Predicting Maintainability with Software Quality Metrics

by

Steven A. Wake

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Science and Applications

APPROVED:

Dr. Sallie M. Henry, Chairperson

Dr. Dennis G. Kafura

Dr. H. Rex Hartson

Blacksburg, Virginia

Predicting Maintainability with Software Quality Metrics

by

Steven A. Wake

Dr. Sallie M. Henry, Chairperson

Computer Science and Applications

(ABSTRACT)

Maintenance of software makes up a large fraction of the time and money spent in the software life cycle. By reducing the need for maintenance these costs can also be reduced. Predicting where maintenance is likely to occur can help to reduce maintenance by prevention. This thesis details a study of the use of software quality metrics to determine high complexity components in a software system. By the use of a history of maintenance done on a particular system, it is shown that a predictor equation can be developed to identify components which needed maintenance activities. This same equation can also be used to determine which components are likely to need maintenance in the future. Through the use of these predictions and software metric complexities it should be possible to reduce the likelihood of a component needing maintenance. This might be accomplished by reducing the complexity of that component through further decomposition.

Acknowledgements

I would like to thank my advisor, Sallie Henry, for all her help and guidance through my Master's work and especially during the preparation of this thesis. Thank you also to the other members of my committee, Dennis Kafura and Rex Hartson, for their support.

Thanks to my wife for supporting me in many ways and keeping me working when others couldn't. Thanks to my friends, and for calling me long distance to ask me when I would be done. I am also grateful for the support of my short distance friends, and Thanks to for help with statistics and SAS.

Last but not least, "thank yous" go to my parents, and and my brothers and sisters, and for their support and encouragement through all the years.

Table of Contents

Chapter 1 Introduction 1

Introduction 1

The Software Life Cycle 2

Software Maintenance 4

Software Quality Metrics and Maintenance 5

Conclusion 6

Chapter 2 Software Metrics 8

Introduction 8

Code Metrics 9

Structure Metrics 14

Hybrid Metrics 17

Software Metric Analyzer 19

Conclusion 19

Chapter 3 The Experiment 21

Introduction 21

The Digital System 23
The Code Library 24
Data Collection 25
Conclusions 26

Chapter 4 Maintenance Predictions **27**
Introduction 27
Intermetric Results 27
Correlations with Changes 28
The Multiple Regression Model 28
Model Development 32
Prediction Example 33
Conclusions 43

Chapter 5 Conclusions and Future Work **44**
Conclusions 44
Future Work 45

Bibliography **47**

Vita **49**

List of Illustrations

Figure 1. Software Metric Analyzer 20
Figure 2. Top 5 NLC Models Selected by PRESS Statistic 34
Figure 3. Top 5 NLC Models Selected by MSE 35
Figure 4. Top 5 NLC Models Selected by Cp 36
Figure 5. Top 5 NCC Models Selected by PRESS Statistic 37
Figure 6. Top 5 NCC Models Selected by MSE 38
Figure 7. Top 5 NCC Models Selected by Cp 39
Figure 8. Best Overall Candidate Models for NLC 40
Figure 9. Best Overall NCC Models 41
Figure 10. Plot of predicted versus actual values using NLC Model 42

List of Tables

Table 1. Intermetric Correlations 29

Table 2. Correlations with Change Data 30

Chapter 1 Introduction

Introduction

Computer scientists are continually attempting to improve software system development. Systems are developed in a top-down fashion for better modularity and understandability. Performance enhancements are implemented for more speed. One area in which a great deal of effort is being devoted is software maintenance. Brooks [BROF82] estimates that fifty percent of the development cost of a software system is for maintenance activities. Since a large portion of the effort of a system is devoted to maintenance, it is reasonable to assume that driving down maintenance costs would drive down the overall cost of the system.

Measuring the complexity of a software system could aid in this attempt. By lowering the complexity of the system or of subsystems within the system, it may be possible to reduce the amount of maintenance necessary. Software complexity metrics were developed to measure the complexity

of software systems. This study relates the complexity of the system as measured by software metrics to the amount of maintenance necessary to that system.

The Software Life Cycle

To better understand a software system, it is helpful to look at how a system is developed. Most software systems follow a software life cycle. Although some stages of the software life cycle may not be formally present, the basics of that stage of the life cycle are almost certainly used in the development of a software system.

Ramamoorthy divides the software life cycle into seven stages [RAMC84]. The first stage is the understanding of requirements and of the problem in question. During this stage, requirements are developed for the problem that is to be solved by this system, the functionality of the system and any constraints which are to be placed on the system.

Specification of requirements is the second stage of the software life cycle. During this stage, software specialists attempt to understand the requirements and develop a set of specifications of what the system is to do without describing how the system is to do it.

Design is the third stage of the software life cycle. The problem is decomposed into a number of modules and the ways in which they interact. Modules may be further broken down into submodules and procedures in order to obtain a unit that can be easily understood and programmed. The sum of all these units is the design of the system and should meet the specification from the previous stage and answer the question of how the system is to be implemented.

The design from the previous stage is then used in the fourth stage, implementation. In implementation, the modules are coded in a suitable language for the problem.

In the testing stage, the code is executed and bugs may be found and corrected before the system is released. These bugs may occur in the implementation, in the requirements or in the design. Individual modules are tested in this stage along with interactions between modules and, finally, the system as a whole is integrated.

The last two stages are maintenance and evolution. These two stages are similar in that there are changes being made to the system. If bugs are found during the operation of the system, this stage is maintenance, but if performance improvements are made, it could be either evolution or maintenance. The boundary between what is evolution and what is maintenance is usually decided by how major a change is to the system. A major change would be evolution and a minor change is maintenance. Since it is a subjective opinion as to what is major, the distinction between the two phases can be difficult.

There are various other software life cycles proposed in the literature. The model proposed by Bennington [BENH56] is the stagewise model. It proposes nine stages for software development: operational plan, machine and operational specifications, program specifications, coding specifications, coding, parameter testing, assembly testing, shakedown, and system evaluation. Boehm [BOEB76b] suggests that software be developed as a sequence of tasks waterfalling into one another. These tasks are: system requirements, software requirements, preliminary design, detailed design, code and debug, test and preoperations, and operations and maintenance. Each step includes validation and verification activities. Balzer [BALR83] proposes a life cycle model for automatic programming. This model includes steps for requirements analysis, validation, maintenance, mechanical optimization, and tuning. Since maintenance is a critical phase in the software development process, all of these software life cycle models include a maintenance phase.

