0.89822	0.9065	0.3689
MODULES	0.89711	0.9175	0.3662
LOC	0.91083	0.7832	0.4020
CMMNTS	0.98539	0.1186	0.7394
LENGTHN	0.94574	0.4590	0.5172
ESTN	0.99541	0.3686E-01	0.8525
IMPLEVEL	0.79592	2.051	0.1900
VOLUME	0.95752	0.3549	0.5678
VOCAB	0.99673	0.2622E-01	0.8754
GLOGON	0.79529	2.059	0.1892
GCOMPL	0.67560	3.841	0.0857
GLINKS	0.62195	4.863	0.0585
GRUNS	0.60946	5.126	0.0534
GTIME	0.68480	3.682	0.0913
TOTTIME	0.78859	2.145	0.1812
TOTDES	0.82804	1.661	0.2334
TOTCOD	0.82940	1.646	0.2355

EXHIBIT G-3

STATISTICS FOR ALL VARIABLES

WITHOUT TEAM 3 FROM TREATMENT GROUP (GROUP 2)

Preceding task required 3.93 seconds CPU time; 5.01 seconds elapsed.

```
  16  0  DISCRIMINANT  GROUPS=GROUP(1,2)/
  17  0    VARIABLES = CLARITY TO blocks,loc /
  18  0    ANALYSIS = CLARITY TO blocks,loc /
  19  0    METHOD = WILKS / PIN = .05 /
  20  0    CLASSIFY = POOLED /
  21  0    STATISTICS = MEAN STDDEV CORR FPAIR UNIVF BOXM RAW COEFF TABLE /
  22  0    PLOT = ALL
```

There are 12,008,464 bytes of memory available.

This DISCRIMINANT analysis requires 4688 bytes of memory.

WILKS' LAMBDA (U-STATISTIC) AND UNIVARIATE F-RATIO
WITH 1 AND 8 DEGREES OF FREEDOM

VARIABLE	WILKS' LAMBDA	F	SIGNIFICANCE
CLARITY	0.94012	0.5095	0.4956
EFFORT	0.85331	1.375	0.2746
LOOPS	0.99568	0.3471E-01	0.8568
SELECTS	0.85881	1.315	0.2846
N1	0.99841	0.1277E-01	0.9128
N2	0.99675	0.2608E-01	0.8757
CALLS	0.85372	1.371	0.2754
DATADIFF	0.83137	1.623	0.2385
DIFFICUL	0.82093	1.745	0.2230
BLOCKS	0.89822	0.9065	0.3689
LOC	0.91083	0.7832	0.4020

EXHIBIT G-4

SPSSX - DISCRIMINANT ANALYSIS COMMANDS - COMPLEXITY

WITHOUT TEAM 3 FROM TREATMENT GROUP (GROUP 2)

```
19-May-90   SPSS-X RELEASE 3.1 FOR VAX/VMS
              Page  17
16:32:59    STUDENT ACCESS NETWORK SPSS-X  on UCBEH::          VMS V5.3


Preceding task required .53 seconds CPU time;  1.26 seconds elapsed.

  23  0  DISCRIMINANT   GROUPS=GROUP(1,2)/
  24  0    VARIABLES = modules to vocab /
  25  0    ANALYSIS = modules TO vocab /
  26  0    METHOD = WILKS /  FIN = .05 /
  27  0    CLASSIFY = POOLED /
  28  0    STATISTICS = MEAN STDDEV CORR FPAIR UNIVF BOXM RAW COEFF TABLE /
  29  0    PLOT = ALL

There are 12,008,576 bytes of memory available.


This DISCRIMINANT analysis requires 2200 bytes of memory.



WILKS' LAMBDA (U-STATISTIC) AND UNIVARIATE F-RATIO
WITH   1 AND         8 DEGREES OF FREEDOM
  VARIABLE   WILKS' LAMBDA         F          SIGNIFICANCE
  --------   -------------   -------------   ------------

  MODULES        0.89711         0.9175          0.3662
  LOC            0.91083         0.7832          0.4020
  CMMNTS         0.98539         0.1186          0.7394
  LENGTHN        0.94574         0.4590          0.5172
  ESTN           0.99541        0.3686E-01       0.8525
  IMPLEVEL       0.79592          2.051          0.1900
  VOLUME         0.95752         0.3549          0.5678
  VOCAB          0.99673        0.2622E-01       0.8754
```

EXHIBIT G-5

SPSSX - DISCRIMINANT ANALYSIS COMMANDS - SIZE

WITHOUT TEAM 3 FROM TREATMENT GROUP (GROUP 2)

Preceding task required .45 seconds CPU time; .88 seconds elapsed.

```
  30  0  DISCRIMINANT  GROUPS=GROUP(1,2)/
  31  0    VARIABLES = glogon to TOTCOD /
  32  0    ANALYSIS = glogon to TOTCOD /
  33  0    METHOD = WILKS / PIN = .05 /
  34  0    CLASSIFY = POOLED /
  35  0    STATISTICS = MEAN STDDEV CORR FPAIR UNIVF BOXM RAW COEFF TABLE /
  36  0    PLOT = ALL
```

There are 12,008,560 bytes of memory available.

