AN INVESTIGATION OF SOFTWARE METRICS AFFECT ON COBOL PROGRAM RELIABILITY

by

Henry Jesse Day, II

Dissertation submitted to the Faculty of the Virginia Polytechnic Institute and State University in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Business

with a Major in Accounting

James O. Hicks, Chairman
Professor of Accounting
Virginia Tech

Robert M. Brown
William S. Gay, Professor of Accounting

Wayne E. Leininger
Professor of Accounting

Tarun K. Sen
Associate Professor of Accounting

Kai S. Koong
Associate Professor
Mount Olive College

June 20, 1996
Blacksburg, Virginia

AN INVESTIGATION OF SOFTWARE METRICS AFFECT ON COBOL PROGRAM RELIABILITY

by

Henry Jesse Day, II

James O. Hicks, Chairman

Department of Accounting

(ABSTRACT)

The purpose of this research was to predict a COBOL program's reliability from software characteristics that are found in the program's source code. The first step was to select factors based on the human information processing model that are associated with changes in computer program reliability. Then these factors (software metrics) were quantitatively studied to determine which factors affect COBOL program reliability. Then a statistical model was developed that predicts COBOL program reliability. Reliability was selected because the reliability of computer programs can be used by systems professionals and auditors to make decisions.

Using the Human Information Processing Model to study the act of creating a computer program, several hypotheses were derived about program characteristics and reliability. These hypotheses were categorized as size, structure, and temporal hypotheses. These characteristics were then used to test several prediction models for the reliability of COBOL programs.

Program characteristics were measured by a program called METRICS. METRICS was written by the author using the Pascal programming language. It accepts COBOL programs as input and produces as output seventeen measures of complexity.

Actual programs and related data were then gathered from a large insurance company over the course of one year. The data were used to test the hypotheses and to find a model for predicting the reliability of COBOL programs. The operational definition for reliability was the probability of a

program executing without abending. The size of a program, its cyclomatic complexity, and the number of times a program has been executed were used to predict reliability. A regression model was developed that predicted the reliability of a COBOL program from a program's characteristics. The model had a prediction error of 9.3%, a R^2 of 15%, and an adjusted R^2 of 13%. The most important thing learned from the research is that increasing the size of a program's modules, not the total size of a program, is associated with decreased reliability.

Acknowledgments

I am grateful to Professors James O. Hicks and Robert M. Brown for their invaluable assistance and guidance throughout this dissertation. In particular Professor Hicks originated the idea for the research and helped me obtain data, and Professor Brown helped me analyze the results. I would also like to thank Wayne E. Leininger, Tarun K. Sen, and Kai S. Koong for their critiques and advice.

I am grateful to Warren Hutcheson from Tazewell High School who inspired my desire to learn about computers and to go to college and on to graduate study. I am grateful to Doug Hess, CPA in Tazewell, Virginia for counseling me and encouraging me to go into accounting. I am indebted to the Department of Accounting at Virginia Tech which provided the opportunity for me to become an accounting and information systems educator and researcher.

I am most grateful to my parents Henry and Eleanor Day who have encouraged me throughout my life. So, this dissertation is dedicated to my parents who made my life and everything I do possible.

Table of Contents

Introduction 1
1.1 Historical Development of Software Quality 1
1.2 The Need for Software Reliability Measurement 3
1.3 Development of Research Purpose 7

Literature Review 9
2.1 Failure Models 9
2.2 Reliability Prediction from Design Characteristics 14
2.2.1 Software Metrics 15
2.2.2 Complexity Studies 20

Model Development 28
3.1 Characteristics of COBOL 28
3.2 Development of Theoretical Model 30
3.2.1 Structured Programming 35
3.2.2 Complexity Metrics 36
3.3 Fault Maintenance Model 44
3.4 Development of Hypotheses 46
3.4.1 Hypothesis One 46
3.4.2 Hypothesis Two 47
3.4.3 Hypothesis Three 47
3.4.4 Hypothesis Four 48
3.4.5 Hypothesis Five 48
3.4.6 Hypothesis Six 51
3.4.7 Hypothesis Seven 51
3.5 Metrics to Test Hypotheses 52
3.5.1 Measurement of Reliability 52
3.5.2 Measurement of Specification Changes 53
3.5.3 Measurement of Complexity 53

3.5.3.1 Measurement of Size Complexity 55
3.5.3.2.1 Surrogate Metrics for Hypothesis Two 59
3.5.3.2 Surrogate Metrics for Hypothesis Three. 61
3.5.3.2.1 Structure Complexity. 64
3.5.3.2.2 Surrogate Metrics for Hypothesis Four 69
3.5.3.2.2 Surrogate Metrics for Hypothesis Five 72
3.5.3.3 Temporal Complexity Metrics 73
3.5.3.3.1 Surrogate Metrics for Hypothesis Six 75
3.5.4 Measurement of Times Executed 76
3.6 Mathematical Model 77
3.7 Statistical Tests 78

Results 80
4.1 Data Gathered 80
4.2 Testing of Hypotheses 85
4.2.1 Correlation Test for Hypothesis One 86
4.2.2 Correlation Test for Hypothesis Two 86
4.2.3 Correlation Test for Hypothesis Three 87
4.2.4 Correlation Test for Hypothesis Four 88
4.2.5 Correlation Test for Hypothesis Five 89
4.2.6 Correlation Test for Hypothesis Six 93
4.2.7 Correlation Test for Hypothesis Seven 92
4.3 Predictive Model 92
4.3.1 Transposed Stepwise Model 93

Summary, Limitations, and Conclusions 97
5.1 Summary of Results 97
5.2 Limitations 99
5.3 Implications for Future Research 100
5.4 Application of Results 101

Appendix A: The METRICS Program. 104
A.1 Data Structures 104
A.2 Functions 107
A.3 Procedures 110
A.4 Reliability of the METRICS Program 116

Appendix B: The Metrics Program Source Code 122

Definition of Model Variables. 184

Bibliography 186

VITA 190

List of Figures

Figure 1: Failure Intensity Models 11
Figure 2: Human Information Processing Model 31
Figure 3: Basic Control Structures 35
Figure 4: Fault Maintenance Model 45
Figure 5: Module Call Complexity 71
Figure 6: Plot of Residuals for Model Sample of Stepwise Model 95
Figure 7: Plot of Residuals for Holdout Sample of Stepwise Model . . . 95
Figure 8: Hierarchy Chart for Metrics Program 104
Figure 9: List of Functions 107

List of Tables

Table 1: Programs Accomplishing the Same Task with Different Characteristics that may Impact Reliability 4
Table 2: Principle Component Regression Model 25
Table 3: A Comparison of Errors with Size and Cyclomatic Complexity 26
Table 4: Parent Examples 29
Table 5: Cyclomatic Complexity Example 40
Table 6: A KNOT Example 42
Table 7: Identifier Span Example 43
Table 8: Reliability for Two Programs 50
Table 9: COBOL Size Complexity 57
Table 10: IF Block 61
Table 11: Literal Examples 64
Table 12: Mathematical Model 78
Table 13: Reliability Frequency Distribution for Total Sample 81
Table 14: Descriptive Statistics of Failure Sample 83
Table 15: Descriptive Statistics of Non Failure Sample 84
Table 16: Spearman's Correlation Table of Metric with Reliability . . . 85
Table 17: Coefficients for Combined Size Metric 90
Table 18: Coefficients for Combined Structure 90
Table 19: Correlation for Interaction 91
Table 20: Transposed Stepwise Model 94

Chapter 1

Introduction

This section explores software quality as a research topic. Beginning with the historical development, it covers situations where software reliability measurement is needed, and ends with the research purpose. A model to predict software quality requires theoretical support. Munson and Khoshgoftaar suggest that the software characteristics that affect quality can be derived from human information processing (HIP) models.[1] The goal of this research is to develop and test a model which predicts software reliability (a type of quality) from computer program characteristics based on HIP.

1.1 Historical Development of Software Quality

When organizations started using electronic computing during the 1950s, the focus was on computing hardware. Software was simple by today's standards. Hardware was very expensive. Software quality was a minor issue because the typical program was a one-time computerized

[1] Munson, J.C. and T.M. Khoshgoftaar, "Regression Modeling of Software Quality: Empirical Investigation," Information and Software Technology, Vol 32, No 2, (March 1990), p. 106-107.

experiment.[2] The first computer programming language, machine language, used binary codes to represent each different CPU operation. Machine languages are machine specific and difficult to use, thus assembly languages were developed. Assembly languages use mnemonics to represent CPU operations and represent the first move toward user-friendly programming. Assemblers allow single statements [a procedure CALL statement] to represent multiple CPU instructions. Although assembly languages reduced program size and increased program understandability, a portability problem remained.[3] Each different CPU design had its own machine and assembly language. When someone wanted to execute a program on a different computer with an incompatible CPU design, the program had to be rewritten.

The first high-level language was FORTRAN (Formula Translator), a scientific-oriented language resembling mathematical notation. High-level languages are much easier for programmers to use. Compiler programs translate high-level languages into the binary languages of a particular CPU. Programs were no longer dependent upon specific CPU designs.[4] Improved

---

[2] Elson, Mark Concepts of Programming Languages, Chicago:Science Research Associates, Inc., (1973), p. xiii.
[3] Sammet, Jean, Programming Languages, Englewood Cliffs NJ: Prentice-Hall, Inc., (1969), pp. 85-86.
[4] Tucker, Allen, Programming Languages, New York: McGraw-Hill Book Company, (1977), p. 65.

portability significantly increased the use and life of many pieces of software. Thus, quality became a valid concern for programmers.[5]

1.2 The Need for Software Reliability Measurement

MIS managers, internal auditors, external auditors, and programmers are all concerned with system quality. System quality is composed of hardware and software quality. Because of hardware's very high quality, the variability in system quality comes almost entirely from software. A method to predict software quality essentially predicts system quality because hardware quality is constantly high.[6] Software quality can be measured many ways, depending on the goals of the measurer. Reliability is a commonly used, user-oriented, measure of "software quality" that can be used as a probability in decision models.[7]

Programmers need information to create reliable programs. A programmer makes many decisions that affect a program's structure and content. If a programmer knew the characteristics of an unreliable program beforehand, program reliability could be increased.

---

[5] Buxton, J.N., Peter Naur, and Brian Randell, Software Engineering, New York: Petrocelli/Charter, (1976), pp. 33.
[6] Musa, Software Reliability: Measurment, Prediction, and Application, McGraw-Hill Book Company, (1987), p. 84.

For example, Table 1 contains two different programs that accomplish the same task. Due to different design characteristics, one of the programs has a higher probability of being reliable. Program one computes CGS earlier than program two. The programmer may forget what the identifier CGS means when coding the NIBT assignment statement for program one and is more likely to put an erroneous statement in the code.

Table 1: Programs Accomplishing the Same Task with Different Characteristics that may Impact Reliability

Program One	Program Two
Read(Sales)	Read(Sales)
CGS=.5*Sales	OE=500
OE=500	CGS = .5*Sales
Gross = Sales - CGS	Gross = Sales - CGS
NIBT = Gross - CGS - OE	STax = Sales * .045
STax = Sales * .045	NIBT = Gross - CGS - OE
Print (NIBT)	Print(NIBT)

Even if the original programmer doesn't make a mistake, an error is more likely to be introduced into program one's code during maintenance because the value for NIBT is computed long before it is printed. Its value could be inadvertently changed by the insertion of new statements in the middle of the

[7] Ibid., p. 35.

program. Software reliability measures for the different programs could help quantify the reliability gain from using program two.

MIS managers are interested in knowing and increasing the reliability of systems.[8] However, to purposefully increase the reliability of a system, it is necessary to observe the system at a greater level of detail where changes can affect overall reliability. For example, suppose a system is 34% reliable and composed of four sequentially-executable application programs each solving a part of the problem in sequence. Assume that the reliability is 40% for the first program and 95% for the others. Because reliability cannot exceed 100%, a larger gain in system reliability can occur from modifying the first program than from any of the other programs. Thus, it is necessary to know or predict the reliability of each program to optimize system reliability.

Both internal and external auditors have a need to measure system reliability. Internal auditors help managers determine which systems are unreliable. A survey by the Institute of Internal Auditors (IIA) provides evidence that upper management is concerned with MIS reliability. The IIA surveyed the internal auditors of a large commercial bank to develop an

---

[8] Miller, James, Living Systems, (New York: McGraw-Hill, 1978), p. 128.

educational case study.[9] The survey suggests that managers believe internal audits can discover unreliable computerized information systems and need a model for predicting the reliability of computer programs.

External auditors need to measure system reliability to determine the type and degree of compliance and substantive audit testing to conduct. From SAS 48, "The extent to which computer processing is used in significant accounting applications, as well as, the complexity of that processing, may also influence the nature, timing, and extent of audit procedures."[10] SAS 47 provides an audit risk model for determining an appropriate level of testing. Audit risk depends on inherent risk, control risk, and detection risk. These elements of the audit risk model from SAS 47 are used to choose the amount of audit evidence needed in a particular situation.

While auditors have no control over inherent or control risks, they estimate both and set detection risk at a value which provides an acceptable audit risk. The level of detection risk determines the amount of audit testing.[11] Control risk is predicted from the internal control system. Inherent

---

[9] Lagman, Bernard, "Audit and Control of Systems Programming Activities," The Institute of Internal Auditors, (1985), p. 72.

[10] AICPA, Statement on Audit Standards No. 48, "The Effects of Computer Processing on the Examination of Financial Statements," (July 1984), p. 43.

[11] Choo and Ferrar, "An EDP Audit Model," Journal of Systems and Software, (August 1986), p. 43.

risk is associated with the MIS system. Inherent risk is high for an unreliable system. A reliability measure is needed to predict inherent risk before audit testing.

1.3 Development of Research Purpose

From the late 60's through the 80's, 75% of all business applications were written in COBOL (Common Business-Oriented Language).[12] Organizations spent half of their MIS budgets to produce approximately five billion lines of COBOL code.[13] COBOL programs performed 60% of all data processing and took more than 50% of maintenance effort.[14] During the early 90's the United States federal government used COBOL for 70% of its computer applications.[15] COBOL is currently the most widely used programming language in the world. There are more than 30 billion lines of COBOL code in operation, and in 1994, 5 billion new lines of code were created.[16] COBOL is expected to continue and evolve into an object-oriented language toward the year 2000 when a newer version of ANSI COBOL will be released.

---

[12] Kolodziej, J., "COBOL Shapes Up," Computerworld, Vol 21, Iss 1a, (Jan 7, 1987), p. 13-14.
[13] Parikh, Girish, "Making the Immortal Language Work," Business Software Review, Vol 6, Iss 4, (April 1987), p. 33-36.
[14] Ibid.
[15] The Office of Technical Assistance Federal Software Support Center Report, (February 1992).

Since many business-oriented programs are written in COBOL, reliability is an important issue for those who maintain, create, and use COBOL programs. For these reasons, COBOL is the programming language for this research.

COBOL has been the most commonly used language for creating business applications. Groups both internal and external to an organization need program reliability estimates, thus a COBOL reliability model is needed. The purpose of this research project is to determine if certain factors are important to the reliability of COBOL programs and then use these factors to predict the reliability of COBOL programs.

---

[16] Lawrence, Andrew, "IBM System User International Survey," (March 1996), Computer Business Review, p. 1-4.

Chapter 2
Literature Review

The literature review is divided into two primary sections. The first section contains reviews of failure models and the second section contains reviews of reliability prediction from design characteristics.

2.1 Failure Models

The theoretical fault maintenance model discussed in chapter 3.3 implies that the number of faults in a program should decrease over time. Specifically, annually executed programs are not tested as well as those that are executed daily. Although the programs have been in operation for the same time, daily executed programs should have fewer faults because of more opportunities to find and fix faults. The expected number of faults in the average program should decrease every time it is executed. Several mathematical models have been developed to predict this decrease.

The basic model predicts failure intensity (λ) from the failures that have occurred (υ_o).[17] Failure intensity is the inverse of reliability. As failure intensity increases, reliability decreases. The basic model assumes that

failure intensity is constantly decreasing by a stable rate. The current failure intensity depends upon the beginning failure intensity, the failures encountered, and the total failures for the program.

$$\lambda(\mu)=\lambda_o(1-\mu/\upsilon_o)$$

While the failure intensity predicted by the basic model decreases at a constant rate, the logarithmic Poisson model (natural logarithm) has a failure intensity that decreases proportionally to the current number of faults.[18] The failure intensity decreases rapidly at first but then slows as more faults are encountered. This model has a decay parameter (θ) to control the rate by which failure intensity decreases.

$$\lambda= \ln (\theta\nu)$$

Using either of the two models to predict failure intensity is difficult. The basic model, because it depends on knowing the total number of failures since the program was written, and the logarithmic Poisson's model because it assumes that the decay parameter is know in advance. Also, both models depend upon the total failures that have occurred over a period which is rarely known and computationally complex because the execution history of each

---

[17] Goel, A.L., and K. Okumoto, "A Markovian Model for Reliability and Other Performance Measures," Proceedings National Computer Conference, (1987), pp. 769-774.

program is in many different files. Figure 1 is a graph of the two failure intensity models.

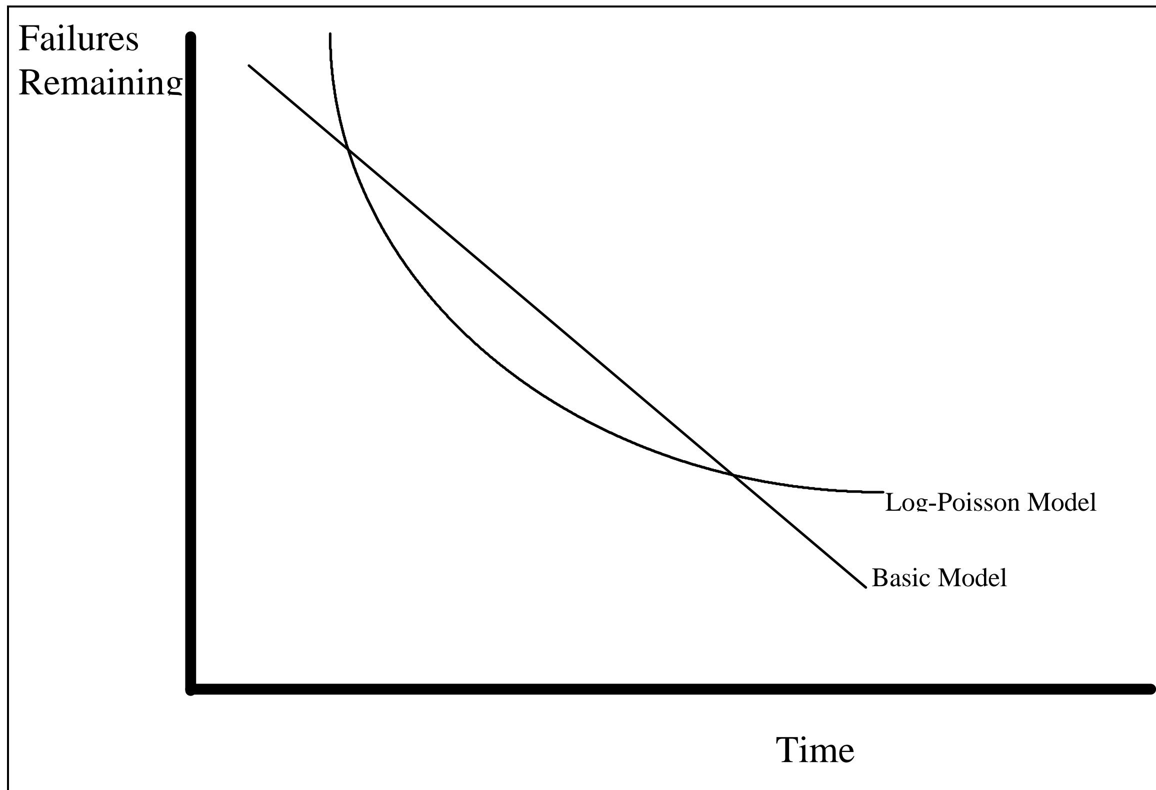


Figure 1: Failure Intensity Models

Alternative forms of the basic and logarithmic Poisson models have been derived. These models estimate failure intensity from the total amount of time a program has executed (τ) .

[18] Musa, John D., Software Reliability: Measurement, Prediction, and Application, McGraw-Hill, (1995),

For the basic model:[19]

$$\mu(\tau)=\upsilon_o(1-EXP(-\lambda_o/\upsilon_o\ \tau))$$

For the log Poisson model:[20]

$$\lambda(\tau)=\lambda_o/(\lambda_o\theta\tau+1)$$

The primary advantage of the execution time models over previous models is that a detailed failure history isn't needed; however, the total execution time since the program was created is essential. It is very unlikely that the exact execution time is known for all programs especially for an on-line real-time transaction processing system. However, it could be estimated for batch programs if the amount of data processed per execution is assumed to be constant throughout the program's life. The basic model predicts a higher failure intensity at the beginning, while the log Poisson model is higher toward the end. However, the curves are similarly shaped.

Although getting failure and execution time data for any program may be difficult, a study by Musa and Okumoto compared the predictive ability of

---

p. 34-36.
[19] Ibid., p. 37.

both the basic and logarithmic Poisson models.[21] The data set contained over three thousand failures from programs written by over 300 programmers. They found that although the basic model had predictive ability, the logarithmic Poisson model was better when the program had a nonuniform operational profile.

A program with a uniform operational profile processes many transactions with few different transaction types. An example could be a demand deposit accounting system for a bank. Although many transactions are processed, two types predominate, deposit and withdrawal. A program with a nonuniform operational profile processes many transactions of which there are many different types. An example could be a registration system at a college. It registers students, prints transcripts, checks for scheduling conflicts, assigns rooms, prints report cards, prints a list of students on academic probation, etc. The Musa and Okumoto (1984) results may occur because the cyclomatic complexity of a non-uniform profile program is much larger. Errors are located and removed from the uniform profile program more quickly because fewer control paths need to be checked (corresponding

---

[20] Ibid., p.38.

to the logarithmic Poisson model). When the operation profile is nonuniform it takes more executions to check all of the control paths and longer to find and remove errors (basic model).

Although the number of times a program has been executed is an important variable affecting software quality, there are several others. In particular the characteristics of the program's source code also contains information that can be used to study software quality. Measures of software characteristics are called complexity metrics. Complexity metrics are related to the complexity of the problem that the programmer faced when writing the program. Thus, software quality is related to software metrics.

2.2 Reliability Prediction from Design Characteristics

This section is divided into two subsections. The first subsection covers the measurement of complexity using software metrics. The second subsection covers the prediction of software quality from design characteristics.

2.2.1 Software Metrics

---

[21] Musa, and Okumoto, "A Comparison of Time Domains for Software Reliability Models," Journal of Systems and Software, 4 (4), (1984), p. 277-287.

Software metrics were created to measure the amount of structure in a program. This section covers studies that relate software characteristics to complexity.

Harold designed a methodology to find if software metrics have significantly different values for structured and unstructured programs.[22] The subjects were students enrolled in a COBOL course. Subjects (36) were divided into two equal groups. The control group learned programming without structured programming techniques while the experimental group learned to program with structured programming techniques. Each student wrote four programs for professionals to evaluate. All professionals had at least three years of programming experience.

Each professional evaluated three different programs by these characteristics:

--	Specifications	--	*Flow of Logic
--	Program Modularity	--	Empirical Modifiability
--	Logical Linkages	--	Logical Simplicity
--	Comments	--	Indentation
--	*Module Size	--	*Program Size
--	Data and Procedure Naming	--	Debugging Difficulty

[22] Harold, Frederick Gordon, "An Experimental Analysis of COBOL Program Quality Through the Application of Software Metrics to Programs Written with and without Structured Programming Techniques," Unpublished Doctoral Dissertation, George Washington University, (1982).

The objective measures (marked with an asterisk) were computed in advance to speed up the evaluation. The professionals evaluated all of the programs by the non-objective characteristics in one afternoon.

Discriminant analysis was used to predict whether or not each program was written with structured techniques from the program's characteristics. The most significant characteristics were program size ($p < .01$) and flow of logic ($p < .01$). Both significant measures are objective and can be computed without a professional. Harold's research shows that objective metrics can measure the degree of structure and quality of a program.

Rodriguez and Tsai used software metrics to predict if programs will be difficult to maintain.[23] Their sample contained 371 programs. Maintenance programmers were asked to pick out the most difficult programs to maintain: this was the complex group. The other programs were put into the non-complex group. Discriminant analysis was used with size, data structure, and control metrics to develop a model to predict a program's complexity classification. The discriminant analysis model correctly predicted the complexity of 96% of the programs. The metrics with the most

---

[23] Rodriguex, Volney, and Tsai, W.T., "A Tool for Discriminant Analysis and Classification of Software Metrics," (1988), Systems, Software Development Section.

discriminating power were: volume of operators and operands, information flow, and number of program statements.

The sample programs were reclassified. Any program with more than 75 lines was put in the complex group. The non-complex group had 313 programs while the complex group had 58. The discriminating metrics were then: volume of operators and operands, cyclomatic complexity, and number of program statements. This new discriminant model correctly classified 88% of the programs and is better than a naive model (84%). The analysis showed that larger programs tend to have larger complexity values.