Software Maintenance

Software maintenance activity is a major part of the software life cycle. Estimates of time and money spent on this stage of the software life cycle range from forty to sixty-seven percent of the total for the entire life cycle [RAMC84, YAUS80]. Lientz [LIEB78] suggests that "maintenance and enhancement tend to be viewed by management as at least somewhat more important than new application software development." Curtis [CURB79] states that "more time is spent maintaining existing software than developing new code." Since "the cost of correcting program errors can (and typically does) increase enormously with time to discovery" [BASV84], it is important to find these errors as early as possible.

Swanson [SWAE76] characterizes three types of maintenance activities. They are corrective maintenance which is performed in response to a failure, adaptive maintenance which is performed in anticipation of a change within the environment of the system, and perfective maintenance which is performed to enhance maintainability. Each of these types of maintenance is an important part of the maintenance process. However, in this study we are looking at corrective maintenance of a software system. By determining where errors occur, we hope to be able to predict where future errors might occur. This allows preventative maintenance to be done to minimize the amount of future corrective maintenance work and makes software systems more sound.

Ramamoorthy [RAMC84] describes three ways to reduce maintenance costs: "(1) The system must be developed with maintenance in mind; (2) The system must be maintained with future maintenance in mind; and (3) The system must be continually upgraded to cope with future technology". He suggests that the use of precise methodologies would solve many maintenance problems. One such method would be formal proofs of correctness. However, proving a program correct is a time consuming process which is sometimes difficult to verify. Another possible method is testing. Since exhaustive testing is not possible, test cases must be carefully designed to cover the entire program. This is generally not possible, therefore bugs can penetrate the code.

Software Quality Metrics and Maintenance

One tool which helps in solving some of the problems of software maintenance is software quality metrics. The metrics quantitatively measure aspects of the system which can be used as indications of the quality of the software system. Metrics can be used at various stages of the life cycle. Ramamoorthy [RAMC84] suggests that metrics can be used for maintenance purposes during the requirements, implementation, testing, and maintenance stages. In this study, we view the software quality metrics as a tool used in the maintenance phase of the software life cycle.

Yau and Collofello [YAUS80, YAUS78] have developed a software metric to measure the ripple effect of modifications in a software system. The ripple effect is "the phenomena by which changes to one program area have tendencies to be felt in other program areas." If the ripple effect is large, a modification to one module of a system may have impact on many other modules in the system. This leads to high maintenance costs and low system reliability.

Basili [BASV83] has attempted to determine the correspondence between the software science measures of Halstead [HALM77] and other related metrics to the number of development errors and to the weighted sum of effort required to isolate and fix these errors on a number of FORTRAN projects. Most of the correlations are weak, but this is attributed to the discrete nature of error reporting and to the fact that most of the modules examined reported zero errors. We are attempting a similar study.

Henry and Kafura [HENS81a] used the information flow metric to analyze the UNIX operating system. They chose UNIX for several reasons including it has large enough size and the fact that it is not a toy or experimental system but a system designed for users. They found that a high

complexity shows stress points in the system or inadequate refinement of the procedure. By correlating changes in the system with the complexity of the procedures they found a high correspondence between these values.

Kafura and Reddy [KAFD87] studied the use of software complexity metrics on several versions of the same software system. The system they studied was a data base management system developed by students at Virginia Tech over a number of years. It is a medium size software system (16,000 lines) written in Fortran. They decided to use a subjective evaluation technique in order to determine whether software metrics could provide information to a maintainer of a system in order to avoid poorly performed maintenance. Subjective evaluation means that they "attempt to relate the quantitative measures defined by the software metrics to the informed judgement of experts who are intimately familiar with the system being studied". An important part of their investigation was examining the changes in the system from one version to the next. They found that the change in the complexities of the software metrics agreed with the changes that one would expect from the changes in the software system. Another interesting finding was that there was a growth in structural complexity as a result of maintenance activity. Two possible uses for software metrics in the maintenance process were suggested. "First, the metrics can be used to identify improper integration of enhancements. ... Second, procedures which are perceived to be complex can lead to improper structuring of the system because maintainers will avoid dealing with this complex procedure when making enhancements, even when the maintainer knows that a major restructuring of the complex component is called for in order to gracefully include the required enhancements."

Conclusion

Software maintenance is an important phase of the software life cycle because large amounts of time and money are spent in the maintenance of existing software systems. The use of software metrics

as indicators of quality could illustrate ways to reduce the effort required for software maintenance and to reduce these high costs.

Chapter 2 describes the various software metrics and the software metric analyzer used in this study. In Chapter 3 the software system which was measured and evaluated for maintenance activity is described. The results of this study using software quality metrics to predict maintenance activity are presented in Chapter 4. Finally, Chapter 5 contains our conclusions and potential directions for future work.

Chapter 2 Software Metrics

Introduction

Software maintenance occurs because software does not do what it was designed to do. Higher quality software is less likely to need maintenance. However, quality is a subjective term. If there is to be an improvement in the quality of software, there must be a way to objectively, or quantitatively, measure quality. This is the realm of software quality metrics. Software quality metrics provide a way to quantitatively measure software quality. These metric values can then be used as indicators of software which is more likely to have necessary maintenance.

There are three classifications of metrics that are used to measure the quality of source code: code metrics which measure physical characteristics of the software such as length or number of tokens; structure metrics which measure the connectivity of the software such as the flow of information through the program and flow of control; and hybrid metrics which are a combination of code and structure metrics.

Code Metrics

Historically, these were the first metrics and are among the simpler metrics to determine. These metrics include length, in lines of code, McCabe's Cyclomatic Complexity and Halstead's Software Science indicators.

Length

Although length may seem a simple measure for software in that it can be determined by just counting the number of lines of code present, there can be different counts depending on what one considers a line of code. For example, comments or blank lines could be included or excluded. Thus, we need a precise definition of what we are counting in order to be able to compare one length to another with confidence. Conte, Shen and Dunsmore propose the following definition for a line of code.

A line of code is any line of program text that is not a comment or a blank line regardless of the number of statements or fragments of statements on the line. This specifically includes all lines containing program headers, declarations, and executable and non-executable statements [CONS86].

In this study the above definition is used for a line of code.