This DISCRIMINANT analysis requires 2688 bytes of memory.

WILKS' LAMBDA (U-STATISTIC) AND UNIVARIATE F-RATIO
WITH 1 AND 8 DEGREES OF FREEDOM

VARIABLE	WILKS' LAMBDA	F	SIGNIFICANCE
GLOGON	0.79529	2.059	0.1892
GCOMPL	0.67560	3.841	0.0857
GLINKS	0.62195	4.863	0.0585
GRUNS	0.60946	5.126	0.0534
GTIME	0.68480	3.682	0.0913
TOTTIME	0.78859	2.145	0.1812
TOTDES	0.82804	1.661	0.2334
TOTCOD	0.82940	1.646	0.2355

EXHIBIT G-6

SPSSX - DISCRIMINANT ANALYSIS COMMANDS - TIME

WITHOUT TEAM 3 FROM TREATMENT GROUP (GROUP 2)

APPENDIX H

SPSSX DISCRIMINANT ANALYSIS CODE AND RELEVANT RESULTS

ALL TEAMS FOR BOTH GROUPS

ALL VARIABLES

TRANSFORMED TO Z SCORES

COMBINED TO COMPUTE P VALUES FOR COMPLEXITY,

SIZE AND TIME

```
19-May-90   SPSS-X RELEASE 3.1 FOR VAX/VMS
              Page   1
17:04:54    STUDENT ACCESS NETWORK SPSS-X  on UCBEH::          VMS V5.3

VAX                    STUDENT ACCESS NETWORK SPSS-X     License Number 19638
This software is functional through June 30, 1990.

Try the new SPSS-X Release 3.0 and 3.1 features:

* Interactive SPSS-X command execution          * The new RANK procedure
* Online,VMS-like Help                          * Improvements in:
* Nonlinear Regression                          *   REPORT and TABLES
* Time Series and Forecasting (TRENDS)          *   Simplified Syntax
* Macro Facility                                *   Matrix I/O

See SPSS-X User's Guide, Third Edition, for more information on these features.

   1  0  DATA LIST FILE='ALLDATA.DAT' RECORDS=3
   2  0   /1 ID 1-2  GROUP 1 REC 4 CLARITY 6-13 EFFORT 15-22 LOOPS 24-28 SELECTS 30-34
   3  0      n1 36-40 n2 42-46 CALLS 48-52 DATADIFF 54-58 DIFFICUL 60-65 BLOCKS 67-71
   4  0   /2 MODULES 12-16 LOC 18-22 CMNTS 24-28 LENGTHN 30-34 ESTN 36-40
   5  0      IMPLEVEL 42-46 VOLUME 48-52 VOCAB 54-58
   6  0   /3 GLOGON 12-16 GCOMPL 18-22 GLINKS 24-28 GRUNS 30-34 GTIME 36-40 TOTTIME 42-46
   7  0      TOTDES 48-52 TOTCOD 54-58
   8  0
   9  0

This command will read 3 records from SYSSSTAFF:[FACULTY.GRANGER.STATS]ALLDATA.DAT;

  10  0  DESCRIPTIVES   VARIABLES= CLARITY  EFFORT  LOOPS SELECTS n1  n2  CALLS
  11  0          DATADIFF DIFFICUL BLOCKS MODULES LOC  CMNTS LENGTHN  ESTN
  12  0          IMPLEVEL VOLUME  VOCAB GLOGON  GCOMPL GLINKS  GRUNS  GTIME
  13  0          TOTTIME TOTDES TOTCOD /
  14  0    SAVE  /
  15  0    STATISTICS = ALL
  16  0
  17  0

There are 12,007,712 bytes of memory available.


1,976 bytes of memory required for the DESCRIPTIVES procedure.
52 bytes have already been acquired.
1,924 bytes remain to be acquired.
```

EXHIBIT H-1

SPSSX - DISCRIMINANT ANALYSIS COMMANDS

The following Z-Score variables have been saved on your active file:

From Variable	To Z-Score	Label	Weighted Valid N
CLARITY	ZCLARITY	Zscore(CLARITY)	11
EFFORT	ZEFFORT	Zscore(EFFORT)	11
LOOPS	ZLOOPS	Zscore(LOOPS)	11
SELECTS	ZSELECTS	Zscore(SELECTS)	11
N1	ZN1	Zscore(N1)	11
N2	ZN2	Zscore(N2)	11
CALLS	ZCALLS	Zscore(CALLS)	11
DATADIFF	ZDATADIF	Zscore(DATADIFF)	11
DIFFICUL	ZDIFFICU	Zscore(DIFFICUL)	11
BLOCKS	ZBLOCKS	Zscore(BLOCKS)	11
MODULES	ZMODULES	Zscore(MODULES)	11
LOC	ZLOC	Zscore(LOC)	11
CMMNTS	ZCMMNTS	Zscore(CMMNTS)	11
LENGTHN	ZLENGTHN	Zscore(LENGTHN)	11
ESTN	ZESTN	Zscore(ESTN)	11
IMPLEVEL	ZIMPLEVE	Zscore(IMPLEVEL)	11
VOLUME	ZVOLUME	Zscore(VOLUME)	11
VOCAB	ZVOCAB	Zscore(VOCAB)	11
GLOGON	ZGLOGON	Zscore(GLOGON)	11
GCOMPL	ZGCOMPL	Zscore(GCOMPL)	11
GLINKS	ZGLINKS	Zscore(GLINKS)	11
GRUNS	ZGRUNS	Zscore(GRUNS)	11
GTIME	ZGTIME	Zscore(GTIME)	11
TOTTIME	ZTOTTIME	Zscore(TOTTIME)	11
TOTDES	ZTOTDES	Zscore(TOTDES)	11
TOTCOD	ZTOTCOD	Zscore(TOTCOD)	11

EXHIBIT H-2

Z SCORES FOR ALL VARIABLES

```
19-May-90   SPSS-X RELEASE 3.1 FOR VAX/VMS
              Page  11
17:04:57    STUDENT ACCESS NETWORK SPSS-X  on UCHEH::          VMS V5.3


Preceding task required .64 seconds CPU time;  1.22 seconds elapsed.

  18  0  LIST VARIABLES=GROUP ZCLARITY  ZEFFORT  ZLOOPS ZSELECTS Zn1  Zn2  ZCALLS
  19  0          ZDATADIFF ZDIFFICUL ZBLOCKS  ZMODULES  ZLOC  ZCOMMTS ZLENGTHN  ZESTN
  20  0          ZIMPLEVEL ZVOLUME  ZVOCAB ZGLOGON  ZGCOMPL ZGLINKS  ZGRUNS  ZGTIME
  21  0          ZIOTTIME ZIOTDES ZIOTCOD /
  22  0

There are 12,009,568 bytes of memory available.


1,387 bytes of memory required for the LIST procedure.
496 bytes have already been acquired.
891 bytes remain to be acquired.
THE VARIABLES ARE LISTED IN THE FOLLOWING ORDER:

LINE   1: GROUP ZCLARITY ZEFFORT ZLOOPS ZSELECTS ZN1 ZN2 ZCALLS ZDATADIF ZDIFFICU ZBLOCKS

LINE   2: ZMODULES ZLOC ZCOMMTS ZLENGTHN ZESTN ZIMPLEVE ZVOLUME ZVOCAB ZGLOGON ZGCOMPL

LINE   3: ZGLINKS ZGRUNS ZGTIME ZIOTTIME ZIOTDES ZIOTCOD




  23  0  COMPUTE ZCOMPL = ZEFFORT + ZLOOPS + ZSELECTS + ZN1 + ZN2 +
  24  0               ZCALLS + ZBLOCKS + zclarity + zloc + zdatadif + zdifficu
  25  0  COMPUTE ZSIZE = ZMODULES + ZLOC + ZLENGTHN + ZVOLUME + ZVOCAB + zcomments +
  26  0               zestn + zimplevo
  27  0  COMPUT ZTIME = ZGRUNS + ZGTIME + ZIOTTIME + ZIOTDES + ZIOTCOD + zglogon +
  28  0               zgcompl + zglinks
  29  0
  30  0  DISCRIMINANT   GROUPS=GROUP (1,2) /
  31  0     VARIABLES = ZCOMPL, ztime, zsize /
  32  0     ANALYSIS = ZCOMPL, ztime, zsize /
  33  0     METHOD = WILKES / FIN = .05 /
  34  0     CLASSIFY = POOLED/
  35  0     STATISTICS = MEAN STDDEV CORR FPAIR UNIVF BOXM RAW COEFF TABLE/
  36  0     PLOT = ALL /
  37  0
  38  0

There are 12,007,072 bytes of memory available.


This DISCRIMINANT analysis requires 1692 bytes of memory.
```