Canning compared different metrics for large systems.[24] Three projects from the NASA/Goddard Space Flight Center were analyzed. Each project had several subsystems within it, and all of the programs were written in FORTRAN (Formula Translator). Sixteen software metrics were computed for each program.[25] The following additional data were collected for each

---

[24] Canning, James, "The Application of Structure and Code Metrics to Large Scale Systems," Unpublished Doctoral Dissertation, Virginia Tech, (1985).

[25] The complexity metrics that were computed for each program were: 1) information flow, 2) stability, 3) syntactic interconnection, 4) invocation complexity metric, 5) Halstead's effort, 6) lines of code, 7) McCabe's cyclomatic complexity, 8) information flow weighted by Halstead's effort, 9) information flow weighted by McCabe's volume, 10) information flow weighted by lines of code, 11) stability measure weighted by Halstead's Effort, 12) stability measure weighted by McCabe's volume, 13) stability measure weighted by lines of code, 14) syntactic interconnection model weighted by Halstead's Effort, 15) syntactic interconnection model weighted by McCabe's volume, and 16) syntactic interconnection model weighted by lines of code.

subsystem: design hours, code total hours, code hours, test hours, and total time (development measures). Code total hours include the time for reading and reviewing the code while code hours is only the time to code the subsystem. A correlation analysis between the two sets of data showed that the software metrics are highly positively correlated with errors and changes in the data (lines of code was the highest). The development measures and software metrics were also highly correlated (cyclomatic complexity was the highest).

Henry and Kafura (1986) studied the relationships among software metrics used within a UNIX environment.[26] Program length was highly (greater than .90) correlated with Halstead's Volume, Halstead's Effort, and McCabes cyclomatic complexity. It was not highly correlated with information flow. Information flow is the flow of data items such as variable names between procedures as a program is executed. Information flow is highly correlated with the occurrence of errors.

McCabe and Butler (1989) have factored McCabe's cyclomatic complexity into three components—module design complexity, design

[26] Henry, Sallie, and Kafura, Dennis, "The Evaluation of Software Systems' Structure using Quantitative Software Metrics," Software—Practice and Experience, Vol 14, (June 86), p 361-573.

complexity, and integration complexity.[27] Module design complexity is the complexity involved between modules. Design complexity is the complexity of the statements within a module. Integration complexity is the complexity from having a module called by more than one calling module.

Henderson-Sellers and Tegarden (1994) have proposed an alternative form of the traditional McCabe's cyclomatic complexity.[28] The Henderson-Sellers cyclomatic complexity is not inflated by multiple calls to the same subroutine from within the calling routine. This type of cyclomatic complexity more reasonably represents psychological complexity than McCabe's complexity because it doesn't repeat the complexity of subroutines called several times from within a single subroutine.

2.2.2 Complexity Studies

This subsection covers studies that relate program complexity to program quality. The studies used different measures of quality such as reliability or faults. Fault measurement is important to those who maintain

[27] McCabe, Thomas, and Butler, Charles, "Design Complexity Measurement and Testing," Communications of the ACM, Vol 32, No. 12, (December 1989), p. 1415-1425.
[28] Henderson-Sellers and Tegarden, "The Theoretical Extension of Two Verssions of Cyclomatic Complexity to Multiple Entry/Exit Modules," Software Quality Journal, Vol 3, (1994), p. 253-269.

programs because it indicates the amount of maintenance time a program will require. Reliability is a more user-oriented measure of software quality. It is used by those who utilize information generated by computer software for decision making.

Boehm conducted one of the early studies to predict reliability from complexity.[29] He wrote two programs to accomplish the same task. The first program used structured techniques and was less complex than the second. Based on an "a priori" model, Boehm believed the reliability for the less complex program would be 95.8% and the more complex program 89.3%. The programs were executed with data from a test deck developed with a random number generator. The percentage of correctly executed cases for each of the two programs were 99.7% for the less complex program and 96.3% for the more complex program. His results suggest that program complexity and reliability are inversely related. However, the above study has several limitations: 1) the researcher wrote the programs, 2) only two programs were tested, and 3) the parameters in the predictive model were not

[29] Boehm, B., "Reliability and Complexity," Proceedings of the TRW Symposium on Reliable, Cost-effective, Secure Software, March 20-21, 1974, Los Angeles, P. 121-130.

developed from actual data (this may be why the model underestimated reliability in both cases).

Thomas Thayer studied the relationship between complexity and the faults in programs from large systems.[30] Four types of complexity were measured: logic, computational, input/output, and readability. The logic complexity metric was a measure of program structure. For each subsystem, complexity was defined as the summation of the logic statements in the subsystem divided by the sum of executable statements. Computational complexity was the number of assignment statements. Input/output complexity was the number of read and write statements. Readability was the number of comment statements. These metrics were combined to form a total complexity model. Each complexity metric in the model was weighted by the researchers preconceived idea of the importance of the metric in computing total complexity. Thayer found that the faults in the software were related to total complexity (correlation .907).

---

[30] Thayer, T, and E. Nelson, Software Reliability: A Study of Large Project Reality, North-Holland Publishing Company, Amsterdam, New York (1978).

Takashashi designed a study to learn if design factors such as size can predict the total number of faults in a program.[31] The data consisted of 30 software projects. The data were normalized by program size (executable statements) to make the other factors comparable. The regression model used to predict the number of faults had the following significant independent variables:

1) specification change activity (times that the program was modified to provide new information).

2) average programmer skill in years, and

3) thoroughness of design documentation, measured in lines of developed documentation per one thousand lines of code.

Specification change activity (user requirement change) was positively associated with more faults while greater programmer skill and thoroughness design documentation was associated with fewer faults.

---

[31] Takashashi, M. "The Linear Software Reliablity Model and Uniform Testing," IEEE Transactions Reliability, R-34 (1), (1988), p. 8-16.

Motley and Brooks found that total faults could be predicted from program size and number of I/O statements.[32] Lipow and Thayer found four significant variables for predicting the number of faults in a program.[33] The four variables were:

1) Number of branches,
2) Number of calls to subroutines,
3) Number of levels of nesting of IF statements
4) Number of assignment statements not involving arithmetic operations.

Munson and Khoshgoftaar studied the ability of software metrics to predict the effort needed to "put programs into service."[34] In particular the dependent variable "putting a program into service" is the number of changes made to a program between the time it was originally written and the time it was put into operation. The data were from two subsystems of 67 programs. The independent variables were software metrics measured from each program.[35]

---

[32] Motley, R.W., and W.D. Brooks, Statistical Prediction of Programming Errors, RADC-TR-77-175, Rome Air Development Center, Griffis Air Force Base, New York, (1977).

[33] Lipow, M. and T.A. Thayer, "Prediction of Software Failures," Proceedings 1977 Annual Reliability and Maintainability Symposium, IEEE CAT NO. 77CH11619RQC, p. 489-497 (1977).

[34] Munson, J.C., and Khoshgoftaar, T.M., "Regression Modelling of Software Quality: Empirical Investigation," Information and Software Technology, Vol 32, No. 2, (March 1990), pp. 106-114.

[35] The software metrics were: machine code instructions, lines of code, modules called, data items, parameters, Halstead's operator, Halastead's operand, total operators, total operands, and McCabe's cyclomatic complexity.

Before conducting the regression analysis, a correlation matrix showed that many software metrics were correlated. Principle components analysis was used to reduce the number of independent variables and eliminate multicollinearity. The factors were given the names: volume, control, and parametric (number of variables) because the contribution of each metric seemed to fit this description. The independent software metrics were reduced to three principle component variables that explained 85% of the variance.

The three principle component variables were used as the independent variables and program changes as the dependent variables in the regression model. Stepwise backward elimination regression was used to find the regression model with the least average PRESS statistic.[36] The best model contained all three factors as independent variables and a significant intercept. The model is in Table 2. The regression model explained 66% of the variation in the number of changes.

---

[36] The press statistic is a measure of prediction error. In particular it is defined as $\sum(\bar{y} - yi)^2$

Table 2: Principle Component Regression Model

Parameter	Estimate	Error
Factor 1	1.50	0.62
Factor 2	3.13	0.62
Factor 3	2.27	0.62
Intercept	5.22	0.65

Basili and Perricone studied the occurrence of software errors and complexity among different size programs.[37] The programs were from a large set of projects in the Software Engineering Laboratory. These programs do planning studies, maneuvering, mission launch, and mission control functions for satellites. The study concentrated on errors in the modules (subroutines) of FORTRAN programs. Each module was manually examined for errors. An error was identified whenever something within a program caused the module to perform incorrectly. A total of 215 errors were detected in 96 of the 370 modules (26% of modules had errors). The data for the number of errors were gathered from program change requests over a period of 33 months (August 1977-May 1980).

[37] Basili, Victor, and Barry Perricone, "Software Errors and Complexity: An Emprical Investigation," Communications of the ACM, (January 1984), Vol 27, no. 1, pp. 42-52.

Table 3: A Comparison of Errors with Size and Cyclomatic Complexity

Module Size	Cyclomatic Complexity	Errors/1000 lines
50	6.2	65.0
100	19.6	33.3
150	27.5	24.6
200	56.7	13.4
>200	77.5	9.7

The results seem ambiguous. Although the total errors per module increased, the average errors per thousand lines of code decreased with module size. Cyclomatic complexity increased with size. Basili and Perricone concluded that there is no need to put an arbitrary limit on size.

Binder and Poore (1990) used a sample of 144,500 lines of COBOL code to try to predict the documentation quality from software metrics.[38] The significant metrics in decreasing order of significance were author of last modifications, total number of exceptions to EMIS standards, date written, negative conditions, alphanumeric literals, read verbs, number of comment lines.

The literature review shows that software quality is related to the number of executions and the characteristics of a program. Both will be used to develop a comprehensive model for software quality. This study will use

reliability as a measure of software quality. Reliability is user-oriented and useful for decision making. Auditors could use the reliability of a system to measure the quality of particular systems when conducting an audit.

---

[38] L.H. Binder and J.H. Poore, "Field Experiements With Local Software Quality Metrics," Software—Practice and Experience,, Vol 20(7), (July 1990), pp. 631-647.

Chapter 3
Model Development

This chapter begins with the unique characteristics of COBOL and covers several theoretical models relating to reliability. Then, a set of hypotheses about COBOL program reliability are presented along with a mathematical model that can be used to develop a statistical predictive model.

3.1 Characteristics of COBOL

COBOL has several characteristics that make complexity measurement different from other high-level languages. COBOL programs are long. A program written in COBOL is longer than the same program written in any other high-level language. COBOL is an English-like language. If a programmer writes two programs that are the same length in COBOL and another high level language, the COBOL program is easier to understand. Auditors and others with a business background can read COBOL programs. Because COBOL programs are easier to read, size complexity shouldn't decrease reliability as much as with other languages.

Although most high-level languages use globally and locally-defined variables to isolate procedures from the rest of the program (loose coupling),

COBOL allows only global variables. The variable definitions for COBOL are also different because COBOL variables can have parents. A parent is a last or common name given to a group of variables. Parents are commonly used for input/output record names. Parents can also organize variables. Table 4 contains an example of various types of taxes organized under the parent TAXES.

Table 4: Parent Example

```
01      TAXES.
        05 STATE        PIC 999V99.
        05 FEDERAL      PIC 999V99.
        05 FICA         PIC 999V99.
```

Generally, COBOL is used to create business programs. COBOL's characteristics make it a good language for generating reports. Many other languages such as FORTRAN and C (developed originally for the UNIX operating system) are better for scientific programming. It is likely that the determinants of reliability for a COBOL program are different because different programming techniques are used when creating business programs. For example, control and multiple control breaks (a programming technique used to generate subtotals in reports) are common in COBOL programs but

rare in scientific programs. It is also likely that complexity affects the reliability of COBOL programs differently because of these COBOL characteristics: English like, long, only global variables, parent names for variables, and primarily a business language.

3.2 Development of Theoretical Model

The studies in the literature review predicted the faults in a program from complexity metrics without providing theoretical support for why each specific metric was chosen. When research lacks theoretical support it is uncertain whether causality exists between the independent and dependent variables. To develop a model to predict program reliability, it is necessary to examine why certain program characteristics depend on program reliability. The human information processing (HIP) model is used to provide that guidance. This model is depicted in figure 2.

Creating computer programs is a human activity, and is constrained by the human information processing model. The processes that programmers are trying to represent are all defined by a list of instructions for a computer to follow. When the instructions are too complicated, programmers make mistakes, thus the program becomes unreliable.

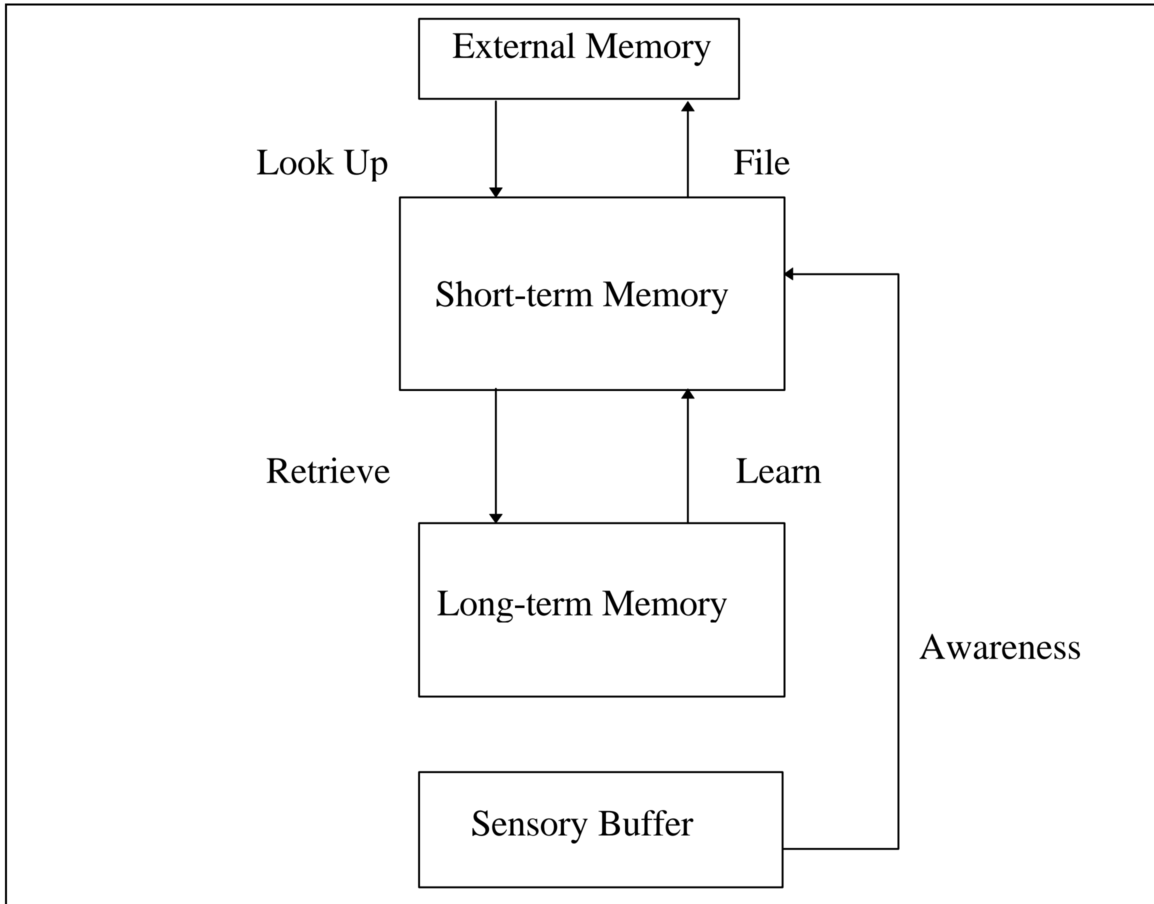


Figure 2: The Human Information Processing Model[39]

Humans receive information from the environment through five different sensory organs. Information received through sensory organs is put into very-short-term memory (VSTM). Very-short-term memory is a buffer that ranks the sensory stimuli to be processed. When the stimulus suggests a

life threatening situation, it is processed immediately. It takes from 1/4 to two seconds to process each stimulus. Since computer programming rarely involves life threatening situations, VSTM is only used to relay information to short-term memory (STM). As information enters STM, it gets the person's attention. Short-term memory holds whatever a person is currently thinking about. Information is organized by chunks and stored within STM but is limited to about seven chunks at any one instant.[40]

Because STM has a limited capacity, information is sometimes lost when attention is focused from one thing to something else. To keep something in STM it must be rehearsed or thought of occasionally. Information can stay in STM for only about 18 to 30 seconds.[41] Long-term memory (LTM) has different characteristics than STM. Long-term memory has an almost unlimited amount of storage capacity. Five to ten seconds are required to store something in LTM. This process is called learning.[42] Information can be retrieved (recall) from LTM into STM whenever needed. However, LTM is diminished with time.

---

[39] MacGregor, James, "Short-Term Memory Capacity: Limitation or Optimization," Psychological Review, 1987, Vol 94, No. 1, pp. 107-108.
[40] Ibid.
[41] Atkinson, R.C., and R.M. Shiffrin, "The Control of Short-Term Memory," Scientific American, 225, (1971), pp. 82-90.

External memory is information stored outside the human body. For example, if a person cannot remember how to spell every word in their speaking vocabulary, then words can be looked up in a dictionary. External memory is less efficient than LTM because the human processor reads information into STM through a sensory buffer. Also, if retrieving from external memory takes more than about 30 seconds, the contents of STM are lost. Retrieving from external memory uses additional STM reducing the remaining capacity available to solve the problem. External memory is usually used to store information that is rarely needed.

This HIP model can be applied to computer programming. Computer programming is the process of writing a list of instructions for a computer to follow.[43] Programming languages have a limited vocabulary of about 100 verbs (commands to do something to data). Data in programs are given variable or identifier names. Verbs use identifiers to specify which data are affected by the command.

A programmer, writing a program, uses long-term and external memory to store:

---

[42] Weinber, G.M., The Psychology of Computer Programming, Von Nostrand Reinhold Company, New York, 1971.

--programming language syntax,
--data base or file structures,
--algorithm for solving problems, and
--program specifications.

Although these items could be stored in external memory such as a book, it is much more efficient to use the programmer's long-term memory. If items are stored in external memory, some STM is required to retrieve the information. This reduces the amount of STM left to solve the problem.

The HIP model implies that the most critical aspect of programming is simultaneously getting everything necessary to solve the problem into the programmers STM. Since the size of STM cannot be increased, large programs must be organized to take up less STM. Structured programming is a methodology programmers use to write large programs with their limited STM.

3.2.1 Structured Programming

Since the early 1970's, structured programming has been used to increase program reliability.[44] Structured programs are organized like a book, the instructions are executed in the order they appear. The structured

---

[43] Dale, Nell, and Chip Weems, Turbo Pascal, D.C. Heath and Company, Lexington, Massachcusetts, 1988, p. 17.

programming methodology involves dividing a problem into subproblems small enough to be solved with STM. Next, each subprogram is coded using a combination of the three basic control structures shown in figure 3.

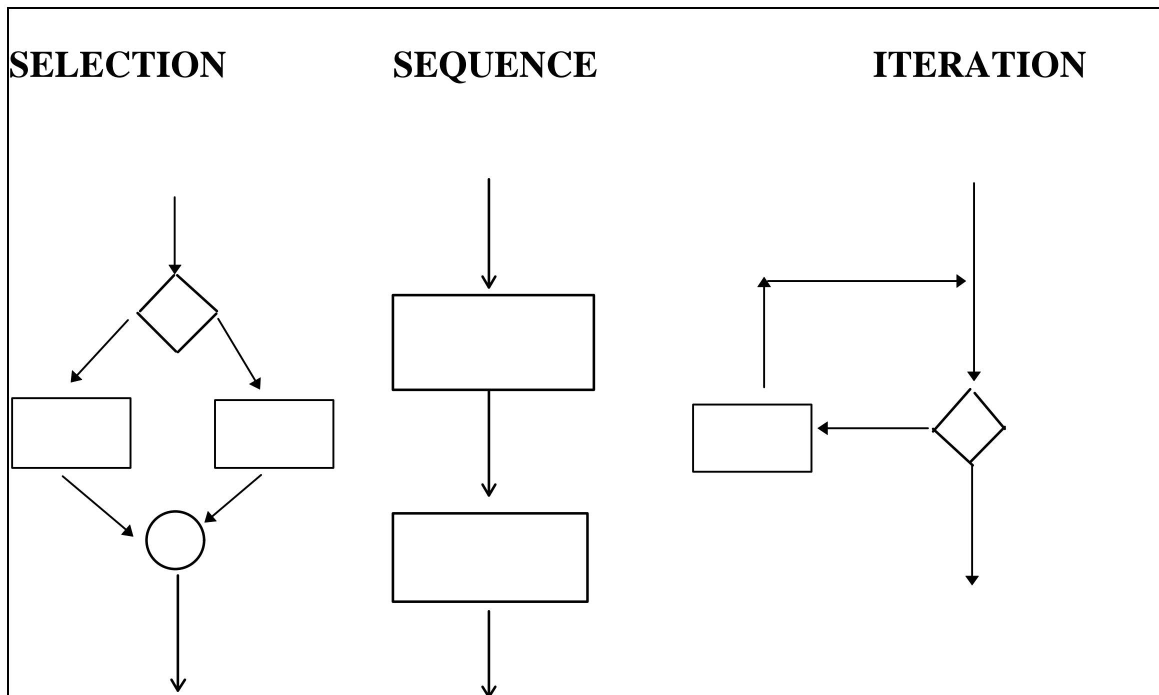


Figure 3: Basic Control Structures

These basic control structures are used to reduce the possible relationships between the statements in a program. By eliminating GOTO statements, each module (subprogram) can have only one entry and one exit point. Each statement is no longer potentially executable directly after every

[44] Donaldson, James R., "Structured Programming," Datamation, Vol. 19, Issue 12, (December 1973), p.

other statement. Any problem can be solved by using the structured programming methodology.[45]

Although in theory structured programming increases the quality of programs, it is potentially very difficult to prove. Every program is different and possess differing degrees of structure. The use of complexity metrics is a way of measuring the varying degrees of structured programming that can exist.

3.2.2 Complexity Metrics

Psychological complexity can be defined as the difficulty encountered by a person trying to understand a program.[46] Psychological complexity is caused by short-term memory limitations. Three basic events can cause a short-term memory failure. First, the problem can be so large that it requires more chunks of short-term memory than are available. For example, suppose computing a tax liability requires the analysis of 10 different characteristics. Then, it is possible that the person's short-term memory would be exceeded by the 7th item making it unlikely that the problem would be solved correctly.

54.

[45] Fainter, Robert, "Program Complexity Measures," Unpublished Masters Thesis, Virginia Tech, (March 1981), p. 1.

[46] Ibid.

Second, inefficiently organized problems require more short-term memory. For example, suppose that someone is evaluating investments, and each investment has about 100 different characteristics. Since only about seven characteristics can be considered at once, it is unlikely that the best investment can be chosen from the group. However, if the investments are broken into a hierarchy by characteristics, the decision becomes more manageable.

Finally, a problem could be temporally spaced making it difficult to formulate into short-term memory at once. Suppose that a problem requires someone to compare seven items and decide whether to hold or sell a stock. Each item is constantly changing and has to be looked up in a large database. Suppose it takes about 30 seconds to look up each item. Since short-term memory is already near its limits when comparing the seven items, as attention is focused to look up additional items, extra STM is required. A short-term memory failure will cause the person to forget some information already in STM. Since the items are constantly changing and it takes so long to look each up, it is practically impossible to get all of them into STM at

once. When the hold/sell decision is made, the first items looked up have been forgotten and are not considered.

As a person writes a program, the contents of short-term memory are constantly changing. After a program is written, it is impossible to get specific information about what was in short-term memory. Even if it were possible to measure the contents of short-term memory, it would not be practical. Before the reliability of a program could be predicted with the model, the short-term memory of every programmer who has worked on the program would have to be measured along with the effects of the programmer's work on the program--greatly decreasing the model's usefulness. Thus, program complexity is commonly used as a surrogate for psychological complexity.

> Program complexity is the set of program characteristics that affect the psychological complexity of a program and may be measured by a set of scales computed directly from the program to numerically characterize the difficulty of comprehending the program.[47]

[47] Fainter, Robert, Ibid. p. 2.

Like psychological complexity, program complexity metrics can be separated into three groups: size, structural, and temporal. Size complexity metrics are measures of the length of a program. A large value for a size metric suggests a higher probability of the programmer running out of short-term memory. Size complexity metrics are common and easy to measure. Examples include number of statements, comment lines, verbs, and variables. Each programming language uses statements in a different way; thus, the same program written in two different languages will rarely be the same length. The primary limitation of traditional size complexity is that measurements cannot be compared for programs written with different languages.[48]

Structural metrics are associated with the interdependencies between the statements in a program. Structural complexity is commonly used to compare programs written in different languages because most languages use the same basic control structures. McCabe's cyclomatic number is the

[48] Halstead developed several metrics to measure size complexity. Halstead's measures are based upon operators and operands. An operand can be either a constant or a variable name. An operator is similar to a mathematical operator (+,-,*, /, or =) but could be any of a computer's operations. Halstead's volume metric is a measure of total size of the program while the effort metric is a measure of the mental complexity to write the program.

number of control flow paths through a program's source code.[49] The cyclomatic number is the total of IF, WHILE, and EVALUATE statements plus one. Table 5 contains an example of cyclomatic complexity for two programs.