McCabe's Cyclomatic Complexity

McCabe's Cyclomatic Complexity, denoted V(G), was designed to measure the number of distinct paths through a particular program by representing the program with a graph and counting the number of nodes and edges [MCCT76]. The cyclomatic complexity for a graph with e edges and n nodes is:

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

This number, derived from graph theory, is the number of linearly independent circuits in the graph and shows the number of different possibilities that must be tried in order to test each distinct path through the program. This formula can be simplified to the number of simple decisions plus one by using results derived by Mills [MILH72].

McCabe believed that V(G) = 10 would be a reasonable upper bound for the value of this metric.

Halstead's Software Science

Halstead considered a computer program as a collection of tokens which can be classified as either operands or operators. From these measures he developed a number of metrics giving a indication of the complexity of the program [HALM77]. The basic measures are:

$n1$ = the number of unique operators

$n2$ = the number of unique operands

$N1$ = the total occurrences of operators

N2 = the total occurrences of operands

An operator is any token that specifies an action, and an operand is any token that represents data or is acted upon. The size of a program, N, expressed in tokens, is

$$N = N1 + N2$$

Vocabulary is defined as:

$$n = n1 + n2$$

This is a measure that shows that with n operands and operators, the program could be written.

These two measures lead to a third measure which Halstead calls volume:

$$V = N \times \log_2(n)$$

There are two interpretations as to the derivation of this formula. One recognizes the fact that it takes $\log_2(n)$ bits to represent each of the symbols in the vocabulary. Multiplying that by the number of occurrences of these symbols gives the number of bits required to store the algorithm. The second interpretation is that of a programmer making a binary search through the possible symbols at each of the N occurrences.

Program Level

The program level is the ratio of the potential volume, V^*, to the actual volume, V

$$L = \frac{V^*}{V}$$

Potential volume represents the most compact form in which the algorithm could be expressed. Since V^* is not known, an estimator is derived.

The least number of operators that could be used is two, a function call and an assignment. There is no limit to how many operators one could use so the ratio

$$L \sim \frac{2}{n1}$$

was developed.

Looking at operands, we cannot determine a unique minimum value so a ratio is developed intuitively. Whenever an operand is repeated, it indicates that the algorithm's implementation is at a lower level. This is shown by the ratio:

$$L \sim \frac{n2}{N2}$$

Combining the two give us the program level estimator:

$$\hat{L} = \frac{2}{n1} \times \frac{n2}{N2}$$

Programming Effort

Programming effort is a measurement of the effort it takes a programmer to translate ideas about a program solution into the implementation of that solution in a language known to the programmer. The formula for the effort indicator is:

$$E = V \times D$$

where V is the program volume measurement discussed earlier and D is the reciprocal of the level metric, L. By replacing D in the equation we have:

$$E = \frac{V}{L}$$

By substituting L's theoretical value for L we derive:

$$E = \frac{V^2}{V^*}$$

This implies the effort to program the solution to an algorithm varies with the square of the volume. Therefore reducing the volume in ways such as modularization can reduce programming effort.

Structure Metrics

Code metrics measure a static feature of a software system, such as length, but software is not a static entity. In order to get a better measure of the complexity of the code structure metrics attempt to measure the intercommunication features of the code. The Information Flow Metric designed by Henry and Kafura is an example of a structure metric. [HENS81a]

Information Flow Metric

In order to measure the complexity of a procedure with respect to its environment, Henry and Kafura developed the Information Flow Metric. This metric attempts to measure the complexity of the code due to the flow of information from one procedure to another. Flows into a routine are called fan-ins and flows of information out of a routine are called fan-outs. A more formal definition for each is

fan-in the number of local flows into a procedure plus the number of global data structures from which a procedure retrieves information

fan-out the number of local flows from a procedure plus the number of global data structures which the procedure updates

Local flows represent the flow of information to or from a routine through the use of parameters and return values from function calls. Combining these with the accesses to global data structures gives all possible flows into or out of a procedure. The complexity of a procedure is defined as

$$C_p = (fan - in \times fan - out)^2$$

where

Cp = complexity of procedure p

fan-in = the number of fan-ins to procedure p

fan-out = the number of fan-outs from procedure p

McClure's Invocation Complexity

McClure's program complexity metric [MCCC78] uses two aspects of module complexity to define a complexity. The first is the complexity of the circumstances of invoking another module and the second is the complexity of a module invoking another module.

The complexity of invoking a module is dependent on the control variables whose values are used to direct program path selection. A number of variables may be involved in the decision of whether to invoke a module. Together they form the invocation control variable set for the module. A module can have more than one invocation control variable set if it is invoked in more than one location. The complexity of the invocation control variable set is:

$$b \times \sum_{i=1}^{e} C(V_i)$$

where,

b = 2 if any V_i is a control variable of a repetition structure. Otherwise, b = 1.

$C(V_i)$ = Control Variable Complexity

e = the number of variables in the invocation control set.

The complexity of a module invoking another module is a function of the number of modules invoked and the average invocation control variable set complexity of all control variable sets used in invoking the module's direct descendants.

Combining the two yields the following formula:

$$M(p) = [F_p \times X(p)] + [G_p \times Y(p)]$$

where,

F_p = the number of modules that invoke p.

$X(p)$ = the average complexity of all invocation control variable sets used to invoke module p.

G_p = the number of modules invoked by module p.

$Y(p)$ = the average complexity of all invocation control variable sets used by module p to invoke other modules.

Hybrid Metrics

Woodfield's Syntactic Interconnection Model

Woodfield's Syntactic Interconnection Model [WOOS80] is a hybrid metric which attempts to relate programming effort to time. He defines a connection relationship which is a partial ordering between A and B such that one must understand the function of module B before one can understand the function of module A.

There are three types of module connections possible: control, data and implicit. A control connection implies an invocation of one module by another. A data connection occurs when a module uses a variable modified by another module. An implicit connection occurs when there are some assumptions used in one module that are also used in another module. One example is that two modules may both make the assumption that input as an expression of eighty characters or less. If this ever changes then both modules have to be modified to reflect that change.

The connection A -> B implies that some aspect of module B must be reviewed and understood before module A is completely understood. The number of times a module must be reviewed is Woodfield's definition of the module's fan-in.

He presents the following general equation for the model:

$$C_b = C_{1b} \times \sum_{k=2}^{fan\text{-}in_i - 1} RC^{k-1}$$

where

C_b = the complexity of module B's code

C_{1b} = the internal complexity of module B's code

fan-in = the sum of the control and data connections for B's code

RC = a review constant

The internal complexity for the module can be any code metric, however in Woodfield's model definition Halstead's Program Effort Metric was used. The model uses a review constant of 2/3 which is a number previously suggested by Halstead.