EXHIBIT H-3

RELEVANT SPSSX COMMANDS

COMBINE Z SCORES FOR EACH CATEGORY

WILKS' LAMBDA (U-STATISTIC) AND UNIVARIATE F-RATIO
WITH 1 AND 9 DEGREES OF FREEDOM

VARIABLE	WILKS' LAMBDA	F	SIGNIFICANCE
ZCOMPL	0.83230	1.813	0.2110
ZTIME	0.39809	13.61	0.0050
ZSIZE	0.97657	0.2159	0.6532

EXHIBIT H-4

SIGNIFICANCE VALUES FOR EACH CATEGORY

APPENDIX I

SPSSX - RELIABILITY - ALPHA MODEL

GROUP 1 - CONTROL GROUP

```
19-May-90   SPSS-X RELEASE 3.1 FOR VAX/VMS
              Page   1
17:30:09    STUDENT ACCESS NETWORK SPSS-X  on UCBEH::          VMS V5.3

VAX                     STUDENT ACCESS NETWORK SPSS-X     License Number 19638
This software is functional through June 30, 1990.

Try the new SPSS-X Release 3.0 and 3.1 features:

* Interactive SPSS-X command execution          * The new RANK procedure
* Online,VMS-like Help                          * Improvements in:
* Nonlinear Regression                          *   REPORT and TABLES
* Time Series and Forecasting (TRENDS)          *   Simplified Syntax
* Macro Facility                                *   Matrix I/O

See SPSS-X User's Guide, Third Edition, for more information on these features.

   1  0  DATA LIST FILE='ALLDATA.DAT' RECORDS=3
   2  0    /1 ID 1-2  GROUP 1 REC 4 CLARITY 6-13 EFFORT 15-22 LOOPS 24-28 SELECTS 30-34
   3  0       n1 36-40 n2 42-46 CALLS 48-52 DATADIFF 54-58 DIFFICUL 60-65 BLOCKS 67-71
   4  0    /2 MODULES 12-16 LOC 18-22 CMMNTS 24-28 LENGTHN 30-34 ESTN 36-40
   5  0       IMPLEVEL 42-46 VOLUME 48-52 VOCAB 54-58
   6  0    /3 GLOGON 12-16 GCOMPL 18-22 GLINKS 24-28 GRUNS 30-34 GTIME 36-40 TOTTIME 42-46
   7  0       TOTDES 48-52 TOTCOD 54-58

This command will read 3 records from SYS$STAFF:[FACULTY.GRANGER.STATS]ALLDATA.DAT;



   9  0  SELECT IF (GROUP EQ 1)
  10  0  DESCRIPTIVES   VARIABLES= CLARITY  EFFORT  LOOPS SELECTS n1  n2  CALLS
  11  0          DATADIFF DIFFICUL BLOCKS MODULES LOC  CMMNTS LENGTHN  ESTN
  12  0          IMPLEVEL VOLUME  VOCAB GLOGON  GCOMPL GLINKS  GRUNS  GTIME
  13  0          TOTTME TOTDES TOTCOD /
  14  0    SAVE  /
  15  0    STATISTICS = ALL
```

EXHIBIT I-1

SPSSX - DISCRIMINANT ANALYSIS COMMANDS

SELECTING GROUP 1 (CONTROL GROUP)

RELIABILITY ANALYSIS

The following Z-Score variables have been saved on your active file:

From Variable	To Z-Score	Label	Weighted Valid N
CLARITY	ZCLARITY	Zscore(CLARITY)	7
EFFORT	ZEFFORT	Zscore(EFFORT)	7
LOOPS	ZLOOPS	Zscore(LOOPS)	7
SELECTS	ZSELECTS	Zscore(SELECTS)	7
N1	ZN1	Zscore(N1)	7
N2	ZN2	Zscore(N2)	7
CALLS	ZCALLS	Zscore(CALLS)	7
DATADIFF	ZDATADIF	Zscore(DATADIFF)	7
DIFFICUL	ZDIFFICU	Zscore(DIFFICUL)	7
BLOCKS	ZBLOCKS	Zscore(BLOCKS)	7
MODULES	ZMODULES	Zscore(MODULES)	7
LOC	ZLOC	Zscore(LOC)	7
CMMNTS	ZCMMNTS	Zscore(CMMNTS)	7
LENGTHN	ZLENGTHN	Zscore(LENGTHN)	7
ESTN	ZESTN	Zscore(ESTN)	7
IMPLEVEL	ZIMPLEVE	Zscore(IMPLEVEL)	7
VOLUME	ZVOLUME	Zscore(VOLUME)	7
VOCAB	ZVOCAB	Zscore(VOCAB)	7
GLOGON	ZGLOGON	Zscore(GLOGON)	7
GCOMPL	ZGCOMPL	Zscore(GCOMPL)	7
GLINKS	ZGLINKS	Zscore(GLINKS)	7
GRUNS	ZGRUNS	Zscore(GRUNS)	7
GTIME	ZGTIME	Zscore(GTIME)	7
TOTTME	ZTOTTME	Zscore(TOTTME)	7
TOTDES	ZTOTDES	Zscore(TOTDES)	7
TOTCOD	ZTOTCOD	Zscore(TOTCOD)	7