Table 5: Cyclomatic Complexity Example

Program A	**Program B**

Read(Sales)
If Sales > 500
    Then Tax=Sales*.1
    Else Tax=Sales*.2.
Print (Tax)
	Read(Sales)
	Tax = .1*Sales
	Print(Tax)

Program A's cyclomatic number is two because there are two paths through the program (One If statement + 1). Program B has just one path through it, so the cyclomatic number is one (No If, While, or Evaluate statements + 1).

An unstructured program can always be reorganized so that the cyclomatic complexity is one or two making it structured.[50] When the

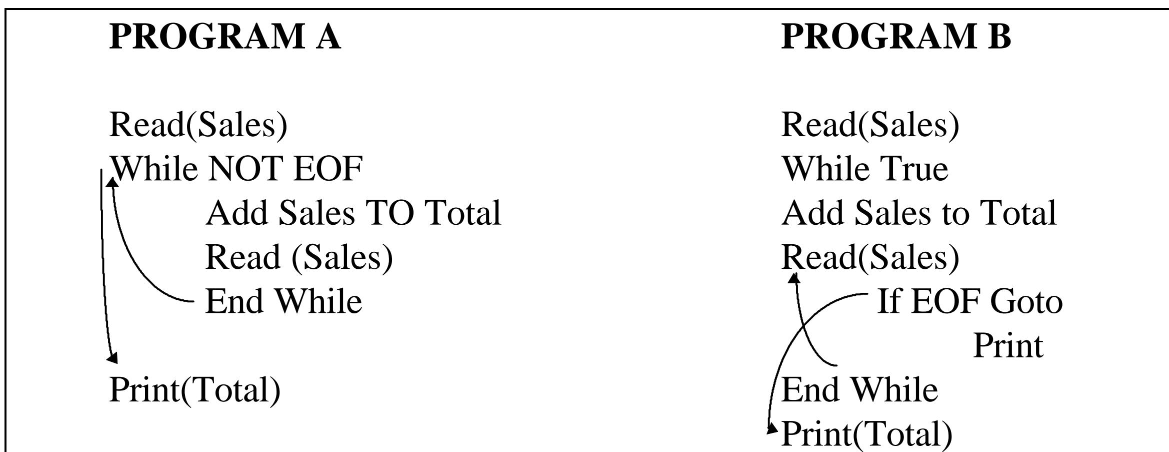
---

[49] J. McCabe, "A Complexity Measure," IEEE Transactions on Software Engineering, SE-2, No. 4, (December 1976), pp. 308-320.
[50] Linger, R.C., H.D. Mills, and D.I. Witt, Structured Programming: Theory and Practice, Addison-Wesley-Publishing Company, Reading, Mass., 1979, p. 78.

cyclomatic number is high, many different potential execution sequences exist for the statements in the program.

Like sentences in a book, a structured program's statements are in the order that they will be executed. Woodward's complexity measure "knots" measures the disarray of statements in a program.[51] A knot exists whenever the control flow paths within a program cross. A structured program has no knots. In COBOL, a knot can be caused by either GOTO or PERFORM statements. Table 6 contains an example of a program with a knot. Lines have been drawn to show the directions of control flow. Whenever the lines cross a knot exists. Program A has no knots. Program B has one knot.

Table 6: A KNOT Example



PROGRAM A	PROGRAM B
Read(Sales)	Read(Sales)
While NOT EOF	While True
Add Sales TO Total	Add Sales to Total
Read (Sales)	Read(Sales)
End While	If EOF Goto Print
Print(Total)	End While
	Print(Total)

[51] Woodward, M.R., M.A. Hennel, and D. Hedley, "A Measure of Control Flow Complexity in Program Text," IEEE Transactions on Software Engineering, SE-5, Issure 1, (January 1979), pp. 45-50.

Another group of complexity metrics is temporal metrics.[52] Temporal metrics measure the difficulty of getting the information required to solve a programming problem into STM at once. A temporal metric (identifier span) was developed by Elshoff.[53] Identifier span is the number of source code statements between two textual references to the same identifier [variable]. When the distance between the two references is great, the programmer is more likely to forget what the identifier means before using it again. Table 7 has an example of identifier span.

Table 7: Identifier Span Example

Program A	**Program B**
Let Count = 3	Read(Sales)
Read (Sales)	Let Count = 3
If Sales > 10	IF Sales > 10
Let Tax = .5*Sales	Let Tax = .5*Sales
End.	End.

Because there are no statements between the Read statement and the If statement for program A, the identifier span is zero for Sales. For program B, there is a Let statement between the Read and the If statements. Thus, the

[52] This classification is used because it tends to correspond to the temporal problem encountered when trying to fit information into short-term memory.

identifier span is one for Sales. Program B is more complex than program A although both do the same thing. Program B is more complex because it is more likely that the programmer has forgotten what Sales means before coding the If statement.

The purpose of the next section is to introduce a fault maintenance model. While human information processing focuses on how errors get into programs, the fault maintenance model focuses on taking errors out of programs.

3.3 Faults Maintenance Model

Program faults result from programming mistakes. Typically, each program failure is caused by only one program fault. Most COBOL programs are executed many times. Some faults in a program are not located and removed until after the program is put into operation. Maintenance programmers modify applications to either correct a previously undetected fault or provide new information reflecting the changing needs of users. After

---

[53] Elshoff, James L, "An Ulosis of Some Commercial PL/1 Programs," IEEE Transactions on Software, SE-2, Vol 2, (June 1976), pp. 113-120.

a program runs correctly, the process is continued again with new data (Figure 4).[54]

The original program may have had certain faults. However, the number of faults is constantly changing. Modifying a program can create more faults. When a failure occurs and the fault is fixed without introducing new faults, the total number of faults decreases. Thus, at any particular time, a program contains a finite number of faults (although no one knows exactly how many).

Program inputs consist of data, and program failures are triggered by data. For example, suppose that a financial statement program contains unknown faults that keep the program from properly recording the sale of a long-term asset.

---

[54] Musa, J., Anthony Ianniino, Kazxuhira Okumoto, Software Reliability: Measurement, Prediction, and Application, McGraw Hill (1987), p. 235.

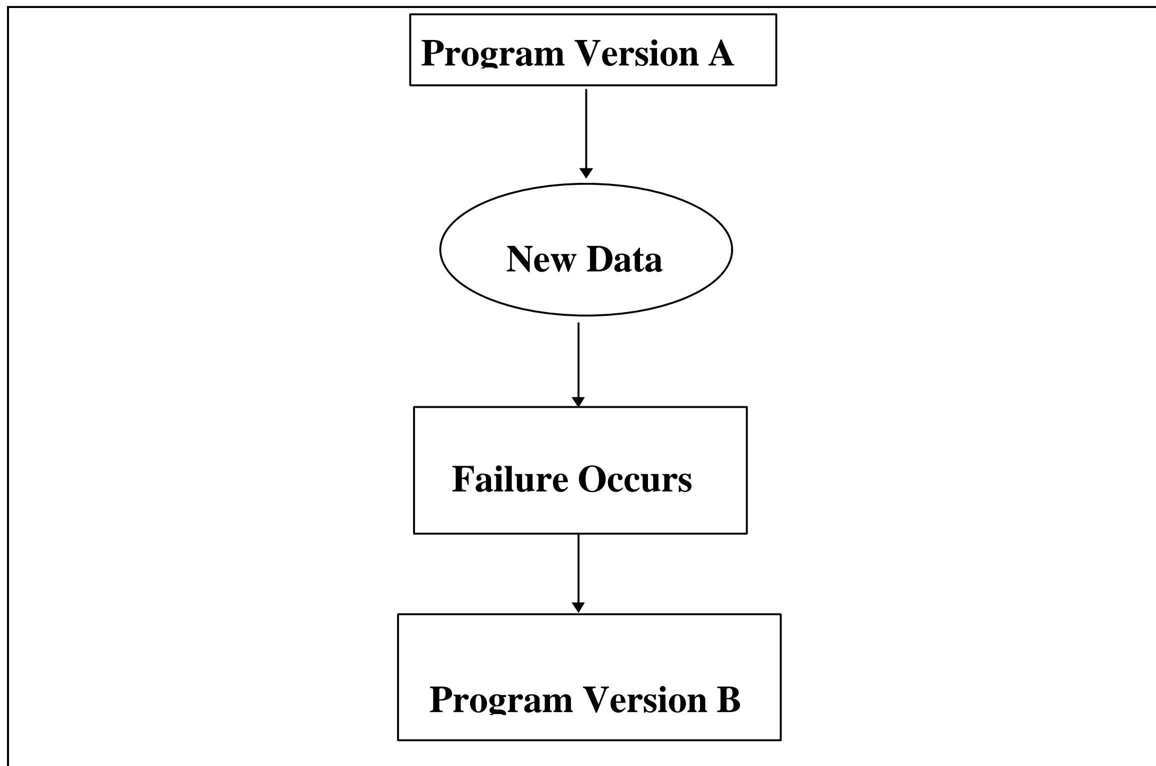


Figure 4: Fault Maintenance Model

A failure will not occur until the sale of a long-term asset transaction is processed. Once discovered, the fault can then be fixed, but the process of removing a single fault sometimes creates other faults. The maintenance process may also be triggered by specification changes. When a specification change occurs the program is modified to produce the desired result. For example, a payroll program may be modified to reflect a change in the tax law.

3.4 Development of Hypotheses

The purpose of this section is to develop a model to predict the reliability of COBOL programs. Each component of software reliability is expressed as an alternative hypothesis. The theoretical model is a combination of the hypotheses derived from the HIP and fault maintenance literature.

3.4.1 Hypothesis One

Because of the continuous maintenance process, programs are often modified because of changes in user requirements or to fix a fault that has been found in the program. When a program is modified there is the possibility that additional faults will be introduced into the program--decreasing reliability[55].

Hypothesis One: The number of program specification changes is inversely related to program reliability.

3.4.2 Hypothesis Two

A programmer can more easily modify a program that is well documented. From the HIP model, a programmer can read an easily accessible form of external memory from a program's documentation that is

simpler to understand than the program itself. It takes less short-term memory to understand the program leaving more short-term memory to make the modification. There is a lower probability that the modification will contain faults. The program will have fewer faults and is more reliable.

Hypothesis Two: Program documentation is positively related to program reliability

3.4.3 Hypothesis Three

A larger program will have more variables and statements for the programmer to remember. Longer programs repetitively saturate short-term memory, and thus, increase the probability of a short-term memory failure. More STM failures increase faults and decrease reliability.

Hypothesis Three: Program size is inversely related to program reliability.

3.4.4 Hypotheses Four

Programs with a complex organization have many relationships between the statements in a program for the programmer to remember. Poor organization uses more short-term memory increasing the probability that a

[55] The operational definition of reliability is the probability of a program executing without abending.

short-term memory (STM) failure will occur. If a STM failure occurs, then the program will contain more faults and be less reliable. For similar sized programs, the program with the simple organization will be more reliable.

Hypothesis Four: Programs' organizational complexity is inversely related to program reliability.

3.4.5 Hypothesis Five

Many COBOL programs are 50,000+ lines long. If size alone decreases reliability then it would be impossible to write a long and reliable program. Many studies in the literature review use size along with other complexity metrics as the independent variables in models. As size increases the potential for additional complexity also increases. All of the complexity metrics are collinear with size. This makes it difficult to test the effectiveness of the other metrics while size is in the model.

If two programs are the same length and both have the same number of faults then it is possible to have a different reliability for each program. A program with a lower cyclomatic complexity[56] has fewer paths, and each fault has a larger probability of producing a failure when a transaction is

[56] Cyclomatic complexity is the number of control flow paths through the program.

processed. Programs with high cyclomatic numbers have many potential control paths. Faults could be hiding in infrequently executed control paths causing few failures.

The example Table 8 contains two programs, each with one fault. The first program incorrectly adds federal tax to gross income when computing net pay. Program two prints out “Excellent” every time a GPA is less than 2.0 when it should print “Probation.” Although both are the same length and have the same number of faults the reliability for each is different. Program one will never compute the correct net pay and has a reliability of 0%. Program two will produce incorrect information only when a GPA under 2.0 is processed. In all other circumstances it produces the correct output. So, program two has a higher reliability than program one.

Table 8: Reliability for Two Programs

Program One	**Program Two**
Read(GR)	Read(NAME,GPA)
ST=GR*.08	If GPA < 2.0
FED=GR*.33	Print(‘Excellent’)
FICA=GR*.07	Else
NET=GR+FED-ST-FICA	Print(‘Good Standing’)
PrintT(NET)	Print(NAME)

Reliability is more independent of size than faults. Reliability is dependent upon the number of faults and the structure (number of paths) of the program. When similar programs are created within the same environment, the program that is longer will have more faults and more paths. Since the paths will tend to hide some of the faults the programs will not necessarily have a different reliability. Thus, the effect of size on a program's reliability is affected by the program's structure.

When a program is efficiently organized more short-term memory is left to hold a larger program. Thus, programmers can create long and reliable programs when the programs are efficiently organized. The effect of size on reliability changes depending upon how well the program is organized.

Hypothesis Five: Larger programs decrease reliability more with complex than simple program organizations.

3.4.6 Hypothesis Six

When two references to the same variable are far apart the programmer might forget exactly what the variable means when using it the latter time. When the program is modified, it is more likely that inserting code between

the references will create additional faults in the program--decreasing reliability.

Hypothesis Six: Programs with greater distances between uses of each identifier are less reliable.

3.4.7 Hypothesis Seven

Both the basic and logarithmic Poisson fault intensity models (Chapter 2.1) predict that failure intensity of a program will decrease with use because faults are located and fixed. Reliability is the inverse of failure intensity. As failure intensity decreases, reliability increases. Thus, the reliability of a COBOL program should increase as it is executed, faults are found and fixed, and it is reexecuted.

Hypothesis Seven: Programs that are executed fewer times are less reliable.

3.5 Metrics to Test Hypotheses

This section takes the theoretical concepts and converts them into variables that can be quantitatively measured in preparation for statistical analyses. This analysis is divided into two sections. The first section develops a way to measure the dependent variable (reliability) while the second develops a way to measure the independent variables (complexity).

3.5.1 Measurement of Reliability

The dependent variable is reliability. A program is a set of instructions that a computer follows to accomplish a task. A failure occurs whenever an abnormal end occurs. The operational definition of reliability for this study is the probability of a program executing without an abend occurring. For this study reliability was computed by the following formula.

$$\text{Reliability} = \frac{\text{Number of times a program runs without abend occurring}}{\text{Number of times a program was executed}}$$

3.5.2 Measurement of Specification Changes

Reliability = F(Specification Changes)

Hypotheses one states that the number of specification changes is inversely related to reliability.[57] A specification change occurs when a program is modified because of a change in the needs of users. Increases in specification changes are expected to decrease reliability.

---

[57] Takashashi (1988) found that increasing specification change activity was associated with increased number of faults in a program. For this research the number of times a program has had a specification change since it was put into operation was tested to find a relationship with reliability.

3.5.3 Measurement of Complexity

According to the Human Information Processing (HIP) model the processing bottleneck for a human programmer writing a computer program is short-term memory. Short-term memory contains what is currently being processed or thought about. It has a limited size of around 7 chunks. Three different events can cause a short-term memory failure, resulting in a fault in a program, and making the resulting program unreliable. First, the program could be so large (size) that it takes more short-term memory than is available. Second, it could be structured so that it takes more memory than is available because of the specific control flow structures used by the programmer. Third, the items that need to be processed in short-term memory could be temporally spaced (distance complexity) making it difficult to keep everything needed to solve the problem in short-term memory.

Structured programming was designed specifically to solve the short-term memory (STM) limitation issue by solving both the size and structural complexity problem. The structured programming methodology involves breaking a problem down into subproblems which are solvable within STM

then coding each subproblem as a separate module.[58] It is not the size of the program that's important but the size of each individual module which a programmer works on at a single time. Structured programming involves writing programs that have simple control flow structures. Only three structures are allowed: selection, iteration, and sequence. A structured program is composed of modules that have only one calling module. A structured program has no GOTO statements or knots.

3.5.3.1 Measurement of Size Complexity

The first type of short-term memory failure results from a program being too large. The size of a program can be measured with size metrics. By far the most common type of size metric is lines of code. The number of verbs is the same as the size of a program in statements since each statement has one verb. Harold found that program size can be used to predict whether or not a program is structured.[59] Rodriguez and Tsai found that program size

---

[58] Conte, Dunsmore, Shen, "Software Engineering Metrics and Models," The Benjamin/Cummings Publishing Company, Inc, (1986), p. 113-120.

[59] Harold, Frederick Gordon, "An Experimental Analysis of COBOL Program Quality Through the Application of Software Metrics to Programs Written with and without Structured Programming Techniques," Unpublished Doctoral Dissertation, George Washington University, (1982).

can be used to predict whether or not a program is complex.[60] Binder and Poore did not find significance when using verbs to predict the quality of documentation for a program.[61] All of these studies measured size with the total number of verbs for the program. This research used the average number of verbs per module because a programmer typically works on one module at a time.

Halstead factored the lines of code metric into operators and operands.[62] In the example X=1+Y. X, 1, and Y are operands and = and + are operators. Halstead thought of programs as being extensions of mathematics and used a similar terminology to look at the size of programs in more detail.

However, COBOL is a different type of language because it is English like. It is not an extension of mathematics, but an extension of the English language. Many people who can't write a program can read a COBOL program and understand what it means. The English language is composed of words that are divided up by parts of speech. The different parts of speech

---

[60] Rodriguex, Volney, and Tsai, W.T., "A Tool for Discriminant Analysis and Classification of Software Metrics," (1988), Systems, Software Development Section.
[61] Binder, L.H., and J.H. Poore, "Field Experiments With Local Software Quality Metrics," Software—Practice and Experience, Vol. 20(7), (July 1990) pp 631-647.
[62] Halstead, M. H., Elements of Software Science, New York: Elsevier North-Holland, 1977,pp25-29.

are stored and processed differently within a person's brains.[63] In developing metrics to study, two things were considered. First, structured programming implies that module size that needs to be considered and not the size of the entire program. Second, the COBOL program should be looked at as a language with different parts of speech which may play a different role in the overall size of COBOL programs.

An example is used to show the different types of size complexity that exist from a COBOL program. The example is a simple module that converts a yearly to a monthly interest rate.

Table 9: COBOL Size Complexity

```
* This is an example COBOL Program.
 Move 12 to months.
 Move .18 to yearly-interest-rate.
 Divide yearly-interest-rate by months giving monthly-interest rate in
 Interest-Variables.
```

Lets look at the example and analyze the parts of speech. "Move" and "Divide" are verbs. Each COBOL statement begins with a verb because the instructions are in command form. The total verbs in a program are not

[63] "The Brain," The Two Brains and Stress and Emotion, Documentary Video by The John D. & Catherine T. MacArthur Foundation Library Video Classics Project (1984).

important, but the average number of verbs per module (structured programming) is important. To measure verbs the average number of verbs per module was computed giving the metric VERBMOD.

Next there are adverbs in the modules. “To,” “giving,” and “by” are adverbs because they support the verb for each statement. Again the total number of adverbs was divided by the number of modules to give ADVBMOD.

There are several types of nouns. These are variable names or identifiers, literal values, and parent names. Identifiers in the example are months, yearly-interest-rate, and monthly-interest rate. The literals are 12 and .18. The parent is Interest-Variables. Each metric is measured by counts per module giving IDMOD, LITMOD, and PARENTS. The “in” word is a preposition. The number of prepositions were not counted because it is the same as the number of parents and would be useless in a statistical model.

In addition to the above COBOL also has comment lines. Comment lines in COBOL have an asterisk in column 7. The number of comments is also computed on a per module basis.

Another metric is MODULES. MODULES is the number of modules in the program. This was put in as a control. If the structured programming methodology was used when writing the programs, then MODULES should have no perceived effect upon reliability.

The length of words can also change the amount of short-term memory required. Longer words are more difficult to learn and use more short-term memory.[64] H. Dunsmore conducted a study to see if longer more meaningful variable names were easier to remember. He found that with longer names it took longer to recall each variable and more mistakes were made.[65] To measure this difficulty in remembering longer names, the metric IDLENGTH was developed in this research. IDLENGTH is the average length in characters of the identifiers and parents in the program.

In summary, the size metrics developed for this study are very similar to those used before; however, they measure more detailed phenomena. The information contained in the number of lines of code is still there, its factored into MODULES and ADVBMOD. Accounting for all of the words in a program is important because people think in words, and the words are the

---

[64] Ibid.

contents of short-term memory. The limited size of short-term memory causes failures when too many words are needed to fit into short-term memory at one single instant.

3.5.3.1.1 Surrogate Metrics for Hypothesis Two

Reliability = F(Comment Lines, Number of Parents, Average length of Identifiers)

Hypothesis two states that the amount of documentation is positively related to reliability. It is impractical to measure all of the forms of external memory that exist when a programmer writes or maintains a program. However, there is some evidence of external memory in a COBOL program itself because programs are documented. Documentation statements can be placed within a COBOL program. The total number of comment lines[66] improves external memory.

---

[65] Dunsmore, H.E., and J.D. Gannon, "Analysis of the Effects of Programming Factors on Programming Effort," The Journal of Systems and Software 1, 2, (1980): 141-153.

[66] Comments was used in the Harold (1982) study which attempted to determine if software metrics have different values for structured and unstructured programs. The comment metric was insignificant. Binder and Poore (1990) used comments to predict the LB ratio {operators/(operands + comments)} which is used as a surrogate for the quality of documentation. The comments metric used for this research is different because instead of using the total number of comments per program, the comments were divided by the number of modules (subroutines) in the program. This was done because programmers typically work on only one module in a program at a time.

COBOL was designed to be self documenting through an English-like command structure and identifier naming. COBOL identifier names are longer and more meaningful than in other languages. Identifiers can have parents that are a part of identifier names. The number of parents[67] is another form of program documentation. A third form of external memory is the average length of the identifier names.[68] All three of the documentation variables (comment lines, parents, and average length of identifiers) are expected to increase reliability.

3.5.3.1.2 Surrogate Metrics for Hypothesis Three

Reliability = F(Verbs/module, Adverbs/module, Identifiers/module, Literals/module, Modules)

Hypothesis three states that program size is inversely related to reliability. Size is a common measure of complexity. Usually it is operationalized as the number of executable statements (periods) in a

---

[67] The number of parents has not been studied before. This is a metric that is unique to the COBOL language. The number of parents is the average number of parents for each identifier (variable).

[68] AvLenId is the average length in characters of the variables used in a program. Since COBOL programs are designed to be self documenting, this measure is used as a surrogate for the information content in the identifier or variable name. This metric was studied by Dunsmore (1980). He found that longer names are more difficult to remember.

program. IF statements can contain many commands or verbs. Table 10 is an example of a COBOL IF block.

Table 10: IF Block

```
IF INCOME > 30000
        MULTIPLY INCOME BY 0.33 GIVING TAX
        ADD TAX TO TOTAL-DEDUCTIONS
        SUBTRACT TOTAL-DEDUCTIONS  FROM
                    INCOME GIVING TAKE-HOME
```

The IF statement contains several verbs (Multiply, Add, and Subtract). Counting the statements (periods) in COBOL would be a biased measure of size because it could miss the complexity inside IF statements. A better measure for the size of COBOL programs is the number of verbs. A verb is a command to do something. It is always the first word in an instruction to do something with data. In Table 10, the words IF, MULTIPLY, ADD, and SUBTRACT are verbs. A verb[69] is similar to Halstead's operator except that

[69] The number of verbs is the same as the size of a program measured by the number of statements since each statement has one verb. Harold (1982) found that program size can be used to predict whether or not a program is structured. Rodriguez and Tsai (1988) found that program size can be used to predict whether or not a program is complex. Binder and Poore (1990) did not find significance when using verbs to predict the quality of documentation for a program. All of these studies used the total number of verbs for

it is more specific. Increasing the number of verbs is expected to decrease the reliability of a program.

Another measure of size complexity is the number of adverbs.[70] An adverb is a helping verb in that it supports the word that starts the instruction. It is a reserved word that is not a verb. In Table 10, BY, TO, and FROM are adverbs. Increasing the number of adverbs is expected to decrease reliability.

The other words in Table 10 are identifiers.[71] An identifier is called a variable in both algebra and mathematical-oriented computer languages. Identifiers are names given to information stored in a computer system. Identifier names are assigned by the programmer. The number of Identifiers is the same as Halstead's operand. Increasing the number of identifiers is expected to decrease reliability.

---

the program. This research used the average number of verbs per module because a programmer typically works on one module at a time.

[70] AdvMod is the average number of adverbs in a module. No other research used this metric. Adverbs are expected to be highly correlated with verbs. I created this metric to account for every word in COBOL programs. The total number of adverbs was divided by modules because a programmer typically works on one module at a time.

[71] IdMod is the average number of identifier (variable) names that occur in a module of the program. This metric has not been studied before, but was created in order to account for every word in a COBOL program. The total number of identifiers was divided by modules to get IdMod because a programmer typically works on only one module of a program at a time.

Literals constitute another form of size. Literals are numeric or character constants used in many COBOL programs. Literals are sometimes referred to as constants in other computer languages. In Table 11, the "7" and "8" are numeric literals, and "John Smith" is a character literal.[72] It is expected that increasing the number of literals decreases the reliability of COBOL programs.