Information Flow Metric

Henry and Kafura's information flow metric can also be used as a hybrid metric. As a hybrid metric, the formula for the complexity is:

$$C_p = C_{ip} \times (fan - in \times fan - out)^2$$

where

C_p = complexity of procedure p

Cip = the internal complexity of procedure p

fan-in = the number of fan-ins to procedure p

fan-out = the number of fan-outs from procedure p

C_{ip} may be any code metric's measure of procedure p's complexity.

Software Metric Analyzer

The Software Metric Analyzer is a tool developed under the direction of Dr. Sallie Henry at Virginia Tech. Given the source code as input it calculates each of the metrics discussed previously [HENS88].

Using the UNIX tools LEX and YACC (yet another compiler compiler) along with a BNF grammar for the given language the language dependent portion, pass 1, of the analyzer calculates the code metrics and translates the source code into a language independent, code encrypted relation language.

The relation manager, pass 2, takes the relation language code from pass 1 and translates it into a series of relations which along with the code metrics from pass 1 are the input to pass 3 of the analyzer [HENS88]. Pass 3 of the analyzer uses the relations to calculate the structure metrics [KAFD82]. Both the code metrics and the structure metrics can be combined to create the hybrid metrics. Pass 3 also displays the metrics in various groupings such as by metric or by procedure.

Conclusion

The software engineering community has developed a number of quantitative measures, or metrics, to use in the measurement of the quality of software. Previous studies have shown that these metrics do point out modules of high complexity within a program. We wish to determine if these modules of high complexity are also the modules most likely to need maintenance [KAFD87, HENS81a]. The next chapter describes the used for this metrics experiment.

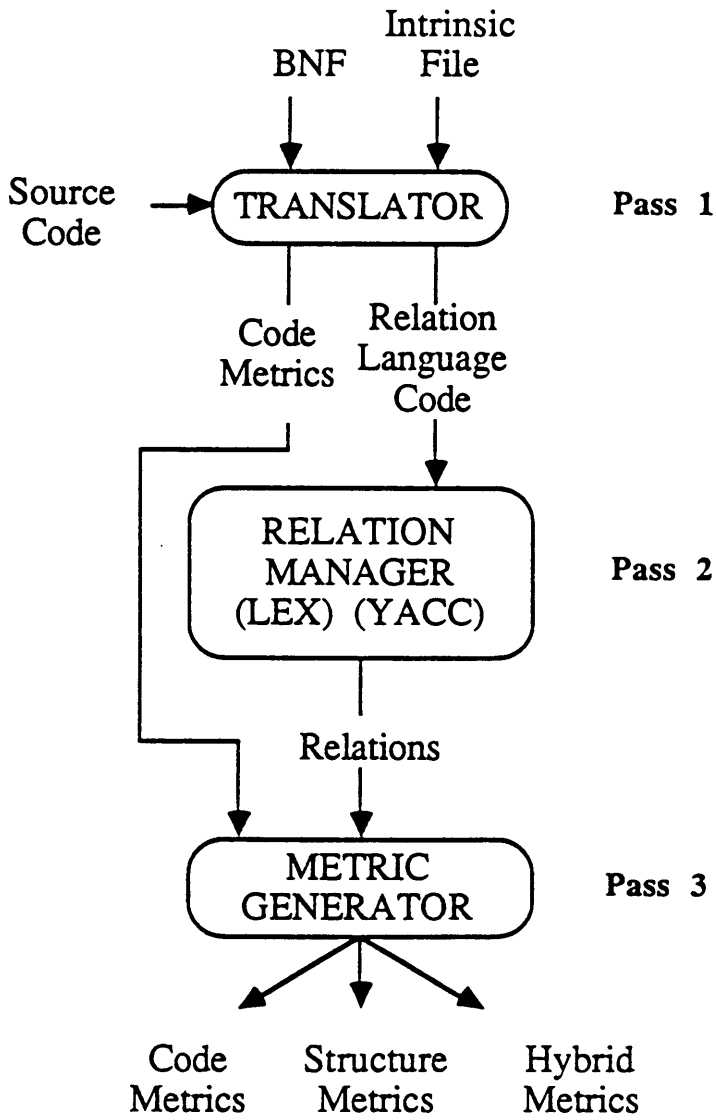

Figure 1. Software Metric Analyzer

Chapter 3 The Experiment

Introduction

In order to perform a valid research effort using Software Quality Metrics to predict system maintenance, actual data must be used. Obtaining real data is a difficult task for academicians. If student data is used, the experiment may be conducted in too controlled an environment and not reflect "real world" conditions.

A student program is typically assigned to teach one or more specific concepts to the student. Typically, the student is told to expect syntactically correct data with an established input format. Students tend to test their programs until the professor's data works correctly, however, they do not test for unexpected or incorrect input data. Student programs are typically throw-away programs which are not maintained. This makes student programs a bit more simplistic than real world programs. The maintenance phase of the software life cycle is the focus of this research.

In the area of maintenance of programs it is even more difficult since students typically use a program only once. A program is designed, coded, debugged, tested, and then run once on the test data provided. The program is then graded and marked down for any bugs. No performance improvements are suggested to be added to the program. This means that there is no maintenance of the program. It is almost impossible to convince students to keep track of their bugs during the development and testing phases because of the nature of the programs (i.e. for a graded assignment).

Thus the best data comes from industry. Since most data from industry comes from proprietary software, it is difficult to obtain this information. Pass 1 of the Software Metric Analyzer described in Chapter 2 disguises the information in the source code into a relation language so that the source code can be measured without revealing the data structures or actual algorithms. [HENS88]

Understandably, most corporations are evasive on disclosing data for fear of someone saying that the quality of a product they develop is low. Developers are also hesitant about letting researchers measure their code for fear that the measurement may be used against them in an evaluation process. Industry generally wants tried and proven measures that they can use "in house" to improve the quality of their software without having the results made public. There is almost no real data in the public domain for researchers to use. However, most research on the measurement of software quality is developed in academic institutions.

A research relationship between the Software Engineering group at Virginia Tech and Digital Equipment Corporation has been ongoing for the last several years. Digital understands the need of academicians to have access to real data. Cooperative research benefits Digital by improving the quality of their software and benefits academia by providing real data for software engineering researchers.