EXHIBIT I-2

Z SCORES FOR ALL VARIABLES

CONTROL GROUP

```
19-May-90   SPSS-X RELEASE 3.1 FOR VAX/VMS
              Page  11
17:30:12    STUDENT ACCESS NETWORK SPSS-X  on UCBEH::          VMS V5.3

Preceding task required .57 seconds CPU time;  1.07 seconds elapsed.

  16  0  LIST
  17  0

There are 12,009,568 bytes of memory available.



Preceding task required .18 seconds CPU time;  .19 seconds elapsed.

  18  0  RELIABILITY   VARIABLES= ZCLARITY  ZEFFORT  ZLOOPS ZSELECTS Zn1  Zn2  ZCALLS
  19  0          ZDATADIFF ZDIFFICUL ZBLOCKS  ZMODULES  ZLOC  ZCMMNTS ZLENGTHN  ZESTN
  20  0          ZIMPLEVEL ZVOLUME  ZVOCAB ZGLOGON  ZGCOMPL ZGLINKS  ZGRUNS  ZGTIME
  21  0          ZTOTTME ZTOTDES ZTOTCOD /
  22  0
  23  0   SCALE(COMPLEX) = ZCLARITY  ZEFFORT  ZLOOPS ZSELECTS Zn1  Zn2  ZCALLS
  24  0          ZDATADIFF ZDIFFICUL ZBLOCKS /
  25  0
  26  0   SCALE(SIZE)= ZMODULES  ZLOC  ZCMMNTS ZLENGTHN  ZESTN
  27  0          ZIMPLEVEL ZVOLUME  ZVOCAB /
  28  0
  29  0   SCALE(TIME) = ZGLOGON  ZGCOMPL ZGLINKS  ZGRUNS  ZGTIME
  30  0          ZTOTTME ZTOTDES ZTOTCOD /
  31  0
  32  0    MODEL=ALPHA /
  33  0    SUMMARY = CORE TOTAL
  34  0
```

EXHIBIT I-3

SPSSX - RELIABILITY COMMANDS

COMBINING VARIABLES INTO CATEGORIES

CONTROL GROUP

RELIABILITY ANALYSIS - SCALE (COMPLEX)

1.	ZCLARITY	Zscore(CLARITY)
2.	ZEFFORT	Zscore(EFFORT)
3.	ZLOOPS	Zscore(LOOPS)
4.	ZSELECTS	Zscore(SELECTS)
5.	ZN1	Zscore(N1)
6.	ZN2	Zscore(N2)
7.	ZCALLS	Zscore(CALLS)
8.	ZDATADIF	Zscore(DATADIFF)
9.	ZDIFFICU	Zscore(DIFFICUL)
10.	ZBLOCKS	Zscore(BLOCKS)

ITEM-TOTAL STATISTICS

	SCALE MEAN IF ITEM DELETED	SCALE VARIANCE IF ITEM DELETED	CORRECTED ITEM-TOTAL CORRELATION	SQUARED MULTIPLE CORRELATION	ALPHA IF ITEM DELETED
ZCLARITY	.0000	34.8304	.9301	.	.8343
ZEFFORT	.0000	34.2710	.9854	.	.8296
ZLOOPS	.0000	39.4717	.5043	.	.8685
ZSELECTS	.0000	34.7797	.9351	.	.8339
ZN1	.0000	39.5461	.4979	.	.8690
ZN2	.0000	36.6111	.7600	.	.8484
ZCALLS	.0000	36.9617	.7276	.	.8511
ZDATADIF	.0000	47.2042	-.1016	.	.9105
ZDIFFICU	.0000	46.1836	-.0276	.	.9038
ZBLOCKS	.0000	34.6089	.9519	.	.8324

RELIABILITY ANALYSIS - SCALE (COMPLEX)

RELIABILITY COEFFICIENTS 10 ITEMS

ALPHA = .8737 STANDARDIZED ITEM ALPHA = .8737

EXHIBIT I-4

ALPHA LEVELS FOR COMPLEXITY VARIABLES

CONTROL GROUP

R E L I A B I L I T Y A N A L Y S I S - S C A L E (S I Z E)

1. ZMODULES Zscore(MODULES)
2. ZLOC Zscore(LOC)
3. ZCMMNTS Zscore(CMMNTS)
4. ZLENGTHN Zscore(LENGTHN)
5. ZESTN Zscore(ESTN)
6. ZIMPLEVE Zscore(IMPLEVEL)
7. ZVOLUME Zscore(VOLUME)
8. ZVOCAB Zscore(VOCAB)