Table 11: Literal Examples

```
ADD 7 TO 8 GIVING TOTAL.
MOVE 'John Smith' TO NAME.
```

Finally, as can be seen from the preceding discussion, there is some controversy over the size metric. Which size is most important, the overall size of the program or the size of each module. From the HIP model, size should be the size of the code that a programmer works on at one particular time to solve one problem.

---

[72] LitMod is the average number of literals in a module. LitMod has not been studied in any other research. It was created in order to account for every word in a COBOL program.

A program can be divided into modules. A module is a subroutine or a group of statements that can be executed by a call from other parts of a program. The statements in a module are highly cohesive and perform one function. A programmer using structured programming techniques works on one module at a time. Thus the number of verbs, adverbs, identifiers, and literals used to measure size complexity should be the average number per module.

The final size is the number of modules.[73] A large structured program has many modules. By using the number of modules measurement, this research does not lose the total size of the program measurement as used in previous studies. Total size is average verbs/module times the number of modules. The information is still in the mathematical model; it is just broken down between two variables.

3.5.3.2 Structure Complexity

[73] Modules is the number of subroutines in a program + 1 for the main routine. This metric was studied by Binder and Poore (1990) although they did not find a significant relationship between modules and documentation quality. Modules in this study is measured the same way. It is the number of modules in the program.

Another type of short-term memory failure occurs when the problem is poorly structured causing it to take up more short-term memory than necessary. Structured programming deals with this problem by having only three acceptable forms of control flow structure. The first is sequential. In a sequential control flow, a statement is executed directly after the statement that it precedes in the source code. The second is selection. Selection in COBOL is implemented with IF statements. When an IF statement is encountered, one to several statements are sometimes executed (or not) depending upon whether or not a condition is true. The selection control flow is more complex than the sequential and will take up more short-term memory. The third type of control flow is iteration. Iteration is looping or repeating the same sequence of code over and over until a condition changes. In COBOL, looping is achieved by using PERFORM UNTIL statements. Iteration is more complex than sequential and uses more short-term memory.

Measuring the amount of complexity from the control flow of iteration and selection statements can be done by measuring the cyclomatic complexity of a program. Cyclomatic complexity is the number of loops and IF statements in a program plus one. Cyclomatic complexity was used by

Binder and Poore to predict documentation quality.[74] However, they found insignificant results with the cyclomatic metric. Basili and Perricone studied the occurrence of software errors and found a significant relationship between errors found and cyclomatic complexity.[75] For this study, it is the cyclomatic complexity inside each module that is important because of structured programming. The CycloMod metric is used to measure the average cyclomatic complexity of each module.

Structured programming has been referred to as GOTO-less programming. The number of GOTO statements has been studied by Binder and Poore.[76] However, they found insignificant results when trying to predict software documentation quality from GOTO statements. For this research, the number of GOTO's in each program was divided by the number of modules, to get GotoMod. The average number per module was used because programmers tend to work on one module at a time and it fits the human information processing model better.

---

[74] L.H. Binder and J.H. Poore, "Field Experiments With Local Software Quality Metrics," Software—Practice and Experience, Vol 20(7), (July 1990), pp. 631-647.

[75] Basili, Victor, and Barry Perricone "Software Errors and Complexity: An Emprical Investigation," Communications of the ACM, (January 1984), Vol 27, no. 1, pp. 42-52.

[76] Ibid, Binder and Poore, pp 631-647.

There is never an absolute requirement to use a GOTO statement in any program. The use of GOTOs imply that the program's statements are not ordered correctly. This wastes short-term memory when trying to understand the program. The metric used to measure GOTO usage is GotoMod. GotoMod is the number of GOTO statements in a program divided by the number of modules. It was divided by the number of modules because a module is what a programmer works on at one time according to structured programming.

GOTO statements also can cause another problem. The control flow lines of the program can cross creating a knot. Knots have been studied before by Woodward.[77] Knots mean that a program is difficult to understand and uses unnecessary short-term memory. Woodward produced a complexity measure which is dependent upon the ordering of the statements in a program. He defined knots as the number of times that the control flow paths in a program cross each other. Woodward tested his metric with 26 FORTRAN subroutines from a numerical algorithms library. He then used the number of

---

[77] Woodward, M.R., M.A. Hennel, and D. Hedley, "A Measure of Control Flow Complexity in Program Text," IEEE Transactions on Software Engineering, SE-5, Issue 1, (January 1979), pp. 45-50.

knots to predict McCabe's cyclomatic complexity metric and found a significant relationship.

For this research the number of Knots was divided by the number of modules to get KnotMod. This was done because programmers tend to work on one module of a program at a time. The KnotMod metric is expected to be correlated with the amount of additional short-term memory that is required to write the program. It is possible to write any program without knots.

Each of the possible control flow structures have been accounted for and measured with the metrics GotoMod, KnotMod, and CycloMod similar to the way that all of the possible words in a COBOL program were accounted for within size.

3.5.3.2.1 Surrogate Metrics for Hypothesis Four

Reliability = F(Cyclomatic Number/module, GOTOs/module, Knots/module, Module Call)

Hypothesis four states that programs' organizational complexity is inversely related to reliability. COBOL is a link between instructions computers follow and instructions people write. A poorly-structured COBOL program will also be difficult to organize into short-term memory. When cyclomatic complexity is high the program is more complex and is expected

to have a lower reliability.[78] Cyclomatic complexity can be computed two different ways. The original McCabe method requires that when a procedure is called multiple times from within a calling module that the complexity for the called module be weighted by the number of times it is called. The Henderson-Sellers and Tegarden (1994) method does not inflate the measure of psychological complexity because each module's complexity is included only once even if it is called many times. For this research, the cyclomatic complexity was always measured once for each separate module which resembles the Henderson-Sellers and Tegarden technique.

Knots make a program difficult to understand because the statements are not in the order that they will be executed.[79] Increases in the number of knots are expected to decrease reliability.

Structured programming is GOTO less programming.[80] Although the structured programming methodology is much more than eliminating the need

---

[78] For this research cyclomatic complexity was divided by the number of modules because a programmer typically works on only one module of a program at a time.

[79] For this research the number of Knots was divided by the number of modules to get KnotMod. This was done because programmers tend to work on one module of a program at a time.

[80] The number of GOTO statements has been studied by Binder and Poore (1990). However, they found insignificant results when trying to predict software documentation quality from GOTO statements. For this research the number of GOTO's in each program was divided by the number of modules to get GotoMod. The average number per module

for GOTO statements, because any program can be written without GOTO statements. Having a GOTO statement in a program implies that the statements are not already in the correct order to be executed. When a program is poorly ordered, it may also be less reliable.

Another rule of structured programming is that a subordinate module cannot have more than one calling module. Module call complexity is having multiple modules call a procedure. Module call complexity can be identified from a program's structure chart. Figure 5 contains an example of module call complexity. The submodule with an "X" is called from more than one parent module.

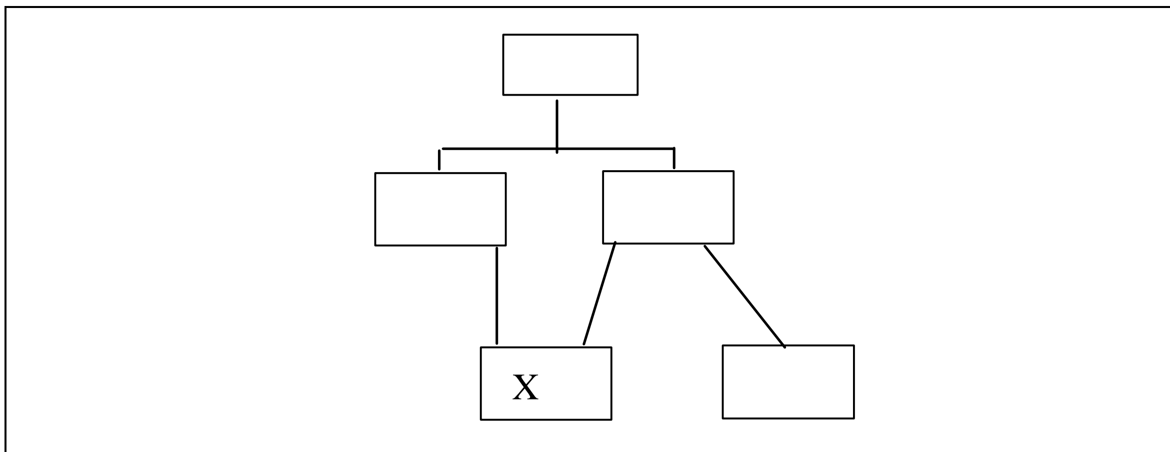


Figure 5: Module Call Complexity

[80 continued] was used because programmers tend to work on one module at a time and it fits the human information processing model better.

Module call complexity is common in COBOL programs.[81] Multiple called modules are usually called utility modules. Specifically, module call complexity for a subordinate module is the number of modules that can potentially call the subordinate module minus 1. For a program, module call complexity is the sum of all module call complexity measurements computed for all modules in the program.

It is expected that increases in module call complexity decrease reliability. Like size complexity, structural complexity should only include the part of a program worked on at once. As above, the metrics are the average per module.

3.5.3.2.2 Surrogate Metrics for Hypothesis Five

Reliability = F (Total Size*Total Structure)

Hypothesis five states that larger programs decrease reliability more with complex than simple organizations. The size and structure relationship can be measured with an interaction variable. The reliability of a program is

[81] ModCall is the average number of calling modules that call each specific called module. If a program is structured it should not be over one. This metric has never been studied before.

dependent upon the interaction between the total size and total cyclomatic complexity.

A potential problem with the size and structure interaction is the existence of many size and structure metrics. Statistical methods can combine the metrics into total size and total structure before computing the interaction. Increases in the interaction variable are expected to decrease reliability.

3.5.3.3 Temporal Complexity Metrics

The third type of short-term memory failure occurs when the items to be remembered are spaced far apart in the program making it difficult to remember what something means when it is encountered again within the program. This metric was referred to as distance complexity by Fainter.[82] The distance is usually measured as the distance between the uses of the nouns in a program. In this study, both variable names and procedure names were studied. The distance of variable names was suggested by Fainter. The distance of procedure names is implied because it is a noun that is sometimes repeated in a program.

---

[82] Fainter, Robert, "Program Complexity Measures," Unpublished Masters Thesis, Virginia Tech, (March 1981).

The metrics for measuring the temporal complexity of a program were defined by Fainter as a measure of distance. However, from structured programming we know that the programmer is working on a single module at a time. For this research two separate distance metrics were designed for program variables. IntraMod is the average number of statements between each use of a variable within a module. InterMod is the average number of modules which use each variable name. The total distance complexity metric proposed by Fainter is still in this study; however, it is factored into more detail as required by structured programming.

Similar to the way that the distance was measured for variables, this study measures the distance for procedure calls. ModCall is the average number of modules that call each individual module in the program. If a program is structured this number should never be more than one. CallSpan is the average number of statements between each call to another subroutine within a single module of the program. The ModCall and CallSpan together make up the total distance complexity for procedure calls. The two variable distance and the two procedure distance metrics together account for all of the distance complexity in the program because of nouns referring to specific

data items or code segments. ModCall was not measured as a temporal metric because it was already included as a structure metric. It is expected that any structured program would have a ModCall complexity of 0 since one of the goals of structured programming is that each module is called by only one other module.

3.5.3.3.1 Surrogate Metrics for Hypothesis Six

Reliability = F (Call Span, Intermodule, Intramodule)

Hypothesis six states that programs with greater distances between uses of each identifier are less reliable. Temporal complexity is the difficulty of getting items into short-term memory because the problem is temporally spaced. The items that a programmer remembers are identifiers (data names) and procedure names.

However, a potential problem exists with temporal complexity. What if there are two references to the same variable and each reference is in different modules? In the COBOL program, other modules may be between these two modules in the source code. Thus, distance complexity is going to be greatly inflated because of the "between" modules although the programmer might have written the two modules containing the identifier

reference consecutively. The temporal complexity for identifiers within a module is intramodule span.[83]

The complexity of having references to the same identifier in different modules can be measured with a different metric called intermodule span.[84] Intermodule span is the number of modules referring to the same identifier. Increases in intermodule span are expected to decrease reliability.

As mentioned earlier, call span is caused by trying to remember what each module does. Call span is similar to intramodule span except that it deals with procedure call statements. Specifically, call span is the distance between two references to the same procedure within a single module.[85] Increases in call span are expected to decrease reliability.

3.5.4 Measurement of Times Executed

Reliability = F(times executed)

---

[83] IntraMod is similar to distance complexity. Distance complexity was defined by Fainter (1981). Fainter defined it as the distance in statements between two uses of the same variable. IntraMod is the distance in verbs between uses of identifiers (variables).
[84] InterMod is the average number of modules that use each identifier or variable. If a program is highly structured it should not be over one. However, this metric has never been studied before.
[85] CallSpan is similar to distance complexity. Distance complexity was first defined by Fainter (1981). Fainter defined it as the distance in statements between two uses of the same variable. Distance complexity has never been studied quantitatively. CallSpan is the distance in verbs between calls to other modules. CallSpan has never been studied before.

Hypothesis seven states that programs that are executed fewer times are less reliable. This hypothesis is tested by combining two different metrics. The first metric is the number of years that a program has been in operation. The other is periodicity. Yearly periodicity is the number of times a program is executed in a year. Times executed is the yearly periodicity multiplied by the number of years that the program has been in operation.[86] The more times a program is executed the more opportunities there are to find and fix the faults in the program. Reliability is expected to increase with the number of times a program is executed

3.6 Mathematical Model

All of the above metrics from each of the hypotheses have been combined to form the model for this research. The model's dependent variable is reliability. The independent variables have been discussed for each hypothesis and come from human information processing and

---

[86] Musa (1987) states that the execution time for a program tends to be associated with decreased number of faults. His sample size was 213. This leads to the conclusion that increasing executions will be associated with increased reliability. For this study, it is the number of times that a program has been executed since being put into operation that is measured.

fault/maintenance theory. The positive relationships and negative relationships are indicated with mathematical symbols

Table 12: Mathematical Model

Reliability = F(	**+ Comments**	**+ Parents**	**- VerbMod**
	- AdvMod	**- LitMod**	**+ AvLenId**
	- IdMod	**- Modules**	**- CycloMod**
	- GotoMod	**- KnotMod**	**- ModCall**
	- SpecChg	**- CallSpan**	**- InterMod**
	- IntraMod	**+ TimExec)**	

3.7 Statistical Tests

The hypotheses all relate one variable (reliability) to other variables or program characteristics. The type of relationship is uncertain. It could be either linear or non-linear, but the hypothesized relationships will be either increasing or decreasing. A correlation test is used to determine whether or not two variables have an overall trend relationship with each other. Pearson's correlation test assumes that the relationship between two variables is linear. Spearman's correlation test assumes that the ranks of the data are correlated and requires more data for the same level of statistical

confidence.[87] Spearman's test is nonlinear and fits the characteristics of the data in this study. Spearman's test is used to test each hypothesis.

Hypothesis five uses two metrics (total size and total structure) that are not directly computed from the COBOL program's source code. Total size and total structure are computed from a principle component model developed with principle component's analysis. Principle components analysis is a method that allows different variables to be combined into one variable. This is necessary so that the interaction between total size and total structure can be computed. The independent variables identified in the mathematical model are used to develop a statistical model to predict reliability. The statistical model is developed using a regression technique. The data may need to be transformed. Because the number of independent variables is large, a technique such as stepwise regression can be used to provide a model with only significant independent variables. The statistical model is validated by using a holdout sample.

[87] Lyman Ott, "An Introduction to Statistical Methods and Data Analysis," Second Edition, PWS Publishers, 1984, p.260.

Chapter 4
Results

Both predictive and explanatory methods are used to relate program reliability to the metrics. Two separate samples are gathered: one for the explanatory methods and the other for prediction. The explanatory (testing of hypotheses) uses the entire sample. The predictive model use a sample that contains the remaining entire sample after the holdout sample is removed. The holdout sample was selected from the entire sample by using systematic random sampling. After principal compenents analysis, the hypotheses are tested using Spearman's correlation. A predictive model is developed using regression. The predictive model is tested using a holdout sample.

4.1 Data Gathered

The data source is a large insurance company. The data are borrowed and must be returned to the company after the completion of this research. To use the mathematical model for predictive purposes as well as test the hypothesis the source code for the COBOL programs, and each program's run history are gathered.

The organization provided the researcher with a sampling frame of 2,467 periodically executed COBOL programs that were in production on April 1, 1991. Two separate samples were created. The sample of nonfailure programs is a stratified random sample of 150 programs randomly chosen from the sample frame. A second sample, the failure sample consists of all of the programs that failed at least once during the period April 1, 1991 through March 31, 1992. Eighty programs constitute the failure sample. Table 13 shows the frequency distribution for the total sample.

Table 13: Reliability Frequency Distribution for Total Sample

Reliability	Percent
0-20	3.9%
21-40	2.6%
41-60	3.5%
61-80	9.3%
81-90	6.6%
91-95	6.2%
95-100	67.9%

Because the goals of this research are both explanatory and predictive, two different sampling techniques are used. The explanatory sample includes all of the programs collected. For explanatory research the goal is to determine the effect of the independent variables (complexity) on the

dependent variable (reliability). For the explanatory research, the two samples (failure and nonfailure) are compared using a correlation analysis.

The predictive sample contains all of the programs gathered except for those selected for the holdout sample. The holdout sample is used to check the accuracy of the model developed from the predictive sample..

For each program the specific type of information gathered are:

1. The number of times the program was executed during the sample period.

2. The number of times the program failed during the sample period

3. The source code for the program at the beginning of the sample period.

4. The number of specification changes that have occurred since the program was put into operation.

Statistics were computed for both the failure and nonfailure samples and the total sample. Table 14 contains the descriptive statistics for the failure sample.

Table 14: Descriptive Statistics of Failure Sample of COBOL programs

METRIC	Minimum	Maximum	Mean	Std Dev
Comments	0	1384.00	211.63	343.47
Parents	0	297.00	30.63	45.12
VerbMod	2.67	20.00	7.88	4.10
AdvMod	2.78	27.11	12.20	7.00
LitMod	0.12	21.40	5.24	4.98
AvLenId	0	16.13	9.92	3.58
IdMod	4.75	45.00	15.40	9.29
Modules	0	215.00	11.16	28.99
Cyclo	1.00	277.00	14.26	35.23
GotoMod	0	1.55	0.36	0.43
KnotMod	0	38.40	6.59	10.25
ModCall	0	3.79	0.54	0.82
SpecChg	0	161.00	5.62	16.00
CallSpan	0	4.47	0.39	0.93
InterMod	0	2.44	0.58	0.84
IntraMod	0	3.05	0.39	0.69
TimExec	1	3134.00	282.10	573.81

By examining the metrics, the metrics all have reasonable values. Both times executed and cyclomatic complexity have a minimum value of 1 or more. All the other metrics have values that are 0 or more. This provides further evidence that the METRICS program worked correctly. Table 15 is the descriptive statistics for the nonfailure sample.

Table 15: Descriptive Statistics of the Nonfailure Sample of COBOL Programs

METRIC	Minimum	Maximum	Mean	Std Dev
Comments	0	686.00	89.41	124.15
Parents	0	119.00	18.58	19.49
VerbMod	2.60	23.78	8.74	4.84
AdvMod	2.81	25.84	11.35	6.37
LitMod	0.60	17.03	4.44	3.66
AvLenId	0	18.25	11.17	3.29
IdMod	3.70	47.73	16.16	9.81
Modules	0	128.00	13.00	22.31
Cyclo	1.00	185.00	20.77	35.06
GotoMod	0	1.21	0.20	0.25
KnotMod	0	72.43	8.45	13.88
ModCall	0	3.13	0.76	0.80
SpecChg	0	46.00	3.92	6.99
CallSpan	0	17.43	0.58	1.88
InterMod	0	3.34	0.82	0.86
IntraMod	0	9.64	0.63	1.14
TimExec	1	3134.00	254.96	553.55

The nonfailure sample contains metrics with reasonable values. Similar to the failure sample, both times executed and cyclomatic complexity have minimum values of 1 or more. All of the other metrics have positive minimum values. This adds further evidence that the METRICS program works correctly.

4.2 Testing of Hypothesis

The hypotheses are tested using correlation analysis. First, a table is presented of the Spearman's correlations and p-values for each metric and reliability (an '*' means significant at the .10 alpha level).

Table 16: Spearman's Correlation Table of Metric with Reliability

METRIC	Correlation	P-value (1 tailed test)
Hyp 1:		
SpecChg	-0.085	0.1005
Hyp 2:		
Comments	0.075	0.1449
Parents	0.090	0.1008
AvLenId	-0.103	0.0732*
Hyp 3:		
VerbMod	-0.147	0.0136*
AdvMod	-0.127	0.0288*
LitMod	-0.125	0.0307*
IdMod	-0.134	0.0222*
Modules	-0.019	0.3910
Hyp 4:		
Cyclo	-0.037	0.3040
GotoMod	-0.082	0.1092
KnotMod	-0.165	0.0067*
ModCall	-0.056	0.2114
Hyp 6:		
InterMod	-0.047	0.2507
IntraMod	-0.050	0.2394
CallSpan	-0.050	0.2420
Hyp 7:		
TimExec	0.296	0.00005*

Now each hypothesis will be examined by comparing each metric's correlation with reliability.

4.2.1 Correlation Test for Hypothesis One

Reliability = F(Specification Changes)

The level of significance required for this study is set at a predetermined alpha level of 0.10. Hypotheses one states that the number of specification changes is inversely related to reliability**.** Although the sign of the correlation was in the right direction, the correlation of reliability with changes has a p-value of 0.1005. This study failed to find a significant relationship between reliability and specification changes.

4.2.2 Correlation Test for Hypothesis Two

Reliability = F(Comment Lines, Number of Parents, Average Length of Identifiers)

Hypothesis two states that documentation is positively related to reliability. It is impossible to measure all of the forms of external memory that exist when a programmer writes or maintains a program. The p-value for the correlation of comment lines with reliability is 0.1449; thus, no significant

relationship was found. The p-value for the correlation of number of parents with reliability is 0.1008. Again, we fail to find a significant relationship. The p-value for the correlation between average length of identifiers and reliability is 0.0732 which is significant. Although this study failed to find a significant relationship with comments and parents to reliability, a significant negative relationship was found between identifier length and reliability.

4.2.3 Correlation Test for Hypothesis Three

Reliability = F(Verbs/module, Adverbs/module, Identifiers/module, Literals/module, Modules)

Hypothesis three states that program size is inversely related to reliability. The correlation between verbs/module and reliability has a p-value of 0.0136. This is a significant p-value. The correlation value is -0.148 which implies a negative relationship as hypothesized.

The correlation between adverbs/module and reliability has a p-value of 0.0288--a significant value. The correlation has a value of -0.127. A negative significant negative relationship exists between adverbs/module and reliability as hypothesized.

The literals/module metric's correlation with reliability was -0.126. The p-value was 0.0307 which was significant. Therefore this study found a significant relationship between literals/module and reliability.

The correlation between the number of modules and reliability had a p-value of 0.3910 which is not significant. This study failed to find a significant relationship between modules and reliability.

The results of this hypothesis are encouraging. The verbs, adverbs, and literals per module were all significant and in the right direction (decreasing reliability). Modules was, however, insignificant. This implies that to optimize reliability each module in a program needs to be kept to a reasonable size; it doesn't appear to matter how many modules are in the program.

4.2.4 Correlation Test for Hypothesis Four

Reliability = F(Cyclomatic Number/module, GOTOs/module, Knots/module, Module Call)

Hypothesis four states that programs' organizational complexity is inversely related to reliability. Although all of the correlations for the metrics for hypothesis four were in the right direction (negative), only one of them

was significant--knotmod. Knotmod had a p-value of 0.0067 with a correlation of -0.166. The negative direction was as anticipated. This study found a significant relationship between knotmod and reliability. This study failed to find a significant relationship between reliability and cyclomod and gotomod. This implies that the most significant aspect of structure is knots with respect to reliability. The number of knots per module decreases reliability.

4.2.5 Correlation Test for Hypothesis Five

Reliability = F (Total Size*Total Structure)

Hypothesis five states that larger programs decrease reliability more with complex than simple organizations. Hypothesis five requires that total size be compared with total structure. The size metrics need to be combined into one single metric, and the structure metrics need to be combined into another single metric. A principle components analysis was used to find an equation that combined the metrics.

Principle components analysis was applied to the full sample and one factor was found. Table 17 contains the factor with the coefficients. The

coefficients were used to compute the combined size metric for the sample. All of the coefficients were about the same.

Table 17: Coefficients for Combined Size Metric

METRIC	Coefficients
IdMod	0.265
AdvbMod	0.265
VerbMod	0.265
LitMod	0.246

Similarly, principle components analysis was performed on the structure metrics. The coefficients for the combined structure were:

Table 18: Coefficients for Combined Structure

METRIC	Coefficients
Cyclo	0.509
KnotMod	0.454
GotoMod	0.413

The combined structure metric was computed for each of the observations in the holdout sample. The coefficients were again all similar, so the combined structure metric is similar to a simple average of the structure metrics.