The Digital System

The software system used for this experiment is an actual Digital product consisting of 193 procedures comprising about 15,000 lines of 'C' code. This number includes comment lines and blank lines. The project is composed of a number of modules, each with a separate function. Each module was composed of one or more procedures having a like function such as all the string handling routines or all the parsing routines. The status of the code at the time of measurement is after the release of version 2.0.

The modules were processed by the front end of the analyzer separately, at Digital, generating a file containing the values for the code metrics and a separate relation file containing the relations for the procedures in the module. Recall from the previous chapter that the code metrics include lines of code, values for Halstead's Software Science indicators and McCabe's Cyclomatic Complexity. These metrics are generated by Pass 1 of the analyzer because they are language dependent counts generated using the original source code before it has been disguised in any way.

The relation files of all the modules are concatenated together to form a single relation file for input to Pass 3 of the analyzer. By putting the relations together in a single file we use all the relationships between procedures whether or not the routines are contained in the same physical module. In this way we are able to determine all relationships between modules for the computation of structure metrics.

In order to verify the interpretation of the metric numbers generated, there must be control data against which the interpretation can be tested. Where software metrics are a guide to maintenance of a software product, it is useful to see what changes are necessary to the product after a major release. Since the last major version of this product was version 2.0, the version which was measured, any modifications to the source code after this time are for maintenance reasons. This could be for bug fixes or performance improvements but is not new development of any kind. This data

was obtained from a code library which was used in the development and maintenance of the product.

The Code Library

A code library was used to monitor accesses to the different modules of the product. All source code in a code library could be accessed by the code librarian program. DEC/CMS is Digital's code management system or code librarian. When a bug is found by a customer a Software Performance Report is sent to the maintainers of the product who determine what changes, if any, are necessary. Any changes made to the source code are done through the use of the code librarian. As used in their development and maintenance strategy, a module of source code is checked out of the code library when changes are to occur. After the change is made, tested, and found to be correct, the changed module is checked back into the code library. Each time new changes are to be made to the code, the module involved must be checked out of the library and the corrected version checked in. This enables an automated history to be kept of accesses to a module of code.

CMS keeps track of code changes so that all versions of a module of source code are always available. This is accomplished by flagging the changes and monitoring their effect on the current version of the source code which is in the library. Thus the addition of one line of code to a module results in the addition of one line of code to the version of the module in the code library along with the addition of several control records to regulate which versions contain this new line of code. The code librarian can then reconstruct the desired version of a module by executing the proper changes. A side effect is that the librarian can store many versions of the source code without needing all the extra disk space required if the separate versions were saved.

It is also possible to group different versions of the modules in the library into a collection called a class. A class contains the desired version of each module at some point in time, such as version 1 or last tested version. By specifying retrieval of the proper class, it is possible to retrieve the version of the source code which was shipped as the product. After retrieving the code as it was at shipping time, the modules can be processed by the Software Metric Analyzer to obtain values for the code for that release. Any changes to the code after that time are changes for maintenance reasons.

The smallest unit of change in CMS is the line. A line can be either added to or deleted from a module. A modification to a line is therefore treated as a deleted line followed with an added line in the same place. For purposes of verification any changes or modifications described are based on the changing of a line of the code. Groups of lines that are all changed at the same time can also be determined.

Data Collection

Through the use of a callable interface to CMS it is possible to make a complete listing of the source code with the lines of code added or deleted flagged to show when they were added and deleted. A routine was written to find these flagged lines in the source code and count the number of additions and deletions and where they occurred. By coupling this with a routine to determine which procedure the changes occurred, a count of the number of changes, the number of lines changed, and how they were changed is obtained. These numbers can serve as control data for the interpretation of the metric numbers as they apply to the maintenance phase of the software life cycle.

By tabulating the results of these data collection routines a list of all the routines along with the corresponding number of lines added after the latest release, number of lines deleted, and number of times these changes were made to each routine in the program. This gives an indication of the maintenance activity which occurred to the program.

Conclusions

In order to show that metrics are applicable to "real world" situations it is necessary to gather data that has been developed in the real world without having the process controlled. Data has been presented which was developed by a computer vendor and is now in the maintenance phase of its life cycle. In the following chapter we show the results of measuring the data and relating these measurements to the history of bugs found in the program.

Chapter 4 Maintenance Predictions

Introduction

This chapter presents the results of the statistical analysis of the data collected. Interrelationships among the various metrics and the changes to the code are shown. A discussion is presented on the various statistics used and the multiple regression model. A statistical analysis of the data is presented using the multiple regression model for both number of changes and number of lines of code changed. Finally, the model developed is presented.

Intermetric Results

Recall from Chapter 2, the various metrics used in this study: Length, Halstead's N, V, E, McCabe's Cyclomatic Complexity, V(G), Woodfield's Complexity, Information Flow, Information Flow with Length, and Information Flow with Effort. Statistical correlations are presented in

Table 1 on page 29. Note that there is a high degree of correlation among the code metrics. This occurs because the code metrics are all attempting to measure some aspect of the code. The code metrics do not correlate well to either the structure or the hybrid metrics. This indicates that each of these types of metrics are attempting to measure a different aspect of the code. These correlation results agree with other studies done in the software metrics area [HENS81b, CANJ85].

Correlations with Changes

Table 2 on page 30 shows the correlation among the various metrics and the number of code changes (NCC) and the number of lines of code changed (NLC). Halstead's Effort, E, shows the best correlation in each of these cases. Since it requires no more work to collect the other metric values, it was decided to determine if a greater degree of accuracy in the prediction of these values could be obtained by using more than just a single value. This can be done by using the multiple regression model.

The Multiple Regression Model

In some cases a linear relationship is not present between a single variable and the dependent variable in a collection of data. When this occurs it may be necessary to express the prediction model as a multiple linear regression model or just multiple regression model. This indicates that more than one independent variable has some bearing on the value of the dependent variable. Looking at correlations is not sufficient to determine which independent variables are to be a part of the

	Length	N	V	E	V(G)	Wood	Info-L	Info-E	Info
Length	1.000								
N	0.842	1.000							
V	0.973	0.862	1.000						
E	0.740	0.370	0.758	1.000					
V(G)	0.840	0.762	0.770	0.420	1.000				
Wood	0.436	0.485	0.434	0.215	0.310	1.000			
Info-L	0.065	0.110	0.067	0.011	0.022	0.088	1.000		
Info-E	0.138	0.170	0.158	0.103	0.091	0.113	0.838	1.000	
Info	-0.077	-0.062	-0.068	-0.049	-0.093	-0.051	0.83	0.502	1.000

Table 1. Intermetric Correlations

	NCC	NLC
Length	0.5420	0.5254
N	0.2645	0.1854
V	0.5427	0.5501
E	0.7490	0.8610
V(G)	0.3034	0.2015
Wood	0.1291	0.1283
Info-L	-0.0354	0.0318
Info-E	-0.0462	0.0423
Info	0.0198	0.0340

Table 2. Correlations with Change Data

multiple regression model since there may be some interactions among two or more independent variables which better explains the variation in the dependent variable.