ITEM-TOTAL STATISTICS

	SCALE MEAN IF ITEM DELETED	SCALE VARIANCE IF ITEM DELETED	CORRECTED ITEM-TOTAL CORRELATION	SQUARED MULTIPLE CORRELATION	ALPHA IF ITEM DELETED
ZMODULES	.0000	23.6616	.1444	.	.8215
ZLOC	.0000	20.0063	.5656	.	.7585
ZCMMNTS	.0000	26.1398	-.1050	.	.8542
ZLENGTHN	.0000	17.6010	.8897	.	.7027
ZESTN	.0000	17.2309	.9438	.	.6927
ZIMPLEVE	.0000	25.0265	.0040	.	.8403
ZVOLUME	.0000	17.4529	.9112	.	.6987
ZVOCAB	.0000	17.2783	.9368	.	.6940

RELIABILITY COEFFICIENTS 8 ITEMS

ALPHA = .7921 STANDARDIZED ITEM ALPHA = .7921

EXHIBIT I-5

ALPHA LEVELS FOR SIZE VARIABLES

CONTROL GROUP

R E L I A B I L I T Y A N A L Y S I S - S C A L E (T I M E)

1. ZGLOGON Zscore(GLOGON)
2. ZGCOMPL Zscore(GCOMPL)
3. ZGLINKS Zscore(GLINKS)
4. ZGRUNS Zscore(GRUNS)
5. ZGTIME Zscore(GTIME)
6. ZTOTTME Zscore(TOTTME)
7. ZTOTDES Zscore(TOTDES)
8. ZTOTCOD Zscore(TOTCOD)

ITEM-TOTAL STATISTICS

	SCALE MEAN IF ITEM DELETED	SCALE VARIANCE IF ITEM DELETED	CORRECTED ITEM-TOTAL CORRELATION	SQUARED MULTIPLE CORRELATION	ALPHA IF ITEM DELETED
ZGLOGON	.0000	21.7978	-.1371	.	.7920
ZGCOMPL	.0000	21.0941	-.0627	.	.7795
ZGLINKS	.0000	14.9072	.7266	.	.6188
ZGRUNS	.0000	14.8701	.7323	.	.6175
ZGTIME	.0000	14.9754	.7161	.	.6213
ZTOTTME	.0000	14.9327	.7226	.	.6198
ZTOTDES	.0000	19.8333	.0768	.	.7549
ZTOTCOD	.0000	14.6959	.7593	.	.6110

RELIABILITY COEFFICIENTS 8 ITEMS

ALPHA = .7180 STANDARDIZED ITEM ALPHA = .7180

EXHIBIT I-6

ALPHA LEVELS FOR TIME VARIABLES

CONTROL GROUP

APPENDIX J

SPSSX - RELIABILITY - ALPHA MODEL

GROUP 2 - TREATMENT GROUP

```
19-May-90   SPSS-X RELEASE 3.1 FOR VAX/VMS
              Page   1
17:30:54    STUDENT ACCESS NETWORK SPSS-X  on UCBEH::          VMS V5.3

VAX                     STUDENT ACCESS NETWORK SPSS-X     License Number 19638
This software is functional through June 30, 1990.

Try the new SPSS-X Release 3.0 and 3.1 features:

* Interactive SPSS-X command execution          * The new RANK procedure
* Online,VMS-like Help                          * Improvements in:
* Nonlinear Regression                          *   REPORT and TABLES
* Time Series and Forecasting (TRENDS)          *   Simplified Syntax
* Macro Facility                                *   Matrix I/O

  See SPSS-X User's Guide, Third Edition, for more information on these features.

   1  0  DATA LIST FILE='SPRZL.DAT' RECORDS=3
   2  0   /1 ID 1-2  GROUP 1 REC 4 CLARITY 6-13 EFFORT 15-22 LOOPS 24-28 SELECTS 30-34
   3  0      n1 36-40 n2 42-46 CALLS 48-52 DATADIFF 54-58 DIFFICUL 60-65 BLOCKS 67-71
   4  0   /2 MODULES 12-16 LOC 18-22 CMMNTS 24-28 LENGTHN 30-34 ESTN 36-40
   5  0      IMPLEVEL 42-46 VOLUME 48-52 VOCAB 54-58
   6  0   /3 GLOGON 12-16 GCOMPL 18-22 GLINKS 24-28 GRUNS 30-34 GTIME 36-40 TOTTME 42-46
   7  0      TOTDES 48-52 TOTCOD 54-58



   8  0  LIST

   9  0  SELECT IF (GROUP EQ 2)
  10  0  DESCRIPTIVES   VARIABLES= CLARITY  EFFORT  LOOPS SELECTS n1  n2  CALLS
  11  0          DATADIFF DIFFICUL BLOCKS MODULES LOC  CMMNTS LENGTHN  ESTN
  12  0          IMPLEVEL VOLUME  VOCAB GLOGON  GCOMPL GLINKS  GRUNS  GTIME
  13  0          TOTTME TOTDES TOTCOD /
  14  0    SAVE  /
  15  0    STATISTICS = ALL
```