The combined size and structure metrics and their interaction were correlated (Spearman's) with reliability. There is an overall correlation with the size metric. The larger the size of a program the less reliable it becomes. The overall structure and interaction were not significant.

Table 19: Correlation for Interaction

METRIC	CORRELATION	P-Value
SIZE	-0.1331	0.047
STRUCTURE	-0.0436	0.540
INTERACTION	-0.0427	0.548

4.2.6 Correlation Tests for Hypothesis Six

Reliability = F(intermod, intramod, callspan)

Hypothesis six states that programs with greater distances between uses of each identifier are less reliable. The correlation of reliability with callspan had a p-value of 0.2420, therefore, this study failed to find a relationship between callspan and reliability. The correlation of reliability with intermod had a p-value of 0.2507, and this study failed to find a significant relationship. The correlation between reliability and intramod had a p-value of 0.2394. Again, this study failed to find a significant relationship

between intramod and reliability. Because all three metrics had a negative result, this study failed to find a significant relationship between temporal complexity and reliability.

4.2.7 Correlation Test for Hypothesis Seven
Reliability = F(TimExec)

Hypothesis seven states that programs that are executed fewer times are less reliable. The Spearman's correlation of timexec with reliability had a p-value of 0.00005. The coefficient was 0.297. Therefore as this study anticipated, a significant positive relationship was found between reliability and times executed.

4.3 Predictive Model

A holdout sample of 47 observations was randomly taken from the entire sample. The remaining observations were used to develop the statistical models. First, a full statistical model with all of the independent variables in the mathematical model was developed using linear regression. The full statistical model had a p-value of 0.111 and an R^2 of 0.24. This model contained many insignificant independent variables. Next, predicted reliability was computed for each observation in the holdout sample. The

average prediction error was 0.754 with a standard deviation of 0.320. This regression model is biased (the average prediction error is above 50%). First the data may need to be transformed, and a technique used to remove the insignificant independent variables from the statistical model.

4.3.1 Transposed Stepwise Model

Whenever a variable is a probability with each observation potentially having a different number of times executed the ARCSIN(SQRT(Y)) is an appropriate transformation to use.[88] This transformation was applied to reliability before developing the stepwise model.

The full statistical model contains many independent variables. Many of the variables may not be necessary to predict reliability. Forward stepwise regression was used to find a model with all significant independent variables. Independent variables were added in decreasing order of significance. Independent variables were added to the model until an insignificant independent variable was encountered. The stepwise model contains only significant (p-value <= 0.10) independent variables. The statistical model has a p-value of 0.0001,an R^2 of 0.156, and an adjusted R^2 of 0.132. The

---

[88] Steel, Robert and James Torrie, Principles and Procedures of Statistics, New York, McGrawl Hill, (1980), p. 150.

average forecast error was -0.060 for the model sample and -0.094 for the holdout sample**.** Table 20 contains the coefficients and p-values for the transposed stepwise model.

Table 20: Transposed Stepwise Model

Variable	**Coefficient**	**P-value**
Intercept	1.1742	0.0001
Parents	0.0019	0.0351
AdvbMod	0.0143	0.0230
KnotMod	-0.0116	0.0022
IntraMod	-0.0743	0.0598
Times	0.0002	0.0017

The residuals for the stepwise regression model were graphed for both the model sample and the holdout sample. The stepwise model sample residuals were compared to the normal curve with a chi-square test. The chi-square test had a p-value of 0.189 which implies that the residuals distribution is not significantly different from the normal curve as expected.

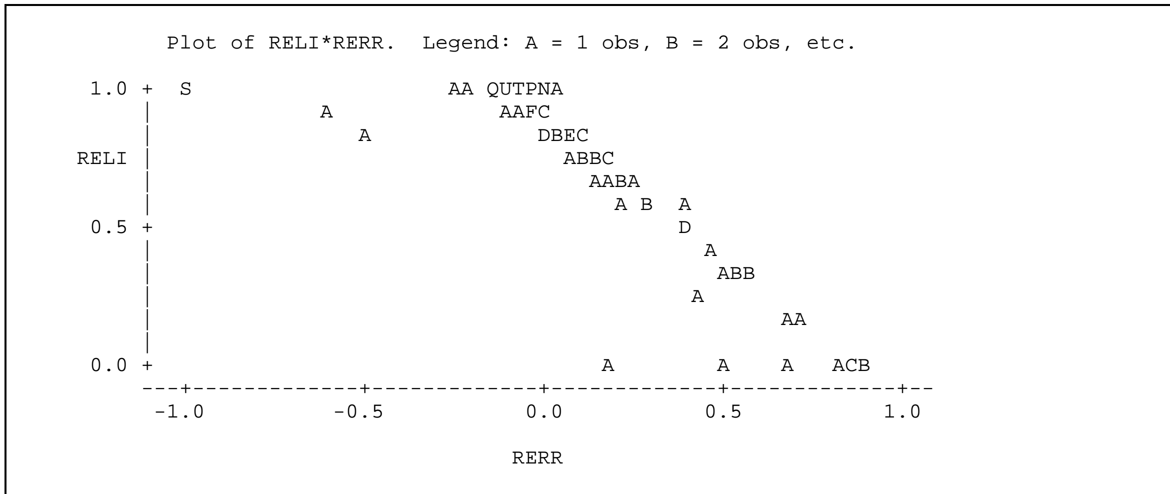


Figure 6: Plot of Residuals for Model Sample of Stepwise Model

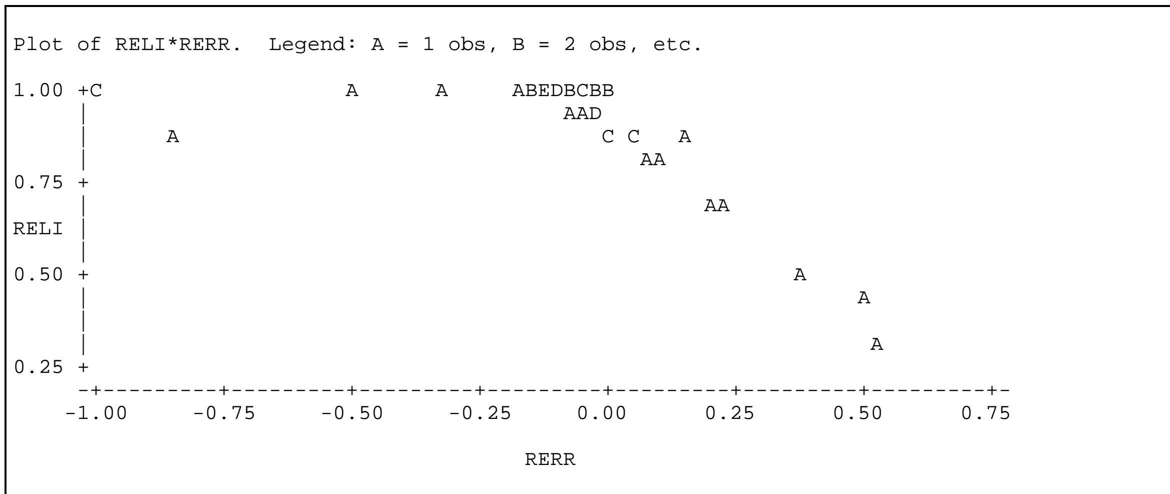


Figure 7: Plot of Residuals for Holdout Sample of Stepwise Model

Figures 6 and 7 show the residual plots for the stepwise model. The frequency distribution of both models appear to be normally distributed because the data bunch in the center and there are few observations on the ends. This also implies that the error for most predictions would be low while it would be higher for only a few observations.

Chapter 5

Summary, Limitations, and Conclusions

This section contains the summary, limitations, and implications for future research, and applications of this study to industry. Although all of the results were not anticipated, they are interesting and add to the current body of knowledge.

5.1 Summary of Results

From the hypothesis tests, the size of a module in verbs, identifiers, adverbs, and literals are significantly inversely related to reliability. However, size as measured by the number of modules is not related to reliability. Thus, when a programmer writes a large program, the size of each module should be kept small, but the number of modules can be large.

A measure of structure based on program knots was inversely related to the program reliability. This implies that programs are made more reliable by using knotless programs.

Also, as predicted from the literature the more times a program is executed the more reliable it becomes. This was the most significant relationship found.

Two different prediction models were developed. The full untransposed model didn't predict well, having a prediction error of 75%. The model was transposed and the most important independent variables were selected, using stepwise regression, to produce a significant model with a 9% prediction error. This model would be useful to internal or external auditors who need to measure the reliability of a system or program. The reliability of each program could be predicted from the model, then combined to find the system reliability.

For example, suppose that a system is composed of program A and program B. The input for program A is gathered at the source, and the output from program A is the input for program B. The output for program B is used by a manager for decision making. If program A is 95% reliable and program B is 80% reliable, then the system is (95%*80%) 76% reliable. Also, by knowing that program B is 80% reliable, system reliability can be improved by at most 20% by modifying program B. System reliability could only be

improved by at most 5% by modifying program A. So, a prediction model from program reliability can help a manager decide the reliability of the information in the reports, and it could also help a systems professional decide which programs need to be improved.

5.2 Limitations

The most limiting factor of this study was the sample. A very large sample taken from a multi-organizational and multi-industry population could not be gathered because of the time and financial constraints. All of the sample came from a single organization with a single set of programmers. The organization was in a single industry. The organization had a single set of programming standards.

Another limiting factor of this study is the language selected. Although COBOL is still heavily used for main frame-based administrative systems, some organizations no longer use COBOL for new applications. This limits the usefulness of the results to aid programmers in industry. However, the findings of this study may also hold true for other programming languages.

5.3 Implications for Future Research

The human information model can be applied to other areas. It has been applied to making computer manuals more easy to understand,[89] to make advertising more effective,[90] and to make computer interfaces easier to use among other things.[91] This research is another successful implementation of the human information processing model. In the future, this research could be extended to other unresearched computer languages and to languages that are evolving.

This study could be applied to other computer environments and languages. Although COBOL was first standardized in the late 1950's, it is still used and continually updated. COBOL is going to evolve into an object-oriented language.[92] This research can then be continued by developing several new metrics to deal with object-oriented COBOL programs, writing a new METRICS program, and gathering a new sample.

---

[89] Stahl, Bob, "Friendly Mainframe Software Guides Users Toward Productivity," Computerworld-. Feb 3, 1986, v20n5, pp. 53-66.
[90] Madden, Charles, "Marketers Battle for Mind Share," Baylor-Business-Review, Spring 1991; v9, pp. 8-9.
[91] Gingrich, Gerry, "The Heart Has Its Reasons That Reason Does Not Know," Journal-of-End-User-Computing. Winter 1995, v7n1, pp. 24-25.
[92] Shelly and Cashman, "Structured COBOL Programming," Roy Forman Boyd & Fraser Publishing Company, (1996)pp. 1-2.

Another group of programming languages are the non-procedural expert systems and database languages. New metrics could be developed for languages like PROLOG, LISP, Dbase, and SQL and used to aid programmers. New models that predict the reliability of programs in these languages could be developed.

5.4 Application of Results

The results from this research can be used to help people write better programs. The size of the module negatively affects reliability while the total number of modules doesn't affect reliability.. Also, knots negatively affect reliability. These results imply that structured programming facilitates building programs with increased reliability.

The statistical model developed in this study can be implemented as a computer program. A user-oriented windows application that reads a COBOL program and predicts the program's reliability may be useful for systems analysts and auditors.

Systems analyst are concerned with both the reliability and maintainability of their systems. However, reliability and maintainability are not independent. Unreliable programs require more maintenance. The

predictive reliability model could be used by systems analyst to point out the programs in a system that are unreliable. Being able to predict reliability is especially useful for programs that don't have long run histories from which reliability could be calculated.

Both internal and external auditors need models to predict the reliability of computer programs. Internal auditors are concerned with the reliability of an organization's records. In a computerized environment the records are generated by computer programs The reliability prediction model can be used by the auditor to determine the reliability of information generated by computer programs.

External auditors are concerned with attesting to the financial statements of an organization. To make decisions about the financial statements, the internal control for the organization being audited is analyzed. The reliability prediction model can be used by external auditors to predict the reliability of computer programs and systems. The reliability predictions can then be used to determine the nature and extent of audit testing.

Appendix A

The METRICS program

The METRICS program's purpose is to compute various metrics from COBOL source code. The metrics program creates an output file that contains the measured metrics for all of the COBOL programs that have been used in this project. METRICS was written with the Pascal computer language and implemented on a PC using a Turbo Pascal Compiler. The system is designed to run a batch of COBOL programs at one time. The METRICS program processes tokens. A token is a word or symbol used in a programming language. Tokens are usually separated by spaces or other tokens. A printed copy of the METRICS source code is in the appendix. Figure 8 contains a hierarchy chart for the METRICS program.

Metrics Program

- **Process Program**
  - **Get Token**
  - **Process Token**
    - **PushSwitch**
    - **PushProcedure**
    - **PopSwitch**
      - **PushNot**
    - **DestroySwitch**
  - **SearchNot**
  - **PrintIdQueue**
  - **PrintNot**
  - **ComputeIdDistance**
  - **ComputeIdSize**
  - **DestroyIdQueue**
  - **ComputePrdistance**
  - **PrintStruc**
  - **RidThru**
    - **InsertCall**
  - **PutJumps**
    - **InsertJump**
    - **InsertMod**
    - **FindKnot**

Figure 8: Hierarchy Chart for Metrics Program

A.1 Data Structures

The data structures in the program are all implemented as linked lists. There are both stacks and queue data structure types. Stacks hold data using a LIFO scheme, while queues use a FIFO scheme. The Identifier Queue holds all of the identifiers in the COBOL program. The identifiers are put into the data structure when the METRICS program is processing the COBOL program's data division.

The Procedure Name Stack holds the names of all of the modules in the COBOL program. It is needed to compute the verbs per module and to find the number of knots in the program.

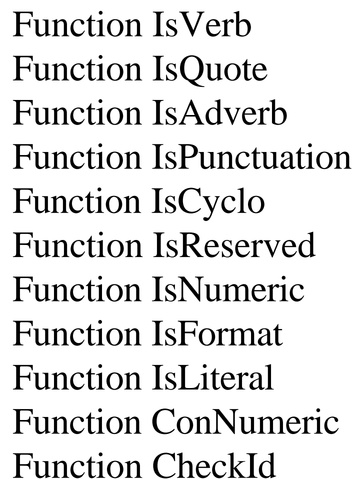
The Procedure Switch Stack holds identifier names temporarily and is used to reverse the order of the identifier and its parent. In the procedure division parent names follow identifier names. In the data division the parent names come first. This stack is used to switch the identifier and parents so that the comparing will be in the correct order when searching the ID queue for the identifier (token) found in the procedure division.

The Not Found Stack is used to hold all tokens not found in the procedure division of the COBOL program being processed. The Not Found

Stack can then be printed out. If something is in the Not Found Stack then an error has occurred.

A.2 Functions

First, the functions contained in the METRICS program are described. Functions are used to identify commands built into the program by the programmer. They are called like built in commands but have different names. They can be called from anywhere on the hierarchy chart of the program and are not modules.



Function IsVerb
Function IsQuote
Function IsAdverb
Function IsPunctuation
Function IsCyclo
Function IsReserved
Function IsNumeric
Function IsFormat
Function IsLiteral
Function ConNumeric
Function CheckId

Figure 9: List of Functions

The function IsVerb returns either a true or a false depending upon whether or not the submitted token is a verb. This function is used to identify

the verbs in the program. A verb in COBOL is usually the first word in a statement.

The function IsQuote returns either a true or a false depending upon whether or not the token is a quote symbol. Either single or double quotes will return a true. This function is used to help identify string literals.

The function IsAdverb returns either a true or a false depending upon whether or not the token is an adverb. Adverbs in COBOL are words that are used to support the verb or command for the sentence. Adverbs are the reserved words that are not verbs found in the procedure division of a COBOL program.

The function IsPunctuation is used to identify the punctuation symbols used in the COBOL program. It returns a true if the token is a punctuation symbol. This function is used to eliminate the punctuation symbols from the COBOL program's other tokens.

The function IsCyclo returns a true if the token adds 1 to the cyclomatic number for the program. The tokens that are considered to be cyclomatic are If, While, When, Depending, and Until. These tokens are also verbs or

adverbs. The IsCyclo function does not uniquely identify the tokens it is used only to determine the number of control flow paths through the program.

The function IsReserved contains all of the reserved words in the COBOL language. It will return a true if it is reserved and a false otherwise. Reserved words include verbs and adverbs but not identifiers (nouns). This function makes writing the code a little simpler because if a token is not reserved it is automatically an identifier without having to identify exactly what type of token it is beforehand.

The function IsNumeric is used to identify numeric literals. It returns a true if the token is a numeric literal and a false otherwise. This function can also be used to identify the level of an identifier in the data division.

The function IsFormat is used to identify the formats used to define an identifier's content. IsFormat knows that a token is a format and needs to be discarded with the prior token is "PIC" or "PICTURE."

The function IsLiteral is used to identify string literals. String literals can be surrounded by either single or double quotes. All string literals are contained in the procedure division.

The function ConNumeric converts a token which is made of a string into a number. The number is then returned and can be used in arithmetic operations.

The function IsId searches the identifier list with the current token to find if it matches any of the elements. If it does it is an identifier and returns a true otherwise it returns a false.

A.3 Procedures

The procedures in this program fit into a structured hierarchy. The main procedure of METRICS opens a file that contains the names of the COBOL programs to be measured, sets up a trace file (which was used in the validation), and then starts the batch run be calling the procedure PROCESS. It gets a COBOL file name, opens that file then calls PROCESS. This process is continued until all of the COBOL programs listed in the name file have been processed.

The procedure Process starts out by opening the COBOL file. Then, it initializes all of the variables, then starts processing the characters from the COBOL program one at a time. It then computes the number of

knots in the program and then writes the computed metrics out to the "METS" file.

The procedure ProcessCharacter is called by the Process procedure and goes through the selected COBOL program's source code one character at a time. It builds the characters that it receives into tokens. The token is complete whenever a end-of-line or end-or-file is read in or a space is encountered.

The procedure DestroyData is called from the Process procedure, and it destroys the data structures that were created when computing the METRICS for an individual COBOL program. DestroyData goes through the data structures one node at a time and reallocates them back to the computers unused memory so that it can be used by the next program. This is necessary to prevent the computer from running out of RAM.

The procedure GetCharacter is called from the Process procedure and it gets the next character from the input stream to be processed. If the file is empty then it communicates this to ProcessCharacter so that it will know that the last token is complete.

The procedure ComputeIdDistance is called toward the end of Process. It computes the IntraMod and InterMod from the data structure that was previously built. It computes the average of the metrics for each identifier in the program.

The procedure ResetThisProcedure goes through the TempIdQueue and resets each identifier to false. This is necessary because distances between uses of each identifier is only computed within modules. This procedure is executed every time a new module heading is found in the input stream by the ProcessToken procedure.

The procedure PushStruc is called from the ProcessToken procedure. PushStruc puts the identifiers in the switch stack. It also puts the module names in the procedure name stack. This procedure is used to create the data structures that hold the tokens found in the input stream.

The procedure PutJumps is called from the Process procedure. It puts a control flow jump in the TempStrucQueue. Whenever a GO TO statement, a module call procedure, or a procedure call statement can cause a control flow jump. This structure can then be used to compute the number of knots.

The procedure InsertMod is used to put a new module name into the TempStrucQueue. It is called whenever a new module name is encountered in the input flow.

The procedure InsertJumps puts a GO TO in the TempStrucQueue. These jumps can be caused by GO TO statements and PERFORM statements. The TempStrucQueue can latter be checked for knots by searching for patterns.

The procedure RidThru is used to put the module calls into the TempStrucQueue whenever a PERFORM THRU statement was found in the source code. A PERFORM THRU in COBOL allows the programmer to call two or more modules with a single statement as long as the modules are concatenated.

The procedure InsertCall creates a node in TempStrucQueue2. It is used whenever a control flow change occurs. This is necessary to compute the knots processing.

The procedure PrintStruc prints out the structure of the program. This procedure is not executed during actual processing. It was put in to help debug and validate the program during the testing period.

The procedure FindKnots searches through the TempStrucStack2 to find the patterns for a knot in the program. FindKnots goes through the structure stack and compares each possible combination of two entries to determine if a knot exists.

The procedure ComputeKnots is called from FindKnots. ComputeKnots looks for the actual pattern between the two combinations from FindKnots to determine when a knot exists. FindKnots is nested inside of the procedure compute knots.

The procedure ComputePrDistance does the computations to compute call span and mod call complexity. These complexity metrics are averages for all of the identifiers in the program. It also prints out some information as the program executes which was used to help validate that the program was working properly.

The procedure PrintProcedure printed out the procedures as they were located in the input stream. PrintProcedure was taken out of this program after testing and debugging and was not executed when computing the metrics.

The procedure PushSwitch is called from the ProcessToken procedure and it puts identifiers on the Switch stack where the order can be reversed so that identifiers that have parents can be processed correctly.

The procedure DestroySwitch is used to destroy the switch stack after each identifier is processed. A switch stack is created for every identifier. After the identifier is moved to the Identifier stack, the switch stack is destroyed.

The PopSwitch procedure gets the most recent item put on the switch stack and returns it to the calling procedure. It reverses the order of an identifier and its parent. The parent will come off first then the identifier. The parent and identifier will then be in the same order as found in the data division of the COBOL program.

The PrintNot procedure is used to print out a list of the tokens not identified but are in the COBOL source code. These are put in a file and printed on the screen. If anything is found in the PrintNot file then there is either an error in the COBOL source code or an error in the METRICS program.

The procedure PushNot is a procedure that pushes identifiers

not found onto the Not stack for latter processing. The VerbNo is stored along with the actual token to make it easier to find out what type of an error has occurred.

The procedure DestroyIdQueue goes through the Id queue one node at a time and releases the node back to the computers unused RAM list. DestroyIdQueue is called by Process after all metrics computations have been made.

The procedure PrintIdQueue prints out on the screen a list of all identifiers along with other metric information. This procedure was used during the debugging and testing phase of program design. This procedure is not used during actual processing.

The procedure ComputeIdSize computes the length of a token in characters. The length has to be less than or equal to 30 characters. If the length of a token is too long the characters over 30 are truncated.

The PushName procedure puts an identifier name onto the identifier stack. The identifier is found whenever a "PIC" or a "PICTURE" is found in the data division.

A.4 Reliability of the METRICS program

The reliability of a program can be accomplished through proofs, black box testing, and white box testing to provide evidence that a program actually works as planned. Proofs are very complex for small programs and become impossible for large programs. Black box testing is effective as long as the test data is similar to the actual data; however, white box testing is the best when the program is large and required to process uncertain data.

White box testing involves examining the contents of the program while executing test data to determine that it works. It was decided that a combination of white box and black box testing would be conducted. The contents of the METRICS program are examined while also comparing the computerized output with manually computed output.

At its most fundamental processing level the METRICS program is a character processor. It reads in characters from a COBOL program's source code and builds tokens (words) which are used to construct the data structures needed to compute the software metrics. The testing was accordingly divided up into stages corresponding to the processing of the METRICS program. The first stage was to test METRICS's ability to divide

up a COBOL program into a sequence of characters. The second stage was to test the METRICS's program's ability to correctly build tokens or words out of the sequence of characters from the first stage. The third stage tests the METRICS program's ability to construct the required data structures. The fourth and last stage was to compare the metrics computed by the METRICS program with manually computed metrics to determine if the program produces the same values.

During the first stage of testing the METRICS program was used to process several COBOL programs. A trace file was created. The METRICS program did create a sequence of characters for each COBOL program tested.

During the second stage a trace file containing tokens (words) was created. Thirty test programs selected from a textbook data disk were used as the input data. The METRICS program was used to develop the trace file. The trace file contained all of the tokens generated from the METRICS program processing the thirty test programs.

During the third stage the METRICS program was modified to print out the contents of the data structures to determine if the tokens were being

used to build the data structures correctly. As each of the data structures discussed earlier in this chapter were coded, each data structure was tested to determine whether or not:

1) the appropriatetokens were put into the data structure
2) the token types were counted as encountered in the input stream of characters, and
3) the token distances from the locations in the program that each token was placed was recorded.

The METRICS program was tested then modified until it created and processed the data structures required to compute the software metrics for the test data.

During the last stage the METRICS program created a data file that contained the computed metrics. Seven complete test programs were written. Each of the programs had different complexity characteristics and each had different values for the complexity metrics. The output metrics from he METRICS program were compared with the manually computed metrics. The METRICS program computed the same metric values as manual computation for the seven test programs.

The reliability testing of the METRICS program was successful. The METRICS program passed the testing described in the previous pages at four

different stages. The testing did not reveal anything that would indicate that the METRICS program is unreliable. The METRICS program processed the data for this study without failing. The results of this study were consistent with the theory which further supports that the METRICS program processed the data successfully.