There are three steps involved in determining the multiple regression equation. First, determine the dependent variable and the candidates for the independent variables. The selection of the independent variables is determined through the use of various statistical measures run for all subsets of the independent variables. Next, form the model equation from the selected variables. Finally, make sure the model does not violate the multiple regression model assumptions of normality and independence. [OTTL84]

There are several different statistics available to consider when choosing the variables for the multiple regression equation. These include the PRESS statistic, Mallows' Cp, and MSE (mean squared error). The best fitting model should have a Cp approximately equal to the number of independent variables and low values for both PRESS and MSE.

The PRESS (Prediction Sum of Squares) statistic [MYER87] is a good statistic for a predictive model because it takes into account the effects of fitting one value out of a set of data to the model specified by the rest of the data. Consider a set of N data observations. To calculate the PRESS statistic we first withhold the first data observation and calculate the coefficients for the model with the other N - 1 observations. This is repeated for each of the N data observations resulting in N prediction errors or residuals. The sum of the N residuals gives us a value for the PRESS statistic. Thus, the model with the lowest PRESS value is a candidate for the best model.

Mean Squared Error (MSE) takes into account the error from the predicted value and the actual value. A model is fit to the observations and an expected value is calculated for each independent variable based on the dependent variables associated with it. The difference between the actual value and the predicted value is squared. This is done for all N observations and the sum of these values is added and a mean calculated. Again, the model with the lowest MSE is a good candidate for the best model.

Mallows' Cp [MYER87] takes into account the fact that there is a prediction variance in the MSE statistic. This variance, summed over the N data observations is equal to the number of independent variables in the model. Thus, the best Cp is the one with the value close to the number of parameters in the model.

Model Development

Recall that 193 procedures were analyzed in the software system studied. Values for each of the metrics were calculated along with observations on the number of code changes (NCC) and the number of lines changed (NLC) in each procedure. Two possibly different multiple regression models can be developed, one for each of the independent variables, NLC and NCC.

The observations were randomly divided into two groups, one with three fourths of the data and the other with the other fourth of the data. The large group is used to determine the predictive model and the small group is used to verify the model. Only the large group of data is used in the determination of the model so that the smaller dataset has unknown values which would be predicted.

Separate attempts at model development are then attempted for each of the independent variables. Using the SAS statistics package, values for each of the three statistics discussed earlier are calculated for each model possible using a subset of the independent variables. Since each of the statistics can give a different "best" model, there is only an indication of what the best models are. A set of best models can then be chosen and the unused data can be used to select the best model of that group.

The following figures present the top five models as selected by each statistic for NLC and NCC. Figure 8 and figure 9 show the best overall candidate models. By calculating the sum of squared errors for each model using the unused data a "best" model is chosen.

The final step in choosing a good multiple regression model is checking that none of the equations that are candidates for the final model violate any assumptions necessary for multiple regression analysis. All of the candidate models for NLC are valid in this respect. However, looking at diagnostics for the NCC models show that there is collinearity in the model using N, V, and E and the model using L, V, and E. Collinearity means that values of one independent variable are strongly related or depend on the values of one or more of the other independent variables. This violates our assumption of independence and makes these models poor choices for the final model. Any of the other choices better explain the variation due to this fact.

Prediction Example

As an example, consider the NLC model with the three independent variables, E, V(G) and Info-E. This is chosen as the best overall model by calculating the sum of squared error for each of the best candidate models over the final quarter of the data and choosing the lowest value. This model is in the top 5 as selected by Cp and MSE. It also did well in the PRESS statistic. The equation for this model is:

$$NLC = 1.51830192 + 0.000054724E - 0.10084685V(G) - 0.000000161798INFO - E$$

To predict the amount of maintenance in lines of code changed for a given procedure take the values calculated for each of the metrics used in the model and put them into the regression equation calculated. Consider values of 145,335 for E, 59 for V(G) and 1,308,016 for Info-E. This yields a value of 3.31 for NLC as opposed to 3.0 which is the number of lines actually changed. Taking values of 12,246 for E, 21 for V(G) and 195,929 for Info-E yields a value of 0.0389 for NLC. This procedure required no changes in actuality. A project with these calculated values would want to concentrate on the higher valued procedure if there is time to preventative maintenance.

$$NLC = 0.42997221 + 0.000050156\ E - 0.0000001992\ INFO\text{-}E$$

$$NLC = 0.45087158 + 0.000049895\ E - 0.000173851\ INFO\text{-}L$$

$$NLC = 0.60631548 + 0.000050843\ E - 0.000029819\ WOOD - 0.000000177\ INFO\text{-}E$$

$$NLC = 0.33675906 + 0.000049889\ E$$

$$NLC = 0.62562353 + 0.000050633\ E - 0.000030739\ WOOD - 0.000147075\ INFO\text{-}L$$

Figure 2. Top 5 NLC Models Selected by PRESS Statistic

$$NLC = 1.27935618 + 0.05500043\ L - 0.001333387\ V + 0.000054797\ E - 0.11960695\ V(G) - 0.0000001429\ INFO\text{-}E$$

$$NLC = 1.51830192 + 0.000054724\ E - 0.10084685\ V(G) - 0.0000001618\ INFO\text{-}E$$

$$NLC = 1.2782025 + 0.05693335\ L - 0.001428534\ V + 0.000054898\ E - 0.11900135\ V(G)$$

$$NLC = 1.30521150 + 0.06024787\ L - 0.001438433\ V + 0.000054545\ E - 0.12321067\ V(G) - 0.000163532\ INFO\text{-}L$$

$$NLC = 1.53080447 - 0.000355426\ V + 0.000056495\ E - 0.084191\ V(G) - 0.0000001493221\ INFO\text{-}E$$