EXHIBIT J-1

SPSSX - DISCRIMINANT ANALYSIS COMMANDS

SELECTING GROUP 2 (TREATMENT GROUP)

RELIABILITY ANALYSIS

The following Z-Score variables have been saved on your active file:

From Variable	To Z-Score	Label	Weighted Valid N
CLARITY	ZCLARITY	Zscore(CLARITY)	4
EFFORT	ZEFFORT	Zscore(EFFORT)	4
LOOPS	ZLOOPS	Zscore(LOOPS)	4
SELECTS	ZSELECTS	Zscore(SELECTS)	4
N1	ZN1	Zscore(N1)	4
N2	ZN2	Zscore(N2)	4
CALLS	ZCALLS	Zscore(CALLS)	4
DATADIFF	ZDATADIF	Zscore(DATADIFF)	4
DIFFICUL	ZDIFFICU	Zscore(DIFFICUL)	4
BLOCKS	ZBLOCKS	Zscore(BLOCKS)	4
MODULES	ZMODULES	Zscore(MODULES)	4
LOC	ZLOC	Zscore(LOC)	4
CMMNTS	ZCMMNTS	Zscore(CMMNTS)	4
LENGTHN	ZLENGTHN	Zscore(LENGTHN)	4
ESTN	ZESTN	Zscore(ESTN)	4
IMPLEVEL	ZIMPLEVE	Zscore(IMPLEVEL)	4
VOLUME	ZVOLUME	Zscore(VOLUME)	4
VOCAB	ZVOCAB	Zscore(VOCAB)	4
GLOGON	ZGLOGON	Zscore([illegible])	4
GCOMPL	ZGCOMPL	Zscore(GCOMPL)	4
GLINKS	ZGLINKS	Zscore(GLINKS)	4
GRUNS	ZGRUNS	Zscore(GRUNS)	4
GTIME	ZGTIME	Zscore(GTIME)	4
TOTTME	ZTOTTME	Zscore(TOTTME)	4
TOTDES	ZTOTDES	Zscore(TOTDES)	4
TOTCOD	ZTOTCOD	Zscore(TOTCOD)	4

EXHIBIT J-2

Z SCORES FOR ALL VARIABLES

TREATMENT GROUP

```
 16  0  LIST
 17  0
 18  0  RELIABILITY   VARIABLES= ZCLARITY  ZEFFORT  ZLOOPS ZSELECTS Zn1  Zn2  ZCALLS
 19  0         ZDATADIFF ZDIFFICUL ZBLOCKS  ZMODULES  ZLOC  ZCMMNTS ZLENGTHN  ZESTN
 20  0         ZIMPLEVEL ZVOLUME  ZVOCAB ZGLOGON  ZGCOMPL ZGLINKS  ZGRUNS  ZGTIME
 21  0         ZTOTTME ZTOTDES ZTOTCOD /
 22  0
 23  0   SCALE(COMPLEX) = ZCLARITY  ZEFFORT  ZLOOPS ZSELECTS Zn1  Zn2  ZCALLS
 24  0         ZDATADIFF ZDIFFICUL ZBLOCKS /
 25  0
 26  0   SCALE(SIZE)= ZMODULES  ZLOC  ZCMMNTS ZLENGTHN  ZESTN
 27  0         ZIMPLEVEL ZVOLUME  ZVOCAB /
 28  0
 29  0   SCALE(TIME) = ZGLOGON  ZGCOMPL ZGLINKS  ZGRUNS  ZGTIME
 30  0         ZTOTTME ZTOTDES ZTOTCOD /
 31  0
 32  0    MODEL=ALPHA /
 33  0    SUMMARY = CORE TOTAL
 34  0
```

EXHIBIT J-3

SPSSX - RELIABILITY COMMANDS

COMBINING VARIABLES INTO CATEGORIES

TREATMENT GROUP

R E L I A B I L I T Y A N A L Y S I S - S C A L E (C O M P L E X)

1.	ZCLARITY	Zscore(CLARITY)
2.	ZEFFORT	Zscore(EFFORT)
3.	ZLOOPS	Zscore(LOOPS)
4.	ZSELECTS	Zscore(SELECTS)
5.	ZN1	Zscore(N1)
6.	ZN2	Zscore(N2)
7.	ZCALLS	Zscore(CALLS)
8.	ZDATADIF	Zscore(DATADIFF)
9.	ZDIFFICU	Zscore(DIFFICUL)
10.	ZBLOCKS	Zscore(BLOCKS)