Appendix B

The Metrics Program Source Code

Following is the Pascal source code for the METRICS program. It is executable on a Pascal compiler.

```
PROGRAM METRICS(INPUT,OUTPUT);

(*****************************************************************)
(* The purpose of this program is to compute metrics that will be *)
(* used to quantify the complexity of a COBOL program.           *)
(*****************************************************************)

VAR trace:text; (* A file output which identifies tokens *)
    values:text;
    dummy:String;
    ProgramToParse:String;

TYPE TokArray=Array[1..30] of char;



PROCEDURE PROCESS(ProgName:String);

Type Parts=(Ident,Environ,Data,Proced);
WordType=(Verb,Adverb,Punctuation,ProcHead,Unknown, SubCall,
        Numeric,Trash,Reserved,AlphaLit,NumLit,Format,Id,Pr);


(* This is a Queue that will hold the identifiers in a program *)

IdPtrType=^IdNodeType;
IdNodeType=Record
  Name:TokArray;          (* The identifer name                    *)
  Level:LongInt;          (* The level number                      *)
  TimesUsed:LongInt;      (* The times the ID is used in program   *)
  LastVerbNo:LongInt;     (* The verb# the last time the id was used *)
  VerbDistance:LongInt;   (* The total distance in verbs between
                             uses of an identifer within a module  *)
```

```
  Procedures:LongInt;         (* The number of modules that an identifier
                              is used in                              *)
  ThisProcedure:Boolean;      (* True when the identifer has been used in
                              this procedure.  *)
  Terminal:Boolean;
  Next:IdPtrType;             (* The address of the next node in the *)
                            (* stack                                *)
end; (* record *)

(* This is a stack that will hold the procedure names *)

ProPtrType=^ProNodeType;
ProNodeType=record
   Name:TokArray;              (* This is the name of the procedure *)
   VerbNo:LongInt;             (* This is the Verb# of the procedure
                               which will be needed when doing knots *)
   TimesUsed:LongInt;
   LastVerbNo:LongInt;
   VerbDistance:LongInt;
   ThisProcedure:    Boolean;
   Procedures   :    LongInt;
   Next:ProPtrType;            (* this is the address of the next address *)

end; (* record *)


(* This is the stack that stores the procedure calls *)
StrPtrType=^StrNodeType;
StrNodeType=record
      VerbNo:LongInt;  (* The verb number that the call is made from       *)
      CallNo:LongInt;  (* The verb number that the call is made to        *)
      CallProc:Array[1..30] of char;  (* The name of the called procedure *)
      Next:StrPtrType;  (* the address of the next node *)
end; (* record *)

(* This procedure is used to switch the procedure names *)

SwtPtrType=^SwtNodeType;
SwtNodeType=record
      ProcName:TokArray; (* The procedure names *)
      Next:SwtPtrType;
end; (* record *)
```

```
(* This Stack holds the tokens not found in the ID stack *)

NotPtrType=^NotNodeType;
NotNodeType=record
        IdName:Array[1..30] of char;
        VerbNo:LongInt;
        Found:Boolean;
        Module:Array[1..30] of char;
        Next:NotPtrType;
end; (* record *)

StrucPtrType=^StructNodeType;
StructNodeType=record
        Kind:Char;
        Par1:Tokarray;
        Par2:Tokarray;
        Next: StrucPtrType;
  end; (* record *)

VAR
 CurProg:Text;          (* The name of the program being processed        *)
 InChar:Char;           (* The current character being processed          *)
 Token:String;          (* The current word being processed               *)
 TokenStart:Integer;    (* The starting position on the card of the token *)
 Position:Integer;      (* The position of the InChar on the card         *)
 Division:Parts;        (* The division that the token is in.             *)
 Token1:String;         (* The prior Token                                *)
 Token2:String;         (* The 2nd prior Token                            *)
 Token3:String;
 TokenStart1:Integer;   (* The starting position of the prior token       *)
 TokenStart2:Integer;   (* The 2nd prior starting position of token       *)
 TokenStart3:Integer;
 TokenType:WordType;    (* The type for the current token                 *)
 TokenType1:WordType;   (* The type for the prior token                   *)
 TokenType2:WordType;   (* The type for the 2nd prior token               *)
 TokenType3:WordType;
 CurModule:String;
 LastModSt:TokArray;

 (* These variables hold the stack pointers *)
 TopIdQueue,TempIdQueue,BotIdQueue:IdPtrType;
 TopProStack,TempProStack:ProPtrType;
```

```
TopPtrStack,TempPtrStack:ProPtrType;
TopSwtStack,TempSwtStack:SwtPtrType;
TopNotStack,TempNotStack:NotPtrType;
FrontStrucQueue,TempStrucQueue,TempStrucQueue1,
BackStrucQueue,TempStrucQueue2:StrucPtrType;



(* Token Status Indicators *)
TokenInProcess:Boolean;  (* The current token contains 1 or more characters *)
CommentInProcess:Boolean;(* A comment line is being read in *)
ContinuationInProcess:Boolean;(* A continuation card is being processed *)
SingleQuoteInProcess:Boolean; (* A literal in single quotes is being processed *)
DoubleQuoteInProcess:Boolean;(* A literal in double quotes is being processed *)
PicFormatInProcess:Boolean;
PerformInProcess:Boolean;

(* Metrics *)
  LinesOfCode:LongInt; (* The number of Lines in a computer program      *)
  CommentsId:LongInt;  (* The number of comments in a computer program   *)
  CommentsEn:LongInt;  (* The number of comments in the environment div  *)
  CommentsDa:LongInt;  (* The number of comments in the data division    *)
  CommentsPr:LongInt;  (* The number of comments in the procedure div    *)
  Verbs:LongInt;       (* The number of Verbs in the procedure division  *)
  Adverbs:LongInt;     (* The number of Adverbs in the Procedure Division*)
  Statements:LongInt;  (* Verbs + Procedure Headings used to count for   *)
                       (* Knots                                          *)
  Punctuations:LongInt;(* The number of punctuation marks in a program   *)
  ProcHeads:LongInt;   (* The number of procedures in a program          *)
  NumChar:REAL;        (* The number of characters in identifiers        *)
  NumUniqueId:LongInt; (* The number of identifiers in environ and data  *)
  NumLits:LongInt;     (* The number of numeric literals in the progran *)
  AlphaLits:LongInt;   (* The number of alpha literals in the program   *)
  Trashs:LongInt;      (* The number of unknown tokens in the program    *)
  LastModNo:LongInt;
  ProcUsed:LongInt;

  Unknowns:LongInt;
  Cyclomatics:LongInt; (* The number of control flow paths in the program *)
  Ids:LongInt;         (* The number of Identifiers in the program       *)
  SubCalls:Longint;    (* The number of subcalls in a program            *)
  ParentsDa:LongInt;   (* The number of parents in the data division *)
  ParentsPr:LongInt;   (* The number of parents in the procedure division *)
```

```
IntraMod:Real;    (* distance *)
InterMod:Real;    (* procedures *)
CallSpan:Real;    (* distance   *)
ModCall:Real;     (* procedures *)
Knots:LongInt;
Gotos:LongInt;
```

```
Function IsVerb(Token:String):Boolean;  (* this is a verb *)
Begin
IsVerb:=False;
If (Token='ACCEPT')     or      (Token='ADD')       or
   (Token='ALTER')      or      (Token='CALL')      or
   (Token='CANCEL')     or      (Token='CLOSE')     or
   (Token='COMPUTE')    or      (Token='CONTINUE')  or
   (Token='DELETE')     or      (Token='DISABLE')   or
   (Token='DISPLAY')    or      (Token='DIVIDE')    or
   (Token='ENABLE')     or      (Token='ERASE')     or
   (Token='EVALUATE')   or      (Token='EXIT')      or
   (Token='GENERATE')   or      (Token='GO')        or
   (Token='GOTO')       or      (Token='IF')        or
   (Token='INITIALIZE') or      (Token='INITIATE')  or
   (Token='INSPECT')    or      (Token='MERGE')     or
   (Token='MOVE')       or      (Token='MULTIPLY')  or
   (Token='OPEN')       or      (Token='PERFORM')   or
   (Token='PURGE')      or      (Token='READ')      or
   (Token='RELEASE')    or      (Token='RETURN')    or
   (Token='RECEIVE')    or      (Token='REWRITE')   or
   (Token='SEARCH')     or      (Token='SEND')      or
   (Token='SET')        or      (Token='SORT')      or
   (Token='START')      or      (Token='STOP')      or
   (Token='STRING')     or      (Token='SUBTRACT')  or
   (Token='SUPPRESS')   or      (Token='TERMINATE') or
   (Token='UNSTRING')   or      (Token='USE')       or
   (Token='WRITE')      or      (Token='REWRITE')   or
   (Token='COPY')       or      (Token='REPLACE')   or
   (Token='CANCEL')     or      (Token='GOBACK')    or
   (Token='EXAMINE')    or      (Token='EXHIBIT')
   then IsVerb:=True;
end; (* end Verb *)
```

```
Function IsQuote(Q:Char):Boolean; (* this is a quote *)
Begin
  If (Q="") or (Q="")
  then IsQuote:=True
  else IsQuote:=False;
end;
```

```
Function IsAdverb(Token:String):boolean;  (* this is an adverb *)
Begin
 IsAdverb:=False;
 If (Token ='Data'       ) or        (Token ='<'          ) or
    (Token ='<='         ) or        (Token ='='          ) or
    (Token ='>'          ) or        (Token ='>='         ) or
    (Token ='ADVANCING'   ) or        (Token ='AFTER'      ) or
    (Token ='ALL'        ) or        (Token ='ALPHABETIC'  ) or
    (Token ='ALPHANUMERIC') or        (Token ='ALPHANUMERIC-EDITED') or
    (Token ='ALSO') or                (Token ='AND' ) or
    (Token ='ANY' ) or                (Token ='ASCENDING') or
    (Token ='AT'  ) or                (Token ='BEFORE'     ) or
    (Token ='BELL'       ) or         (Token ='BOLD'       ) or
    (Token ='BY'         ) or         (Token ='CHARACTERS' ) or
    (Token ='COLLATING'  ) or         (Token ='CONTENT'    ) or
    (Token ='CONVERTING' ) or         (Token ='CORR'       ) or
    (Token ='CORRESPONDING') or       (Token ='COUNT'      ) or
    (Token ='DATA'       ) or         (Token ='DATE'       ) or
    (Token ='DAY'        ) or         (Token ='DAY-OF-WEEK') or
    (Token ='DEBUGGING'  ) or         (Token ='DELIMITED'  ) or
    (Token ='DEPEDING'   ) or         (Token ='DESCENDING' ) or
    (Token ='DOWN'       ) or         (Token ='DUPLICATES' ) or
    (Token ='EGI'        ) or         (Token ='ELSE'       ) or
    (Token ='EMI'        ) or         (Token ='END'        ) or
    (Token ='END-ADD'    ) or         (Token ='END-CALL'   ) or
    (Token ='END-COMPUTE') or         (Token ='END-DELETE' ) or
    (Token ='END-DIVIDE' ) or         (Token ='END-EVALUATE') or
    (Token ='END-IF'     ) or         (Token ='END-MULTIPLY') or
    (Token ='END-OF-PAGE' ) or        (Token ='END-PERFORM' ) or
    (Token ='END-READ'    ) or        (Token ='END-RECEIVE' ) or
    (Token ='END-RETURN'  ) or        (Token ='END-REWRITE' ) or
    (Token ='END-SEARCH'  ) or        (Token ='END-START'   ) or
    (Token ='END-STRING'  ) or        (Token ='END-SUBTRACT') or
    (Token ='END-UNSTRING') or        (Token ='END-WRITE'   ) or
    (Token ='EOP'        ) or         (Token ='EQUAL'      ) or
    (Token ='ERROR'      ) or         (Token ='ESI'        ) or
    (Token ='EXCEPTION'   ) or        (Token ='EXTEND'     ) or
    (Token ='FALSE'      ) or         (Token ='FIRST'      ) or
    (Token ='FOR'        ) or         (Token ='FROM'       ) or
    (Token ='GIVING'     ) or         (Token ='GLOBAL'     ) or
    (Token ='GREATER'    ) or         (Token ='I-O'        ) or
    (Token ='IN'         ) or         (Token ='INITIAL'    ) or
    (Token ='INPUT'      ) or         (Token ='INTO'       ) or
```

```
(Token ='INVALID'     ) or          (Token ='IS'          ) or
(Token ='KEY'         ) or          (Token ='LEADING'     ) or
(Token ='LESS'        ) or          (Token ='LINE'        ) or
(Token ='LINES'       ) or          (Token ='LOCK'        ) or
(Token ='MESSAGE'     ) or          (Token ='MULTIPLY'    ) or
(Token ='NAMED'       ) or          (Token ='NEXT'        ) or
(Token ='NO'          ) or          (Token ='NOT'         ) or
(Token ='NUMERIC'     ) or          (Token ='NUMERIC-EDITED') or
(Token ='OF'          ) or          (Token ='OFF'         ) or
(Token ='ON'          ) or          (Token ='OR'          ) or
(Token ='ORDER'       ) or          (Token ='OTHER'       ) or
(Token ='OUTPUT'      ) or          (Token ='OVERFLOW'    ) or
(Token ='PAGE'        ) or          (Token ='PLUS'        ) or
(Token ='POINTER'     ) or          (Token ='PRINTING'    ) or
(Token ='PROCEDURE'   ) or          (Token ='PROCEDURES'  ) or
(Token ='PROCEED'     ) or          (Token ='REEL'        ) or
(Token ='REFERENCE'   ) or          (Token ='REFERENCES'  ) or
(Token ='REMOVAL'     ) or          (Token ='REPLACING'   ) or
(Token ='REPORTING'   ) or          (Token ='REWIND'      ) or
(Token ='REWRITE'     ) or          (Token ='ROUNDED'     ) or
(Token ='RUN'         ) or          (Token ='SEGMENT'     ) or
(Token ='SENTENCE'    ) or          (Token ='SEQUENCE'    ) or
(Token ='SETNENCE'    ) or          (Token ='SIZE'        ) or
(Token ='SPACE'       ) or          (Token ='STANDARD'    ) or
(Token ='TALLYING'    ) or          (Token ='TERMINAL'    ) or
(Token ='TEST'        ) or          (Token ='THAN'        ) or
(Token ='THROUGH'     ) or          (Token ='THRU'        ) or
(Token ='TIME'        ) or          (Token ='TO'          ) or
(Token ='TOP-OF-PAGE' ) or          (Token ='TRUE'        ) or
(Token ='UNIT'        ) or          (Token ='UNTIL'       ) or
(Token ='UP'          ) or          (Token ='UPON'        ) or
(Token ='USING'       ) or          (Token ='VARYING'     ) or
(Token ='WHEN'        ) or          (Token ='WITH'        ) or
(Token ='ZERO'        ) or          (Token ='ZEROS'       ) or
(Token ='ZEROES'      ) or          (Token = '-'          ) or
(token = '*'          ) or          (token = '**'         ) or
(token = '/'          ) or          (token = '+'          ) or
(token = '='          ) OR          (TOKEN = 'COLUMN')      OR
(TOKEN = 'RECORD')      OR          (TOKEN = 'SCREEN')      OR
(TOKEN = 'SPACES')      OR          (TOKEN = 'DEFAULT')     OR
(TOKEN = 'PROGRAM')     OR          (TOKEN = 'BLINKING')    OR
(TOKEN = 'DIVISION')    OR          (TOKEN = 'REVERSED')    OR
(TOKEN = 'PROTECTED')   OR          (TOKEN = 'CONVERSION')  OR
```

```
    (TOKEN = 'UNDERLINED')  OR          (TOKEN = 'HIGH-VALUES') OR
    (TOKEN = 'RETURN-CODE') OR          (TOKEN = 'CURRENT-DATE')
    Then IsAdverb:=True
    else IsAdverb:=False;
end; (* end adverb *)

Function IsPunctuation(Token:String):Boolean; (* this is a punctuation symbol *)
Begin
  If (Token='.') or
    (Token=',') or
    (Token='(') or
    (Token=')')
  then IsPunctuation:=True
  else IsPunctuation:=False;
end; (* End IsPunctuation *)
```

```
Function IsCyclo(Token:String):Boolean; (* is a cyclomatic token *)

(* This is a cyclomatic token *)

Begin
  If (Token='IF') or
     (Token='WHILE') or
     (Token='WHEN') or
     (Token='DEPENDING') or
     (Token='UNTIL')
  Then IsCyclo:=true
  else IsCyclo:=false;
end;
```

```
Function IsReserved(Token:String):Boolean;

(* These are all the reserved words in COBOL *)
Begin
  IF
(Token='ACCEPT') or         (Token='ACCESS') or
(Token='ADD') or            (Token='ADVANCING') or
(Token='AFTER') or          (Token='ALL') or
(Token='ALPHABETIC') or     (Token='ALSO') or
(Token='ALTER') or          (Token='ALTERNATE') or
(Token='AND') or            (Token='ARE') or
(Token='AREA') or           (Token='AREAS') or
(Token='ASCENDING') or      (Token='ASSIGN') or
(Token='AT') or             (Token='AUTHOR') or
(Token='BEFORE') or         (Token='BLANK') or
(Token='BLOCK') or          (Token='BOTTOM') or
(Token='BY') or             (Token='CALL') or
(Token='CANCEL') or         (Token='CD') or
(Token='CF') or             (Token='CH') or
(Token='CHARACTER') or      (Token='CHARACTERS') or
(Token='CLOCK-UNITS') or    (Token='CLOSE') or
(Token='COBOL') or          (Token='CODE') or
(Token='CODE-SET') or       (Token='COLLATING') or
(Token='COLUMN') or         (Token='COMMA') or
(Token='COMMUNICATION') or  (Token='COMP') or
(Token='COMPUTATIONAL') or  (Token='COMPUTE') or
(Token='CONFIGURATION') or  (Token='CONTAINS') or
(Token='CONTROL') or        (Token='CONTROLS') or
(Token='COPY') or           (Token='CORR') or
(Token='CORESPONDING') or   (Token='COUNT') or
(Token='CURRENCY') or       (Token='DATA') or
(Token='DATE') or           (Token='DATE-COMPILED') or
(Token='DATE-WRITTEN') or   (Token='DAY') or
(Token='DE') or             (Token='DEBUG-CONTENTS') or
(Token='DEBUG-ITEM') or     (Token='DEBUG-LINE') or
(Token='DEBUG-NAME') or     (Token='DEBUG-SUB-1') or
(Token='DEBUG-SUB-2') or    (Token='DEBUG-SUB-3') or
(Token='DEBUGGING') or      (Token='DECIMAL-POINT') or
(Token='DECLARATIVES') or   (Token='DELETE') or
(Token='DELIMITED') or      (Token='DELIMITER') or
(Token='DEPENDING') or      (Token='DESCENDING') or
(Token='DESTINATION') or    (Token='DETAIL') or
(Token='DISABLE') or        (Token='DISPLAY') or
```

```
(Token='DIVIDE') or         (Token='DIVISION') or
(Token='DOWN') or           (Token='DUPLICATES') or
(Token='DYNAMIC') or        (Token='EGI') or
(Token='ELSE') or           (Token='EMI') or
(Token='ENABLE') or         (Token='END') or
(Token='END-OF-PAGE') or    (Token='ENTER') or
(Token='ENVIRONMENT') or    (Token='EOP') or
(Token='EQUAL') or          (Token='ERROR') or
(Token='ESI') or            (Token='EVERY') or
(Token='EXCEPTION') or      (Token='EXIT') or
(Token='EXTEND') or         (Token='FD') or
(Token='FILE') or           (Token='FILE-CONTROL') or
(Token='FILLER') or         (Token='FINAL') or
(Token='FIRST') or          (Token='FOOTING') or
(Token='FOR') or            (Token='FROM') or
(Token='GENERATE') or       (Token='GIVING') or
(Token='GO') or             (Token='GREATER') or
(Token='GROUP') or          (Token='HEADING') or
(Token='HIGH-VALUE') or     (Token='HIGH-VALUES') or
(Token='I-O') or            (Token='I-O-CONTROL') or
(Token='IDENTIFICATION')or  (Token='IF') or
(Token='IN') or             (Token='INDEX') or
(Token='INDEXED') or        (Token='INDICATE') or
(Token='INITIAL') or        (Token='INITIATE') or
(Token='INPUT') or          (Token='INPUT-OUTPUT') or
(Token='INSPECT') or        (Token='INSTALLATION') or
(Token='INTO') or           (Token='INVALID') or
(Token='IS') or             (Token='JUST') or
(Token='JUSTIFIED') or      (Token='KEY') or
(Token='LABEL') or          (Token='LAST') or
(Token='LEADING') or        (Token='LEFT') or
(Token='LENGTH') or         (Token='LESS') or
(Token='LIMIT') or          (Token='LIMITS') or
(Token='LINAGE') or         (Token='LINAGE-COUNTER') or
(Token='LINE') or           (Token='LINE-COUNTER') or
(Token='LINES') or          (Token='LINKAGE') or
(Token='LOCK') or           (Token='LOW-VALUE') or
(Token='LOW-VALUES') or     (Token='MEMORY') or
(Token='MERGE') or          (Token='MESSAGE') or
(Token='MODE') or           (Token='MODULES') or
(Token='MOVE') or           (Token='MULTIPLE') or
(Token='MULTIPLY') or       (Token='NATIVE') or
(Token='NEGATIVE') or       (Token='NEXT') or
```

```
(Token='NO') or            (Token='NOT') or
(Token='NUMBER') or        (Token='NUMERIC') or
(Token='OBJECT-COMPUTER') or (Token='OCCURS') or
(Token='OF') or            (Token='OFF') or
(Token='OMITTED') or       (Token='ON') or
(Token='OPEN') or          (Token='OPTIONAL') or
(Token='OR') or            (Token='ORGANIZATION') or
(Token='OUTPUT') or        (Token='OVERFLOW') or
(Token='PAGE') or          (Token='PAGE-COUNTER') or
(Token='PERFORM') or       (Token='PF') or
(Token='PH') or            (Token='PIC') or
(Token='PICTURE') or       (Token='PLUS') or
(Token='POINTER') or       (Token='POSITION') or
(Token='POSITIVE') or      (Token='PRINTING') or
(Token='PROCEDURE') or     (Token='PROCEDURES') or
(Token='PROCEED') or       (Token='PROGRAM') or
(Token='PROGRAM-ID') or    (Token='QUEUE') or
(Token='QUOTE') or         (Token='QUOTES') or
(Token='RANDOM') or        (Token='RD') or
(Token='READ') or          (Token='RECEIVE') or
(Token='RECORD') or        (Token='RECORDS') or
(Token='REDEFINES') or     (Token='REEL') or
(Token='REFERENCES') or    (Token='RELATIVE') or
(Token='RELEASE') or       (Token='REMAINDER') or
(Token='REMOVAL') or       (Token='RENAMES') or
(Token='REPLACING') or     (Token='REPORT') or
(Token='RREPORTING') or    (Token='REPORTS') or
(Token='RERUN') or         (Token='RESERVE') or
(Token='RESET') or         (Token='RETURN') or
(Token='REVERSED') or      (Token='REWIND') or
(Token='REWRITE') or       (Token='RF') or
(Token='RH') or            (Token='RIGHT') or
(Token='ROUNDED') or       (Token='RUN') or
(Token='SAME') or          (Token='SD') or
(Token='SEARCH') or        (Token='SECTION') or
(Token='SECURITY') or      (Token='SEGMENT') or
(Token='SEGMENT-LIMIT') or (Token='SELECT') or
(Token='SEND') or          (Token='SENTENCE') or
(Token='SEPARATE') or      (Token='SEQUENCE') or
(Token='SEQUENTIAL') or    (Token='SET') or
(Token='SIGN') or          (Token='SIZE') or
(Token='SORT') or          (Token='SORT-MERGE') or
(Token='SOURCE') or        (Token='SOURCE-COMPUTER') or
```

```
(Token='SPACE') or          (Token='SPACES') or
(Token='SPECIAL-NAMES') or  (Token='STANDARD') or
(Token='STANDARD-1') or     (Token='START') or
(Token='STATUS') or         (Token='STOP') or
(Token='STRING') or         (Token='SUB-QUEUE-1') or
(Token='SUB-QUEUE-2') or    (Token='SUB-QUEUE-3') or
(Token='SUBTRACT') or       (Token='SUM') or
(Token='SUPPRESS') or       (Token='SYMBOLIC') or
(Token='SYNC') or           (Token='SYNCHRONIZED') or
(Token='TABLE') or          (Token='TALLYING') or
(Token='TAPE') or           (Token='TERMINAL') or
(Token='TERMINATE') or      (Token='TEXT') or
(Token='THAN') or           (Token='THROUGH') or
(Token='THRU') or    (Token='TIME') or
(Token='TIMES') or    (Token='TO') or
(Token='TOP') or    (Token='TRAILING') or
(Token='TYPE') or    (Token='UNIT') or
(Token='UNSTRING') or    (Token='UNTIL') or
(Token='UP') or    (Token='UPON') or
(Token='USAGE') or    (Token='USE') or
(Token='USING') or    (Token='VALUE') or
(Token='VALUES') or    (Token='VARYING') or
(Token='WHEN') or    (Token='WITH') or
(Token='WORDS') or    (Token='WORKING-STORAGE') or
(Token='WRITE') or    (Token='ZERO') or
(Token='ZEROES') or    (Token='ZEROS') or
(Token='ALPHABET') or    (Token='ALPHABETIC-LOWER') or
(Token='ALPHABETIC-UPPER') or    (Token='ALPHANUMERIC') or
(Token='ALPHANUMERIC-EDITIED') or    (Token='BINARY') or
(Token='CLASS') or    (Token='CONTENT') or
(Token='CONTINUE') or    (Token='CONVERTING') or
(Token='DAY-OF-WEEK') or    (Token='FALSE') or
(Token='GLOBAL') or    (Token='INITIALIZE') or
(Token='NUMERIC-EDITED') or    (Token='ORDER') or
(Token='OTHER') or    (Token='PACKED-DECIMAL') or
(Token='PADDING') or    (Token='PURGE') or
(Token='REFERENCES') or    (Token='REPLACE') or
(Token='STANDARD-2TEST') or    (Token='THEN') or
(Token='TRUE')
then IsReserved:=true
else IsReserved:=false;
end;
```

```
Function IsNumeric(Token:String):Boolean;

(* this is a numeric token *)
VAR i:integer;
    periods:integer;
    CanNumeric:Boolean;
Begin
   CanNumeric:=True;

   (* Check and remove a leading + or - symbol *)
   If ((Token[1]='+') or (Token[1] = '-')) and (Length(Token) > 1)
      then Token[1]:='0';

   (* Count the number of periods *)
   periods:=0;
   For i:=1 to Length(token) do
     If (Token[i]='.') then
        periods:=periods+1;

   (* a number can have only one period *)
   If periods > 1 then CanNumeric:=False;

   For i:=1 to Length(token) do
     If CanNumeric then
        If (Token[i]<> '1') and
           (Token[i]<> '2') and
           (Token[i]<> '3') and
           (Token[i]<> '4') and
           (Token[i]<> '5') and
           (Token[i]<> '6') and
           (Token[i]<> '7') and
           (Token[i]<> '8') and
           (Token[i]<> '9') and
           (Token[i]<> '0') and
           (Token[i]<> '.')
         then CanNumeric:=False;

   IsNumeric:=CanNumeric;
end; (* IsNumeric *)
```

```
Function IsFormat(Token:String):Boolean;

(* This is a Format statement*)
Begin
  If (Token1='PIC') or (Token1='PICTURE')
     then IsFormat:=true
     else IsFormat:=false;
end;
```

```
Function IsLiteral(Token:String):Boolean;
(* This is a token within quotes *)

Begin
  If ((Token[1] = '"') and (Token[Length(Token)]='"'))
                    or
     ((Token[1] = '''')  and  (Token[Length(Token)]=''''))
     then IsLiteral:=True
     else IsLiteral:=False;
end; (* end IsLiteral *)
```

```
    Procedure ProcSwtStack;
    (* Prints out the contents of the switch stack *)

    VAR dummy:integer;
       Temp1:SwtPtrType;

    Begin
        Repeat begin
               TempSwtStack:=TopSwtStack;

(*                 Writeln('++',TempSwtStack^.ProcName);*)
               Temp1:=TempSwtStack;
               TempSwtStack:=TempSwtStack^.Next;
               Release(TempSwtStack);
             end until (TempSwtStack=NIL);
        Readln(Dummy);
    end; (* end SwtStack *)
```

```
Function ConNumeric(Tok:String):LongInt;
VAR z      :integer;
    Numeric:LongInt;
 Begin
 Numeric:=0;
 For z:=1 to Length(Tok) do
    Case Tok[z] of
      '0':Numeric:=10*Numeric;
      '1':Numeric:=10*Numeric+1;
      '2':Numeric:=10*Numeric+2;
      '3':Numeric:=10*Numeric+3;
      '4':Numeric:=10*Numeric+4;
      '5':Numeric:=10*Numeric+5;
      '6':Numeric:=10*Numeric+6;
      '7':Numeric:=10*Numeric+7;
      '8':Numeric:=10*Numeric+8;
      '9':Numeric:=10*Numeric+9;
     end;
   ConNumeric:=Numeric;
End; (* end of function ConNumeric *)


Procedure PushName(Token:String);
(* This procedure puts a name into the identifier stack *)

VAR i:integer;
Begin
  New(TempIdQueue);

  (* Put the name into the Id data strucuure *)
  for i:=1 to Length(Token) do
      TempIdQueue^.Name[i]:=Token[i];

  (* Clear out rest of Token *)
  If Length(token) < 30
     then for i:=Length(Token)+1 to 30 do
              TempIdQueue^.Name[i]:=' ';

  TempIdQueue^.TimesUsed:=0;
  TempIdQueue^.LastVerbNo:=0;
  TempIdQueue^.VerbDistance:=0;
  TempIdQueue^.Level:=ConNumeric(Token2);
  TempIdQueue^.Procedures:=0;
```

```
  TempIdQueue^.Terminal:=False;
  TempIdQueue^.ThisProcedure:=False;

  (* Set the top pointer to the stack *)
  If TopIdQueue=NIL then TopIdQueue:=TempIdQueue;
  If BotIdQueue=NIL then BotIdQueue:=TempIdqueue
                  else BotIdQueue^.Next:=TempIdQueue;
  TempIdQueue^.Next:=NIL;
  BotIdQueue:=TempIdQueue;
end;
```

```
Procedure ComputeIdSize;

VAR Count,Spaces:LongInt;

    Z:INTEGER;
Begin
   Count:=0;
   Spaces:=0;

   TempIdQueue:=TopIdQueue;
   While TempIdQueue<>NIL do
    Begin
(*        Writeln('Id>>',TempIdQueue^.Name); *)
     z:=30;
     While TempIdQueue^.Name[z] = ' ' do
          z:=z-1;
     Spaces:=Spaces+z;
(*        Writeln(z);*)
(*        Writeln(spaces);*)
     readln;
     Count:=Count+1;
     TempIdQueue:=TempIdQueue^.Next;
    end;
    TempProStack:=TopProStack;
    While TempProStack<>NIL do
      Begin
(*           Writeln('Pr>>',TempProStack^.Name);*)
        z:=30;
        While TempProStack^.Name[z]=' ' do
            z:=z-1;
        Spaces:=Spaces+z;
(*           Writeln(z);*)
(*           Writeln(spaces);*)
        readln;
        Count:=Count+1;
        TempProStack:=TempProStack^.Next;
      end;
    If Count> 0 then NumChar:=(Spaces/Count)
                else NumChar:=0;

 end;
```

```
    Procedure PrintIdQueue;
    (* This Prints Out the contents of the identifier queue *)
    Begin
      TempIdQueue:=TopIdQueue;
      While TempIdQueue <> Nil do
          begin

            (* Print out the name of the stack                          *)
            (* Print out the number of times the Identifier was used *)
       (*      write(TempIdQueue^.Name,' ',TempIdQueue^.Level:10,' ');*)

            (* Print out the identifier distance *)
        (*     Write(TempIdQueue^.TimesUsed:5,' ');
(*             Write(TempIdQueue^.Procedures:5,' ');*)
(*             Write(TempIdQueue^.LastVerbNo:5,' ');*)
(*             If (TempIdQueue^.Terminal) then write('p ')
                                  else write('d ');*)
(*             If TempIdQueue^.TimesUsed > 0 then
                Writeln(TempIdQueue^.VerbDistance/TempIdQueue^.TimesUsed:10)
               else Writeln('Infinite');         *)
            TempIdQueue:=TempIdQueue^.Next;
          end;
    end; (* end of Printstack *)
```

```
Procedure DestroyIdQueue;
(* This procedure destroys the identifier stack *)
Begin
  While TopIdQueue <> Nil do
    begin
       TempIdQueue:=TopIdQueue;
       TopIdQueue:=TopIdQueue^.Next;
       Dispose(TempIdQueue);
    end;
end; (* destroystack *)
```

```
Procedure PushNot(Tok:TokArray; VerbNo:LongInt);
VAR I:Integer;
Begin
  New(TempNotStack);
  TempNotStack^.Next:=TopNotStack;
  for I:=1 to 30 do
    Begin
      TempNotStack^.IdName[I]:=Tok[I];
      If Length(CurModule)>= I
         then TempNotStack^.Module[I]:=CurModule[I]
         else TempNotStack^.Module[I]:=' ';

    end;
  TempNotStack^.VerbNo:=VerbNo;
  TempNotStack^.found:=false;
  TopNotStack:=TempNotStack;
end; (* PushNot *)
```

```
   Procedure PrintNot;
   Begin
(*     Writeln('>>>>>>> Not found list '); *)
    TempNotStack:=TopNotStack;
    While TempNotStack<> NIL do
      begin
(*        If not (TempNotStack^.Found) then Writeln(TempNotStack^.IdName);
(*        If (TempNotStack^.Found) then (* write('PRO')   *)
(*                                 else write('ERR');  *)
        TempNotStack:=TempNotStack^.Next;
      end;
    end;
```

```
  Procedure SearchNot;

  VAR TopModule:Array[1..30] of char;
      i:integer;
  Begin
  For i:=1 to 30 do TopModule[I]:=' ';
  TempNotStack:=TopNotStack;
  TempProStack:=TopProStack;