Figure 3. Top 5 NLC Models Selected by MSE

$$NLC = 1.47735619 + 0.000054638\ E - 0.10017668\ V(G) - 0.00000673\ WOOD$$

$$NLC = 1.57295997 + 0.000054615\ E - 0.1003767\ V(G) - 0.0000051632\ WOOD - 0.00015725\ INFO\text{-}L$$

$$NLC = 1.51830192 + 0.000054724\ E - 0.10084685\ V(G) - 0.0000001618\ INFO\text{-}E$$

$$NLC = 1.45518829 + 0.00005456\ E - 0.10199539\ V(G)$$

$$NLC = 1.57353731 - 0.002446765\ N + 0.000054672\ E - 0.08863879\ V(G)$$

Figure 4. Top 5 NLC Models Selected by Cp

$$NCC = 0.40552034 + 0.00001163\ E - 0.000006267\ WOOD - 0.0000000478\ INFO\text{-}E$$

$$NCC = 0.38710077 + 0.00001158\ E - 0.000006897\ WOOD$$

$$NCC = 0.41056545 + 0.00001157\ E - 0.000006517\ WOOD - 0.000039367\ INFO\text{-}L$$

$$NCC = 0.36846091 + 0.00001149\ E - 0.00000005238\ INFO\text{-}E$$

$$NCC = 0.34394979 + 0.000011418\ E$$

Figure 5. Top 5 NCC Models Selected by PRESS Statistic

$$NCC = 0.25438629 + 0.0043307\ N - 0.00062705\ V + 0.00001471\ E - 0.0000000525\ INFO\text{-}E$$

$$NCC = 0.17374520 + 0.0127384\ L + 0.00369907\ N - 0.00089433\ V + 0.000014349\ E$$

$$NCC = 0.25250119 + 0.0039729\ N - 0.00059868\ V + 0.00001454\ E$$

$$NCC = 0.32020501 + 0.0136926\ L - 0.00048185\ V + 0.00001230\ E$$

Figure 6. Top 5 NCC Models Selected by MSE

$$NCC = 0.34394979 + 0.000011418\ E$$

$$NCC = 0.41704985 - 0.000129275\ V + 0.000012345\ E$$

$$NCC = 0.36846091 + 0.000011488\ E - 0.00000005238\ INFO\text{-}E$$

$$NCC = 0.3871077 + 0.000011583\ E - 0.0000068966\ WOOD$$

$$NCC = 0.25250119 + 0.00397286\ N - 0.00059868\ V + 0.000014538\ E$$

Figure 7. Top 5 NCC Models Selected by Cp

$$NLC = 0.42997221 + 0.000050156\ E - 0.00000019921\ INFO\text{-}E$$

$$NLC = 0.45087158 + 0.000049895\ E - 0.000173851\ INFO\text{-}L$$

$$NLC = 0.60631548 + 0.000050843\ E - 0.000029819\ WOOD - 0.0000001773\ INFO\text{-}E$$

$$NLC = 0.33675906 + 0.000049889\ E$$

$$NLC = 1.51830192 + 0.000054724\ E - 0.10084685\ V(G) - 0.0000001618\ INFO\text{-}E$$