ITEM-TOTAL STATISTICS

	SCALE MEAN IF ITEM DELETED	SCALE VARIANCE IF ITEM DELETED	CORRECTED ITEM-TOTAL CORRELATION	SQUARED MULTIPLE CORRELATION	ALPHA IF ITEM DELETED
ZCLARITY	.0000	42.3944	.9706	.	.8862
ZEFFORT	.0000	42.1246	.9945	.	.8846
ZLOOPS	.0000	44.9192	.7546	.	.8996
ZSELECTS	.0000	42.8346	.9320	.	.8886
ZN1	.0000	45.5014	.7066	.	.9025
ZN2	.0000	45.2087	.7306	.	.9010
ZCALLS	.0000	51.2265	.2660	.	.9273
ZDATADIF	.0000	51.7163	.2307	.	.9292
ZDIFFICU	.0000	49.4713	.3954	.	.9203
ZBLOCKS	.0000	42.8755	.9284	.	.8889

R E L I A B I L I T Y A N A L Y S I S - S C A L E (C O M P L E X)

RELIABILITY COEFFICIENTS 10 ITEMS

ALPHA = .9128 STANDARDIZED ITEM ALPHA = .9128

EXHIBIT J-4

ALPHA LEVELS FOR COMPLEXITY VARIABLES

TREATMENT GROUP

RELIABILITY ANALYSIS - SCALE (SIZE)

1. ZMODULES Zscore(MODULES)
2. ZLOC Zscore(LOC)
3. ZCMMNTS Zscore(CMMNTS)
4. ZLENGTHN Zscore(LENGTHN)
5. ZESTN Zscore(ESTN)
6. ZIMPLEVE Zscore(IMPLEVEL)
7. ZVOLUME Zscore(VOLUME)
8. ZVOCAB Zscore(VOCAB)

ITEM-TOTAL STATISTICS

	SCALE MEAN IF ITEM DELETED	SCALE VARIANCE IF ITEM DELETED	CORRECTED ITEM-TOTAL CORRELATION	SQUARED MULTIPLE CORRELATION	ALPHA IF ITEM DELETED
ZMODULES	.0000	32.9313	.9587	.	.9187
ZLOC	.0000	32.8053	.9715	.	.9177
ZCMMNTS	.0000	32.8345	.9685	.	.9179
ZLENGTHN	.0000	33.8963	.8620	.	.9257
ZESTN	.0000	32.5624	.9964	.	.9159
ZIMPLEVE	.0000	46.5069	-.1886	.	.9911
ZVOLUME	.0000	33.5072	.9006	.	.9229
ZVOCAB	.0000	32.5600	.9966	.	.9158

RELIABILITY COEFFICIENTS 8 ITEMS

ALPHA = .9394 STANDARDIZED ITEM ALPHA = .9394

EXHIBIT J-5

ALPHA LEVELS FOR SIZE VARIABLES

TREATMENT GROUP

R E L I A B I L I T Y A N A L Y S I S - S C A L E (T I M E)

1. ZGLOGON Zscore(GLOGON)
2. ZGCOMPL Zscore(GCOMPL)
3. ZGLINKS Zscore(GLINKS)
4. ZGRUNS Zscore(GRUNS)
5. ZGTIME Zscore(GTIME)
6. ZTOTTME Zscore(TOTTME)
7. ZTOTDES Zscore(TOTDES)
8. ZTOTCOD Zscore(TOTCOD)

ITEM-TOTAL STATISTICS

	SCALE MEAN IF ITEM DELETED	SCALE VARIANCE IF ITEM DELETED	CORRECTED ITEM-TOTAL CORRELATION	SQUARED MULTIPLE CORRELATION	ALPHA IF ITEM DELETED
ZGLOGON	.0000	27.6293	-.5984	.	.8711
ZGCOMPL	.0000	24.6702	-.3354	.	.8356
ZGLINKS	.0000	15.7535	.7036	.	.6483
ZGRUNS	.0000	15.6943	.7124	.	.6463
ZGTIME	.0000	13.9657	.9865	.	.5819
ZTOTTME	.0000	14.7427	.8590	.	.6127
ZTOTDES	.0000	15.3575	.7631	.	.6349
ZTOTCOD	.0000	14.2194	.9440	.	.5923

RELIABILITY COEFFICIENTS 8 ITEMS

ALPHA = .7336 STANDARDIZED ITEM ALPHA = .7336

EXHIBIT J-6

ALPHA LEVELS FOR TIME VARIABLES

TREATMENT GROUP