  While(TempProStack<>NIL)(* and Not(TempNotStack^.Found) *) do

    Begin
      TempNotStack:=TopNotStack;
      While TempNotStack<>NIL do
       begin
           If TempProStack^.Name=TempNotStack^.IdName
              then Begin
                     TempNotStack^.Found:=true;
                     (* Mark find *)

                     TempProStack^.TimesUsed:=TempProStack^.TimesUsed+1;
                     If (TempProStack^.LastVerbNo=0) or
                        (TopModule <> TempNotStack^.Module)
                        then Begin
                               (* new module *)
(*                                Writeln('---NewModule');
                               Writeln(TempNotStack^.Module);
                               Writeln(TempNotStack^.VerbNo); *)
                               Readln;
                               TempProStack^.Procedures:=
                               TempProStack^.Procedures + 1;
                               For i:=1 to 30 do
                                 TopModule[I]:=TempNotStack^.Module[I];
                               (* error must save verbs *)
                               TempProStack^.LastVerbNo:=TempNotStack^.VerbNo;
                             end
                        else Begin
(*                                Writeln('---Prior Module');
                              (* name used more than once within module *)
(*                               Writeln('Last',TempProStack^.LastVerbNo);
                              Writeln('Curr',TempNotStack^.VerbNo);
                              Writeln('Dist',TempProStack^.VerbDistance);
                              Writeln(TempProStack^.Name); *)
                              Readln;
```

```
                TempProStack^.VerbDistance:=TempProStack^.VerbDistance +
                   TempProStack^.LastVerbNo-TempNotStack^.VerbNo;
                TempProStack^.LastVerbNo:=TempNotStack^.VerbNo;
               end
          end
        else TempNotStack^.Found:=false;
       TempNotStack:=TempNotStack^.Next;
     end;
     TempProStack:=TempProStack^.Next;
   end;
 end; (* Search Not *)
```

```
Procedure PopSwitch(Top:SwtPtrType;TempIdQueue:IdPtrType;Level:Integer);

VAR TerminalFind:Boolean;

Begin
(*    writeln('+++'); *)
 While (TempIdQueue <> NIL) and (TempIdQueue^.Name<>Top^.ProcName) do
      TempIdQueue:=TempIdQueue^.Next;
   If TempIdQueue <> NIL then
     begin
       If Top^.Next=NIL
          then   Begin
          (*         Writeln('Terminal find ',Level,' ',Top^.ProcName);*)
                 (* Mark find *)
                 TempIdQueue^.Terminal:=false;
                 TempIdQueue^.TimesUsed:=TempIdQueue^.TimesUsed+1;
                 If TempIdQueue^.ThisProcedure
                    then Begin
                         TempIdQueue^.VerbDistance:=TempIdQueue^.VerbDistance
                                                +Verbs
                                                -TempIdQueue^.LastVerbNo;
                         TempIdQueue^.LastVerbNo:=Verbs;
                        end;
                 If not TempIdQueue^.ThisProcedure
                    then Begin
                          TempIdQueue^.Procedures:=TempIdQueue^.Procedures
                                                 + 1;
                          TempIdQueue^.ThisProcedure:=True;
                        end
               end
          else begin
         (*       Writeln('         find ',Level,' ',Top^.ProcName);*)
               PopSwitch(Top^.Next,TempIdQueue,Level+1);
             end;
     end
     else
        Begin
    (*       writeln('not found ',level,' ',Top^.ProcName); *)
          PushNot(Top^.ProcName,Verbs);
        end;
 end; (* PopSwitch *)
```

```
Procedure DestroySwitch(Top:SwtPtrType);

Begin
  While Top <> NIL do
   Begin
      TempSwtStack:=Top^.Next;
      Dispose(Top);
      Top:=TempSwtStack
   end;
   TopSwtStack:=NIL;
end; (* Destroy Switch *)
```

```
   Procedure PushSwitch(Token:String);

   (* This procedure pushes a procedure name that will be reversed *)
   VAR Z:Integer;
   Begin
     New(TempSwtStack);
(*      Writeln('P++ ',Token); *)
     Z:=1;
     While (Z<=30) do
        Begin
         If z<=Length(Token) then TempSwtStack^.ProcName[z]:=Token[z]
                          else TempSwtStack^.ProcName[z]:=' ';
         Z:=Z+1;
        end;
     TempSwtStack^.Next:=TopSwtStack;
     TopSwtStack:=TempSwtStack;
   End; (* PushSwitch *)
```

```
Procedure PushProcedure(Token:String);

VAR i:Integer;

Begin
  New(TempProStack);
  For i:=1 to 30 do
     TempProStack^.Name[i]:=' ';
  For i:=1 to Length(Token) do
     TempProStack^.Name[i]:=Token[i];
  TempProStack^.Next:=TopProStack;
  TempProStack^.TimesUsed:=0;
  TempProStack^.Procedures:=0;
  TempProStack^.VerbDistance:=0;
  TempProStack^.LastVerbNo:=0;
  TopProStack:=TempProStack;
End; (* end PushProcedure *)
```

```
Procedure PrintProcedure;

Begin
(*  Writeln('>>>Procedures');*)
  While TopProStack<>NIL do
   Begin
 (*    Write(TopProStack^.Name);*)
 (*    Writeln(' ',TopProStack^.TimesUsed); *)
     TempProStack:=TopProStack;
     TopProStack:=TopProStack^.Next;
     Dispose(TempProStack);
   end;
(*   Writeln('');*)
end;
```

```
    Procedure ComputePrDistance;
    VAR totalD,TotalP,count:LongInt;
    Begin
      TempProStack:=TopProStack;
      TotalP:=0;
      TotalD:=0;
      Count:=0;
      While TempProStack <> NIL do
        Begin
          If TempProStack^.TimesUsed > 0 then
           Begin
             TotalD:=TotalD+TempProStack^.VerbDistance;
             Count:=Count+1;
             TotalP:=TotalP+TempProStack^.Procedures;
           end;
(*             Writeln(TempProStack^.VerbDistance);
           Writeln(TempProStack^.Procedures);
           Writeln(TempProStack^.TimesUsed);
           Writeln(TempProStack^.Name);
*)
           TempProStack:=TempProStack^.Next;
           REadln;
       end;
      TempProStack:=TempProStack^.Next;
      If Count=0
           then begin
                  CAllSpan:=0;
                  ModCall:=0;
                end
           else Begin
                  CallSpan:=TotalD/Count;
                  ModCall:=TotalP/Count;
                end;

    end;
```

```
Procedure ComputeKnots(Parmeter1,Parmeter2:StrucPtrType);

(* This procedure searches through the FrontStrucQueue to *)
(* find the ordering of every pair of GOTO's in the Queue *)
(* the ordering can be used to determine if a Knot exists *)

VAR  Par1G:Integer;
     Par1M:Integer;
     Par2G:Integer;
     Par2M:Integer;
     NumFound:Integer;

Begin
   Par1g:=0;
   Par1m:=0;
   Par2g:=0;
   Par2m:=0;
   NumFound:=0;
   TempStrucQueue2:=FrontStrucQueue;
   While TempStrucQueue2 <> NIL do
      Begin
        If TempStrucQueue2^.Kind = 'G'
          then Begin
                If TempStrucQueue2 = Parmeter1
                  then begin
                         NumFound:=NumFound+1;
                         Par1G:=NumFound;
                       end; (* if *)
                If TempStrucQueue2 = Parmeter2
                  then begin
                         NumFound:=NumFound + 1;
                         Par2G:=NumFound;
                       end; (* if *)
              end; (* IF *)

        If TempStrucQueue2^.Kind = 'M'
          then Begin
                If TempStrucQueue2^.Par1 = Parmeter1^.Par1
                  then begin
                         NumFound:=NumFound + 1;
                         Par1M:=NumFound;
                       end; (* if *)
                If TempStrucQueue2^.Par1 = Parmeter2^.Par1
```

```
                then begin
                   NumFound:=NumFound + 1;
                   Par2M:=Numfound;
                  end; (* if *)

              end; (* IF *)
          TempStrucQueue2:=TempStrucQueue2^.Next;
         end; (* While *)

(*          Writeln(Parmeter1^.Par1,' ',Parmeter2^.Par1,' ',Par1G,Par1M,Par2G,Par2M); *)

          If ((Par1G=3) and
              (Par2G=4) and
              (Par1M=1) and
              (Par2M=2)  )
               or
             ((Par1G=1) and
              (Par2G=2) and
              (Par1M=3) and
              (Par2M=4)  )
               or
             ((Par1G=3) and
              (Par2G=2) and
              (Par1M=1) and
              (Par2M=4)  )
               or
             ((Par1G=2) and
              (Par2G=3) and
              (Par1M=4) and
              (Par2M=1)  )
               or
             ((Par1G=1) and
              (Par2G=4) and
              (Par1M=3) and
              (Par2M=2)  )
               or
             ((Par1G=4) and
              (Par2G=1) and
              (Par1M=2) and
              (Par2M=3)  )

             then Knots:=Knots+1;
   end; (* Compute Knots *)
```

```
  Procedure FindKnots;

   Begin
     TempStrucQueue:=FrontStrucQueue;
     While TempStrucQueue^.Next <> NIL do
       Begin

(*          Writeln(TempStrucQueue^.Kind);
         Writeln(TempStrucQueue^.Par1);
         Writeln(TempStrucQueue^.Par2); *)
         Readln;
         If TempStrucQueue^.Kind='G'
           then begin
                  TempStrucQueue1:=TempStrucQueue;
                  While TempStrucQueue1^.Next <> NIL do
                    Begin
                      If (TempStrucQueue1^.Kind='G') and
                        (TempStrucQueue1^.Par1 <> TempStrucQueue^.Par1)
                        then ComputeKnots(TempStrucQueue,
                                          TempStrucQueue1);
                      TempStrucQueue1:= TempStrucQueue1^.Next;
                    end;
                end;
         TempStrucQueue:=TempStrucQueue^.Next;
       end; (* While *)
   end; (* findknots *)
```

```
Procedure PrintStruc;

(* This procedure prints out the contents of the Structure Queue *)

Begin
  TempStrucQueue:=FrontStrucQueue;
  While TempStrucQueue <> NIL do
    Begin
(*        Write(TempStrucQueue^.Kind);
      Write('|');
      Write(TempStrucQueue^.Par1);
      Write('|');
      Writeln(TempStrucQueue^.Par2); *)
      TempSTrucQueue:=TempStrucQueue^.Next;

    end; (* While *)
    Readln;

end;
```

```
Procedure InsertCall(Parmeter1:TokArray);
Begin
  New(TempStrucQueue2);
  TempStrucQueue2^.Kind:='P';
  TempStrucQueue2^.Par1:=Parmeter1;
  TempStrucQueue2^.Par2:='                              ';
  TempStrucQueue2^.Next:=TempStrucQueue^.Next;
  TempStrucQueue^.Next:=TempSTrucQueue2;
end;
```

```
   Procedure RidThru;
   Begin
     TempStrucQueue:=FrontStrucQueue;
     While TempStrucQueue<>NIL do
       Begin
        If TempStrucQueue^.Kind='T'
          then Begin
                TempStrucQueue1:=TempStrucQueue;
                While (TempStrucQueue1^.Par1 <> TempStrucQueue^.Par1)
                                 and
                    (TempStrucQueue1^.Kind ='M')
                    do TempStrucQueue1:=TempStrucQueue1^.Next;
(*                Writeln('>>',TempSTrucQueue1^.Par1); *)
                TempStrucQueue1:=TempStrucQueue;
                While (TempStrucQueue1^.Par1 <> TempStrucQueue^.Par2)
                                  or
                    (TempStrucQueue1^.Kind <> 'M') do
                  Begin
                   If TempSTrucQueue1^.Kind='M'
                     then InsertCall(TempStrucQueue1^.Par1);
                   TempSTrucQueue1:=TempStrucQueue1^.Next;
                  end; (* while *)
                InsertCall(TempStrucQueue1^.Par1);
(*                Writeln('Thru found'); *)
              end;
        TempStrucQueue:=TempStrucQueue^.Next;
       end;

   End;
```

```
Procedure InsertJump(Parmeter1:Tokarray);
VAr I:Integer;
Begin;
   New(TempStrucQueue2);
(*     Writeln('**',Parmeter1); *)
   TempStrucQueue2^.Kind:='G';
   TempStrucQueue2^.Par1:=Parmeter1;
   For I:=1 to 30 do
     TempStrucQueue2^.Par2[i]:=' ';
   TempStrucQueue2^.Next:=TempStrucQueue1^.Next;
   TempStrucQueue1^.Next:=TempStrucQueue2;
end;
```

```
 Procedure InsertMod(Parmeter1:Tokarray);
 VAR I:integer;
 Begin
    New(TempStrucQueue2);
(*     Writeln('**',Parmeter1); *)
    TempStrucQueue2^.Kind:='M';
    TempStrucQueue2^.Par1:=Parmeter1;
    For I:=1 to 30 do
      TempStrucQueue2^.Par2[i]:=' ';
    TempStrucQueue2^.Next:=TempStrucQueue1^.Next;
    TempStrucQueue1^.Next:=TempStrucQueue2;

 end;
```

```
    Procedure PutJumps;
    VAR i:Integer;
       NumStr:String;
    Begin
      TempStrucQueue:=FrontStrucQueue;
      While TempStrucQueue^.Next <> NIL do
        Begin
          If TempStrucQueue^.Kind = 'P'
            then begin
                   TempStrucQueue1:=TempStrucQueue;
                   InsertJump(TempStrucQueue^.Par1);
                   LastModNo:=LastModNo+1;
                   Str(LastModNo,NumStr);
                   For I:=1 to 30 do
                    If I <= Length(NumStr)
                       then LastModSt[I]:=NumStr[I]
                       else LastModSt[I]:=' ';
                   InsertMod(LastModSt);
                   TempStrucQueue1:=FrontStrucQueue;
                   While (TempStrucQueue1^.Par1 <> TempStrucQueue^.Par1)
                                 or
                         (TempStrucQueue1^.Kind <> 'M')
                         Do TempStrucQueue1:=TempstrucQueue1^.Next;
(*                      Writeln(LastModSt); *)

                   InsertJump(LastModSt);
                 end; (* If *)
          TempStrucQueue:=TempStrucQueue^.Next;
        end; (* while *)

    end;
```

```
Procedure PushStruc(Par1:Char;Par2:String;Par3:String);

(* The purpose of this procedure is to build the structure stack *)

VAR I:Integer;

Begin
(*    Writeln(Par1,Par2,Par3);*)
  New(TempStrucQueue);
  TempStrucQueue^.Kind:=Par1;
  For I:=1 to 30 do
    begin
      TempStrucQueue^.Par1[I]:=' ';
      TempStrucQueue^.Par2[I]:=' ';
    end;
  For I:=1 to Length(Par2) do
      TempStrucQueue^.Par1[I]:=Par2[I];
  For I:=1 to Length(Par3) do
      TempStrucQueue^.Par2[I]:=Par3[I];
  If FrontStrucQueue = NIL
    then  FrontStrucQueue:=TempStrucQueue
     else
         BackStrucQueue^.Next:=TempSTrucQueue;

  BackStrucQueue:=TempStrucQueue;
  TempStrucQueue^.Next:=NIL;

end; (* PushStruc *)
```

```
Procedure ComputeIdDistance;

VAR totald,totalP,count:Longint;

Begin
  TempIdQueue:=TopIdQueue;
  TotalP:=0;
  TotalD:=0;
  Count:=0;
  While TempIdQueue<> NIL do
    Begin
(*      Writeln('ID>>',TempIdQueue^.Name);*)
(*      Writeln(TempIdQueue^.VerbDistance);
      Writeln(TempIdQueue^.Procedures); *)
      If TempIdQueue^.TimesUsed > 1
      then begin
             TotalD:=TotalD+TempIdQueue^.VerbDistance;
             Count:=Count+1;
             TotalP:=TotalP+TempIdQueue^.Procedures;
           end;
      TempIdQueue:=TempIdQueue^.Next;
    end;
  If Count>0 then InterMod:=TotalP/Count
             else InterMod:=0;
  If Count>0 then IntraMod:=TotalD/Count
             else IntraMod:=0;
end;
```

```
Procedure GetCharacter;
(* This procedure will get a character from the current program *)
(* and if it is lower case convert it to upper case and update  *)
(* the position counter on the card. *)

Begin
  Read(CurProg,InChar);

  (* Convert to Upper Case *)
  Case InChar of
  'a':InChar:='A';
  'b':InChar:='B';
  'c':InChar:='C';
  'd':InChar:='D';
  'e':InChar:='E';
  'f':InChar:='F';
  'g':InChar:='G';
  'h':InChar:='H';
  'i':InChar:='I';
  'j':InChar:='J';
  'k':InChar:='K';
  'l':InChar:='L';
  'm':InChar:='M';
  'n':InChar:='N';
  'o':InChar:='O';
  'p':InChar:='P';
  'q':InChar:='Q';
  'r':InChar:='R';
  's':InChar:='S';
  't':InChar:='T';
  'u':InChar:='U';
  'v':InChar:='V';
  'w':InChar:='W';
  'x':InChar:='X';
  'y':InChar:='Y';
  'z':Inchar:='Z';
 end; (* case *)
Position:=Position+1;
End;  (* GetCharacter *)
```

```
Function CheckId (Token:String):Boolean;
VAR Tok:TokArray;
    i:integer;
Begin
    CheckId:=false;
    for i:=1 to 30 do
      tok[i]:=' ';
    for i:=1 to 30 do
       If i<=length(Token)
       then Tok[i]:=Token[i];
    TempIdQueue:=TopIdQueue;
    While (TempIdQueue <> NIL) do
      Begin
        If TempIdQueue^.Name = Tok
           then CheckId:=true;

        TempIdQueue:=TempIdQueue^.Next;
      end;

end;
```

```
Procedure ProcessCharacter;
(* This procedure takes the characters and builds them into tokens *)
(* or words to be processed latter    *)


  Procedure ProcessToken;
  (* This Procedure processes each token *)

  Var z:STRING;
     ds:string; (* Division Type String *)
     ts:string; (* Token Type String    *)


  Begin
    (* Determine if the Division has changed *)

    If (Token='.') and
      (Token1='DIVISION')
      then If (Token2='IDENTIFICATION') then Division :=Ident
          else
          If (Token2='ENVIRONMENT') then Division :=Environ
          else
          If (Token2='DATA') then Division := Data
          else
          If (Token2='PROCEDURE') then Division :=Proced;

    (* Put current division in ds *)
    ds:='Un';
    CASE Division of
        Ident:    ds:='Id';
        Environ:  ds:='En';
        Data:     ds:='Da';
        Proced:   ds:='Pr';
      end;

    (* Determine the token type *)
    TokenType:=Unknown;
    If Division=Proced then
      begin
        If Token = 'PERFORM' then PerformInProcess:=True;

        (* Determine the Token Type *)
        If IsVerb(Token) then TokenType:=Verb
```

```
            else If IsAdverb(Token) then TokenType:=Adverb
            else If IsPunctuation(Token) then TokenType:=Punctuation
            else If IsLiteral(Token) then TokenType:=AlphaLit
            else If IsNumeric(Token) then TokenType:=NumLit
            else If TokenStart >=12    (* an identifier *)
                  then Begin
(*                           writeln('Id ',Token); *)
                     If CheckId(Token)
                       then TokenType:=Id
                       else TokenType:=Pr;
                      PushSwitch(Token);
                     end
                  else Begin        (* a procedure *)
 (*                          writeln('Pr ',Token); *)
                       TokenType:=ProcHead;
                       PushProcedure(Token);
                     end;
            If (TopSwtStack<>NIL) and (Token<>'IN') and (TokenType<> ID)
              Then Begin
                   PopSwitch(TopSwtStack,TopIdQueue,1);
                   DestroySwitch(TopSwtStack);
                  end;

        end
        else  (* Check for data division *)

           If (Division = Data) then
           begin (* there's an error here because it doesn't *)
                (* fit the data structure *)

            If IsNumeric(Token) then TokenType:=NumLit;

            (* Check to see if it begins with FD *)
            (* Starts with a number and ends with a period *)
            (* or Picture or Value                        *)

            If IsNumeric(Token2) and ((Token='.') or
              (Token='VALUE') or (Token='PIC') or (Token='PICTURE'))
              or (Token2='FD')
              then
                   Begin          (* Its an identifier *)
                    PushName(Token1);
                    TokenType1:=Id;
```

```
            end;
   end;

 (* Build Type String *)
 Case TokenType of
  VERB:        ts:='Vb';
  ADVERB:      ts:='Av';
  Punctuation: ts:='Pu';
  Unknown:     ts:='Un';
  Numeric:     ts:='Nm';
  Trash  :     ts:='Tr';
  ProcHead:    ts:='Pr';
  Reserved:    ts:='Rs';
  NumLit:      ts:='NL';
  AlphaLit:    ts:='AL';
  NumLit :     ts:='NL';
  Format:      ts:='Fm';
  Id    :      ts:='Id';
  SubCall:     ts:='Sc';
  Pr:          ts:='PU';
  end; (* case *)

 If Division = Proced then
 begin
  (* Update Procedure counts *)
  If TokenType = Verb        then Begin
                                  Verbs:=Verbs+1;
                                  Statements:=Statements+1;
                                 end;
  If TokenType = Adverb      then Adverbs:=Adverbs+1;
  If TokenType = Punctuation then Punctuations:=Punctuations+1;
  If TokenType = ProcHead    then Begin
                                  ProcHeads:=ProcHeads+1;
                                  Statements:=Statements+1;
                                  CurModule:=token;
                                 end;
  If TokenType = Trash       then Trashs:=Trashs+1;
  If TokenType = Unknown     then Unknowns:=Unknowns+1;
  If TokenType = ID          then Ids:=Ids+1;
  If TokenType = SubCall     then SubCalls:=SubCalls+1;
  If TokenType = NumLit      then NumLits:=NumLits+1;
  If TokenType = AlphaLit    then AlphaLits:=AlphaLits+1;
  If TokenType = Pr          then ProcUsed:=ProcUsed+1;
```

```
  (* Count the number of control flow paths *)
  If IsCyclo(Token) and (Division=Proced)
    then CYCLOMATICS:=CYCLOMATICS+1;

  (* Count the number of parents in the procedure division *)
  If (Token='IN') then ParentsPr:=ParentsPr+1;

  If (Token3='GO') and (Token2 ='TO')
    then Begin
         PushStruc('G',Token1,'');
         Gotos:=Gotos+1;
        end
    else If (Token3='PERFORM') and (Token1='THRU')
          then PushStruc('T',Token2,Token)
          else If (Token3='PERFORM')
              then PushSTruc('P',Token2,'')
              else If (TokenStart3 <12) and (TokenStart3 >=8)
                  then PushStruc('M',Token3,'');

end
else If Division = Data then
    Begin
(*        Writeln('Data Division Found Here ****************'); *)
     (* An identifier parent in the data division *)
     If (TokenType2=NumLit) and (Token='.')
     and (Division=Data)
     then ParentsDa:=ParentsDa+1;
   end;


(* Trace out token characteristics *)
Writeln(Trace,Token:15,' ',ts,' ',ds,' ',LinesOfCode:6,ProgName);

TokenInProcess:=False;
Token3:=Token2;
Token2:=Token1;
Token1:=Token;
Token:='';
TokenType3:=TokenType2;
TokenType2:=TokenType1;
TokenType1:=TokenType;
TokenType:=Trash;
TokenStart3:=TokenStart2;
```

```
      TokenStart2:=TokenStart1;
      TokenStart1:=TokenStart;
      TokenStart:=0;
    End; (* ProcessToken *)


  Begin   (*  Process Character *)

(*      Writeln(InChar,' ',Position);   *)


    (* ABSOLUTE RULES *)

    (* A rule that is in effect on every token *)

    (* If an end of line character not followed by end of line character*)
    If ord(InChar)=13
      Then Begin
           LinesOfCode:=LinesOfCode+1;
          end;

    (* Reset Position Counter Rule *)
    If (Ord(InChar)=10) or (Ord(InChar)=13)
      Then Begin
           Position:=0;
           CommentInProcess:=False;
           ContinuationInProcess:=False;
          end;


    (* Check for Continuation Line *)
    If (Position=7) and (InChar='-')
      Then Begin
            ContinuationInProcess:=True;
          End
       else
          If (Position=7) and TokenInProcess then
             ProcessToken;

    (* Check for Picture Format *)
    If (Token1 = 'PIC') or (Token1='PICTURE')
      then PicFormatInProcess:=True
      else PicFormatInProcess:=False;
```

```
(* Parenthesis Rule *)
(* If (InChar='(') or (InChar=')')
   then InChar:=' ';          *)

(* CONDITIONAL RULES *)

(* Only one rule can be true on any one token *)

(* If continuation and no token in process then cancel *)

If ContinuationInProcess and NOT TokenInProcess
   then ContinuationInProcess:=False;

(* Skip the first 6 characters or line numbers of a card Rule  *)
(* This must be checked for because * is a mathematical symbol *)
(* but the math symbol occures in columns 12+                  *)

If Position <=6 then
else
(* Check for comment line *)
If (Position=7) and (InChar='*')
   Then Begin
         CommentInprocess:=True;
         Case Division of
          Ident  : CommentsId:=CommentsId+1;
          Environ: CommentsEn:=CommentsEn+1;
          Data   : CommentsDa:=CommentsDa+1;
          Proced : CommentsPr:=CommentsPr+1;
        end;

        end
else

(* Skip the characters in comment line *)
If CommentInProcess
   then
else

(* Single Quote Start Rule *)
If (InChar='''') and (not SingleQuoteInProcess)
   and (not DoubleQuoteInProcess)
   then  begin
```

```
            SingleQuoteInProcess:=True;
            TokenInProcess:=True;
            Token:=Token+InChar;
            TokenStart:=Position;
          end
else

(* Double Quote Start Rule *)
If (InChar='"') and (not DoubleQuoteInProcess)
   and (not SingleQuoteInProcess)
   then begin
           DoubleQuoteInProcess:=True;
           TokenInProcess:=True;
           Token:=Token+InChar;
           TokenStart:=Position;
         end
else

(* Single Quote End Rule *)
If (InChar='''') and SingleQuoteInProcess and
   not ContinuationInProcess
   then begin
            SingleQuoteInProcess:=False;
            Token:=Token+InChar;
            ProcessToken;
         end
else

(* Double Quote End Rule *)
If (InChar='"') and DoubleQuoteInProcess and
   not ContinuationInProcess
   then begin
            DoubleQuoteInProcess:=False;
            Token:=Token+InChar;
            ProcessToken;
         end
else

(* Continue with next line *)
If (SingleQuoteInprocess or DoubleQuoteInProcess)
                         and
           ((InChar='''') or (InChar='"'))
                         and
```

```
                 ContinuationInProcess
then ContinuationInProcess:=False
else

(* Add characters in quote to token rule *)
If SingleQuoteInProcess or DoubleQuoteInProcess
  then Token:=Token+InChar
else

(* Check for Hard Delimter *)
If (InChar=' ') and TokenInProcess
   then ProcessToken
else

If ContinuationInProcess and InChar<>' '
   then begin
          ContinuationInProcess:=False;
          Token:=Token+InChar;
        end
else

(* If parenthesis then process them *)

If ((Inchar=')') or (InChar='(')) and not PicFormatInProcess
   then begin
          If TokenInProcess then ProcessToken;
          Token:=Token+InChar;
          TokenStart:=Position;
          ProcessToken;
          TokenInProcess:=False;
        end
else

(* Check for new token starting *)

If (InChar <> ' ') and not TokenInProcess
   then begin
          TokenInProcess:=True;
          Token:=Token+InChar;
          TokenStart:=Position;
         end
else
```

```
  (* If punctuation then process token *)
  If ((InChar = '.') or (InChar = ',') and not PicFormatInProcess)
    then begin
           ProcessToken;
           Token:=Token+InChar;
           TokenStart:=Position;
           TokenInProcess:=True;
         end
  else

  (* Add to the current token *)
  If (InChar <> ' ') and TokenInProcess
    then Token:=Token+InChar;

End; (* ProcessCharacter *)
```

```
egin (* of Process a Program *)

   (* opens the files *)
   Assign (CurProg,ProgName);
   Reset(CurProg);
   Position:=0;

   (* initialize token conditions *)

   TokenInProcess       :=False;
   CommentInProcess     :=False;
   ContinuationInProcess:=False;
   SingleQuoteInProcess :=False;
   DoubleQuoteInProcess :=False;
   PicFormatInProcess   :=False;
   PerformInProcess     :=False;
   TokenStart:=0;

   (* initialize Metrics *)
   LinesOfCode  :=0;
   NumUniqueId  :=0;

   CommentsId   :=0;
   CommentsEn   :=0;
   CommentsDa   :=0;
   CommentsPr   :=0;

   ParentsDa    :=0;
   ParentsPr    :=0;

   Verbs        :=0;
   Adverbs      :=0;
   Punctuations :=0;
   Statements   :=0;

   NumChar      :=0;
   AlphaLits    :=0;

   ProcHeads    :=0;

   Trashs       :=0;
   Unknowns     :=0;
   Ids          :=0;
```

```
  SubCalls    :=0;
  NumLits     :=0;
  LastModNo   :=0;
  Cyclomatics:=1;
  Knots:=0;
  ProcUsed:=0;
  GOTOS:=0;

  token    :='';
  token1   :='';
  token2   :='';
  Token3   :='';
  CurModule :='';
  TokenStart:=0;
  TokenStart1:=0;
  TokenStart2:=0;
  TokenStart3:=0;

  Division:=Ident;

  TopIdQueue:=Nil;
  BotIdQueue:=Nil;
  TopProStack:=Nil;
  TopPtrStack:=Nil;
  TopSwtStack:=Nil;
  TopNotStack:=Nil;
  FrontSTrucQueue:=Nil;
  GetCharacter;
  While NOT EOF(CurProg) Do
    Begin
      ProcessCharacter;
      GetCharacter;
    End; (* end while *)
  ProcessCharacter;




  PopSwitch(TopSwtStack,TopIdQueue,1);
  DestroySwitch(TopSwtStack);
  SearchNot;
  PrintIdQueue;
(*  PrintProcedure;*)
```

```
   PrintNot;
   ComputeIdDistance;
   ComputeIdSize;
   DestroyIdQueue;
   ComputePrDistance;
   Close(CurProg);
   Readln(Dummy);
   PrintStruc;
   RidThru;
   PrintStruc;
   PutJumps;
   PrintStruc;
   FindKnots;
   Writeln(Values,'  Comments Id     ',CommentsId+
           CommentsEn+CommentsDa+CommentsPr:10);          (* Comments*)
   Writeln(Values,ParentsDa:10);                          (* Parents*)
   Writeln(Values,Verbs:10);                              (* Verbs *)
   Writeln(Values,Adverbs:10);                            (* Adverbs *)
   Writeln(Values,NumLits:10);                            (* Num Literals *)
   Writeln(Values,AlphaLits:10);                          (* Alpha Literals *)
   Writeln(Values,NumChar:10);                            (* Char Length *)
   Writeln(Values,ProcHeads:10);                          (* Procedures *)
   Writeln(Values,Cyclomatics:10);                        (* Cyuclomatics *)
   Writeln(Values,Ids:10);                                (* Identifiers  *)
   Writeln(Values,Intermod:10:2);                         (* InterMod *)
   Writeln(Values,Intramod:10:2);                         (* IntraMod *)
   Writeln(Values,CallSpan:10:2);                         (* Call Span *)
   Writeln(Values,ModCall:10:2);                          (* ModCall *)
   Writeln(Values,Knots:10);                              (* Knots *)
   Writeln(Values,Gotos:10);                              (* Gotos *)
   Writeln(Values,ProgName);                              (* Programs *)
   Writeln(Values,' ');

End; (* end process *)
```

```
BEGIN (* main program *)
  (* initialize trace *)
  Assign(trace,'trace');
  Assign(Values,'Values');
  rewrite(trace);
  Append(values);
  Writeln('>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>');
  ProgramToParse:=Paramstr(1);
  PROCESS(ProgramToParse);
                                        close(trace);End.
```

Definition of Model Variables

Reliability	the probability of a program executing without an abend occuring
Comments	the average number of lines of remark statements in each module of the program
Parents	the average number of parents that each identifier has for the entire program
VerbMod	the average number of verbs occuring in each module of the program
AdvMod	the average number of adverbs occuring in each module of the program
LitMod	the average number of literals occuring in each module of the program
AvLenId	the average length in characters of each identifier defined the data division of the program
IdMod	the average number of identifiers (variables) occuring in each module of the program
Modules	the number of modules (subroutines + 1) in the program
CycloMod	the number of control flow paths through the program divided by the number of modules in the program
GotoMod	the number of GOTO statements in the program divided by the number of modules in the program

ModCall	the average number of modules that call each individual module in the program.
KnotMod	the number of times the control flow paths cross in the program divided by the number of modules
CallSpan	the average number of verbs between each call to another module (or subroutine) within a single module of the program
InterMod	the average number of modules that use each identifier (variable)
IntraMod	the average number of verbs between each use of the identifiers (variables) within a single module of the program.
SpecChg	the number of specification changes that were made to the program after it was put into operation
TimExec	the number of times the program has been executed since it was put into operation

Bibliography

AICPA, Statement on Audit Standards No. 48, "The Effects of Computer Processing on the Examination of Financial Statements," (July 1984).

Atkinson, R.C., and R.M. Shiffrin, "The Control of SShort-term Memory," Scientific American, 225, (1971), pp. 82-90.

Binder, L.H. and J.H. Poore, "Field Experiements With Local Software Quality Metrics," Software—Practice and Expericne, Vol 20(7), (July 1990).

Basili, Victor, and Barry Perricone "Software Errors and Complexity: An Emprical Investigation," Communications of the ACM, (January 1984), Vol 27, no. 1, pp. 42-52.

Boehm, B., "Reliability and Complexity," Proceedings of the TRW Symposium on Reliable, Cost-effective Secure Software, March 20-21, 1974, Los Angeles, P. 121-130.

Buxton, J.N., Peter Naur, and Brian Randell, Software Engineering, (New York: Petrocelli/Charter, 1976).

Canning, James, "The Application of Structure and Code Metrics to Large Scale Systems," Unpublished Doctoral Dissertation, VPI (1985).

Choo and Ferrar, "An EDP Audit Model," Journal of Systems and Software, (August 1986), p. 22-26.

Conte, Dunsmore, Shen, "Software Engineering Metrics and Models," The Benjamin/Cummings Publishing Company, Inc., (1986), p. 113-120.

Dale, Nell, and Chip Weems, Turbo Pascal, D.C. Heath and Company, Lexington, Massachcusetts, 1988.

Donaldson, James R., “Structured Programming,” Datamation, Vol. 19, Issue 12, (December 1973).

Dunsmore, H.E., and J.D. Gannon, “Analysis of the Effects of Programming Factors on Programming Effort,” The Journal of Systems and Software 1, Vol. 2, (1980), pp. 141-153.

Elshoff, James L., “An Ulosis of Some Commercial PL/1 Programs,” IEEE Transactions on Software, SE-2, Vol 2, (June 1976), pp. 113-120.

Mark Elson, Concepts of Programming Languages, (Chicago:Science Research Associates, Inc., 1973).

Fainter, Robert, “Program Complexity Measures,” Unpublished Masters Thesis, Virginia Tech, (March 1981).

Gingrich, Gerry, “The Heart Has Its Reasons That Reason Does Not Know,” Journal of End-User Computing, (Winter 1995), Vol 7, No 1, pp. 24-25.

Goel, A.L., and K. Okumoto, “A Markovian Model for Reliability and other Performance Measures,” Proceedings National Computer Conference, (1987), pp. 769-774.

Halstead, M.H., Elements of Software Science, New York: Elsevier North-Holland, (1977), pp. 25-29.

Harold, Frederick Gordon, “An Experimental Analysis of COBOL Program Quality Through the Application of Software Metrics to Programs Written with and without Structured Programming Techniques,” Unpublished Doctoral Dissertation, George Washington University, (1982).

Henderson-Sellers and Tegarden, “The Theoretical Extension of Two Versions of Cyclomatic Complexity to Multiple Entry/Exit Modules,” Software Quality Journal, Vol 3, (1994), p. 253-269.

Henry, Sallie, and Kafura, Dennis, “The Evaluation of Software Systems’ Structure using Quantitative Software Metrics,” Software—Practice and Experience, Vol 14, (June 86).

Kolodziej, J., “COBOL Shapes Up,” Computerworld, Vol 21, Iss 1a, (Jan 7, 1987), p. 13-14.

Lagman, Bernard, “Audit and Control of Systems Programming Activities,” The Institute of Internal Auditors, (1985).

Lawrence, Andrew, “IBM System User International Survey,” (March 96), Computer Business Review, p. 1-4.

Linger, R.C., H.D. Mills, and D.I. Witt, Structured Programming: theory and practice, Addison-Wesley-Publishing Company, Reading, Mass., 1979.

Lipow, M. and T.A. Thayer, “Prediction of Software Failures,” Proceedings 1977 Annual Reliability and Maintainability Symposium, IEEE CAT NO. 77CH11619RQC, p. 489-497 (1977).

Lyman, Ott, “An Introduction to Statistical Methods and Data Analysis,” Second Edition, PWS Publishers, p. 260.

MacGregor, James, “Short-Term Memory Capacity: Limitation or Optimization,” Psychological Review, 1987, Vol 94, No. 1, pp. 107-108.

McCabe, Thomas, “A Complexity Measure,” IEEE Transactions on Software Engineering, SE-2, No. 4, (December 1976), pp. 308-320.

McCabe, Thomas, and Butler, Charles, “Design Complexity Measurement and Testing,” Communications of the ACM, Vol 32, No. 12, (December 1989).

Miller, James, Living Systems, (New York: McGraw-Hill, 1978).

From an interview with David Mitchell, VP of Information Systems at Atlantic Mutual Co., August 14, 1991.

Motley, R.W., and W.D. Brooks, Statistical Prediction of Programming Errors, RADC-TR-77-175, Rome Air Development Center, Griffis Air Force Base, New York, (1977).

Munson, J.C. and T.M. Khoshgoftaar, "Regression Modeling of Software Quality: Empirical Investigation," Information and Software Technology, Vol 32, No 2, (March 1990), p. 106-107.

Musa, Software Reliability: Measurment, Prediction, and Application, McGraw-Hill Book Company, (1987).

Musa, and Okumoto, "A Comparison of Time Domains for Software Reliability Models," Journal of Systems and Software, 4 (4), (1984), p. 277-287

The Office of Technical Assistance Federal Software Support Center Report (February 1992).

Rodriguex, Volney, and Tsai, W.T., "A Tool for Discriminant Analysis and Classification of Software Metrics," (1988), Systems, Software Development Section.

Parikh, Girish, "Making the Immortal Language Work," Business Software Review, Vol 6, Iss 4, (April 1987), p. 33-36.

Sammet, Jean, Programming Languages, Englewood Cliffs NJ: Prentice-Hall, Inc., (1969).

Shelly and Cashman, "Structured COBOL Programming," Roy Forman Boyd & Fraser Publishing Company, pp. 1-2.

Stahl, Bob, "Friendly Mainframe Software Guides Users Toward Productivity," Computer World, February 3, (1986), v20, n5, pp. 53-66.

Steel, Robert and James Torrie, Principles and Procedures of Statistics, New York, McGrawl Hill, (1980).

Takashashi, M. “The Linear Software Reliablity Model and Uniform Testing,” IEEE Transactions Reliability, R-34 (1), (1988), p. 8-16

The Office of Technical Assistance Federal Software Support Center Report, (February 1992).

Thayer, T, and E. Nelson, Software Reliability: A Study of Large Project Reality, North-Holland Publishing Company, Amsterdam, New York (1978).

Tucker, Allen B., Programming Languages, (New York: McGraw-Hill Book Company, 1977).

Weinber, G.M., “The Psychology of Computer Programming,” Von Nostrand Reinhold Company, New York, 1971.

Woodward, M.R., M.A. Hennel, and D. Hedley, “A Measure of Control Flow Complexity in Program Text,” IEEE Transactions on Software Engineering, SE-5, Issue 1, (January 1979), pp. 45-50.

VITA

Henry Jesse Day, II was born on July 18, 1963 in Roanoke Virginia. From then until the age of two his family lived at John's Creek in Craig County, Virginia. Then his family moved to Tazewell, Virginia where his parents still live.

Henry attended Tazewell High School. While there he received a top scholar award in Physics in 1981. Henry also received the American Legion Citizenship award for God and Country in 1981.

In 1981, Henry began studies at Bluefield College in Bluefield, Virginia. He majored in Business Administration and minored in Mathematics. In 1982, Henry received an award for outstanding achievement in College Chemistry. In 1984, Henry graduated with summa cum laude honors. After graduating in 1984, Henry accepted his first teaching position at First Assembly Christian Academy.

At First Assembly Christian Academy Henry taught a variety of courses. Most of the courses were in Mathematics and Science. Henry left First Assembly Christian Academy to pursue graduate studies.

In 1985, Henry entered the Master of Accountancy program at Virginia Tech, which was followed by the Doctor of Philosophy program. In the Ph.D. program, Henry majored in Accounting and minored in Information Systems and Management Science. Henry co-authored the journal article "Computer Ownership and Grades," with Dr. Robert M. Brown and Nancy Meade which was published in **Computers and Education: An International Journal** in January 1989.

From1990 unttil 1995, Henry was an Assistant Professor of MIS at Bluefield College in Southwest Virginia. Henry taught a variety of courses in the computer science program along with quantitative methods and business policy. In 1992, Henry passed the examination and met the experience requirements for the Certified Systems Professional certificate with honors from the American Institute of Certified Computer Professionals (AICCP).

From 1995 until the present Henry has been employed as an Assistant Professor of Business at Mount Olive College in Easter North Carolina. In 1996, Henry passed the exam and was awarded a Certified Computer Programmer Certificate from the AICCP. Henry teaches information systems and accounting courses.