$$NLC = 1.45518829 + 0.00005456\ E - 0.10199539\ V(G)$$

Figure 8. Best Overall Candidate Models for NLC

$$NCC = 0.34294979 + 0.000011418\ E$$

$$NCC = 0.36846091 + 0.000011488\ E - 0.00000005238\ INFO\text{-}E$$

$$NCC = 0.38710077 + 0.000011583\ E - 0.0000068966\ WOOD$$

$$NCC = 0.25250119 + 0.003972857\ N - 0.000598677\ V + 0.000014538\ E$$

$$NCC = 0.32020501 + 0.01369264\ L - 0.000481846\ V + 0.000012304\ E$$

Figure 9. Best Overall NCC Models

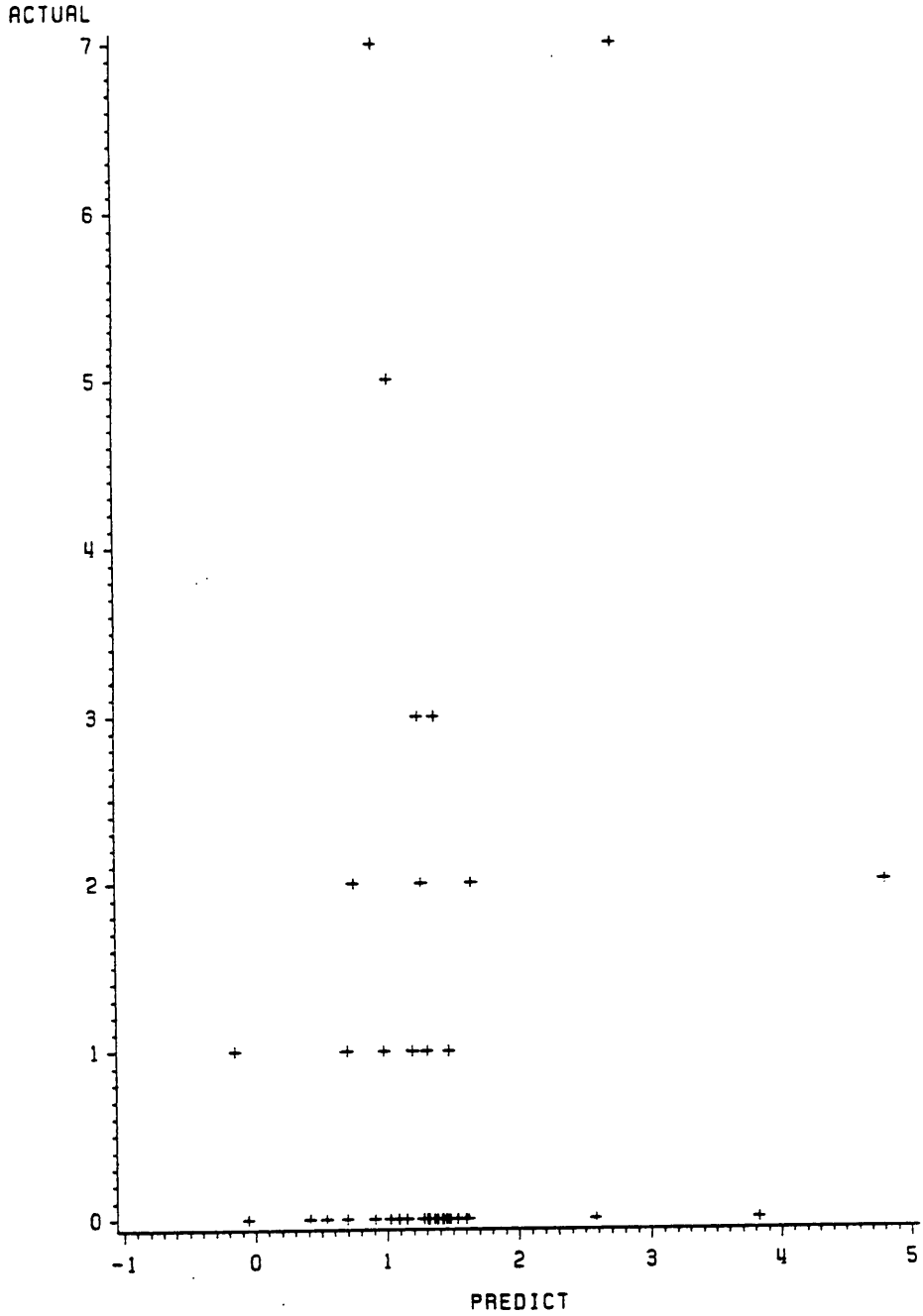

Figure 10. Plot of predicted versus actual values using NLC Model

Figure 10 shows a plot of predicted against actual values using the model with E, V(G) and Info-E for the quarter of the data used to check the model. The large number of zero values in the actual data gives the model some trouble but overall the highest actual values tended to be in the highest predicted values. The multiple regression model does not come up with values of zero due to the fact that it is fitting a line to a number of data points with different values for the independent variables. This makes predictions have higher for the low values and lower for the high values. This is another reason that the developed equation should be used to rank procedures and not to obtain exact values.

Conclusions

Using multiple regression techniques helps to develop a predictor equation which fits the data specified. The multiple regression model is a powerful method of determining which independent variables best work together to predict the dependent variable. A number of statistics are available for use in determining the best predictor equation.

Past maintenance activity along with metric values for the procedures in a system are used together to develop an equation which explains the data. This equation is used to predict future maintenance activity or areas where further work is needed.

The predictor equation is shown to work for several cases. It is not intended to determine an exact value but rather a relative value which can be compared to other values for procedures in the system.

Chapter 5 Conclusions and Future Work

Conclusions

Analyzing a software system along with its past maintenance history can lead to indications of future maintenance work. Preventative maintenance can help hold down the high cost of maintenance. By determining where maintenance might occur and improving that code, future problems and costs can be minimized.

Values calculated using the regression equation are not meant to be an exact value for the amount of maintenance that will occur on a given procedure. Rather they are meant to give an indication of how much maintenance will occur. Analyzing a number of procedures creates a ranking of these procedures which can be used to determine on which preventative maintenance would be most useful. Future maintenance (and costs) could be lessened with an approach like this.

No single metric seems to be able to determine the overall quality of a software system. For this reason the multiple regression approach presented here seems to be a good method for developing

a predictor for a system. For an organization to use this approach, it must first collect a significant amount of error data (on maintenance of systems) and the corresponding metric values. Second, fit the error data to the metrics using multiple regression analysis. All organizations develop software using different tools and different environments. Developing the multiple regression equation for a specific environment requires data on programs that is specific to the environment in which it is developed, the application, and the language used by the software development organization. After developing and verifying the predictor equation for the data collected, it is ready to be used in more productive ways. This equation should be applied during coding, testing and maintenance. During coding, complicated procedures can be considered for redesign. At the end of the coding phase, procedures can be ranked as to likelihood of necessary maintenance. Preventative maintenance at this point will hold down future costs.

Future Work

This thesis presented results from one software system, in one language, developed in one specific environment. Although an approach for an entire organization is presented, no work has been done in this area. The next step would be to find several software systems developed in the same environment and the same language and determine if they can be used to find a single predictor equation which will cover all systems developed in that same manner.

Another possibility is to see how different languages and different environments affect the results. These results are very specific to a language and environment. Changing either of these parameters could lead to interesting relationships between the language, the environment and the type of software system.

Cooperation between industry and academics results in benefits to both. Academics receives real world data for testing of theories and new approaches to problems. Industry receives more thor-

ough and realistic solutions to problems. This mutually beneficial approach should continue for best results for both.

Bibliography

[BALR83] Balzer, R., Cheatham, T.E., Green, C., "Software Technology in the 1990's: Using a New Paradigm", *IEEE Computer*, November 1983.

[BASV84] Basili, V.R., Perricone, B.T., "Software Errors and Complexity: An Empirical Investigation", *Communications of the ACM*, January 1984.

[BASV83] Basili, V.R., Selby, R.W., Phillips, T., "Metric Analysis and Data Validation Across Fortran Projects", *IEEE Transactions on Software Engineering*, November 1983.

[BENH56] Bennington, H.D., "Production of Large Computer Programs", *Proceedings of ONR Symposium on Advanced Programming Methods for Digital Computers*, June 1956.

[BOEB76a] Boehm, B.W., Brown, J.R., Lipow, M., "Quantitative Evaluation of Software Quality", *Proceedings Second International Conference on Software Engineering*, 1976.

[BOEB76b] Boehm, B.W., "Software Engineering", *IEEE Transactions on Computers*, December 1976.

[BROF82] Brooks, Jr., F.P., *The Mythical Man Month*, Reading, MA, Addison-Wesley Publishing Co., 1982.

[CONS86] Conte, S.D., Dunsmore, H. E., Shen, V. Y., *Software Engineering Metrics and Models*, Menlo Park, CA, The Benjamin/Cummings Publishing Company, Inc., 1986.

[CURB79] Curtis, B., Sheppard, S.B., Milliman, P., Borst, M.A., Love, T., "Measuring the Psychological Complexity of Software Maintenance Tasks with the Halstead and McCabe Metrics", *IEEE Transactions on Software Engineering*, March 1979.

[HALM77] Halstead, M., *Elements of Software Science*, New York, NY, Elsevier North Holland, Inc., 1977.

[HENS79] Henry, S., *Information Flow Metrics for the Evaluation of Operating Systems' Structure*, Ph. D. Dissertation, Iowa State University, Computer Science Department, 1979.

[HENS81a] Henry, S.M., Kafura, G.D., "Software Structure Metrics Based on Information Flow", *IEEE Transactions on Software Engineering*, September 1981.

[HENS81b] Henry, S.M., Kafura, G.D., Harris, K., "On the Relationships Among Three Software Metrics", *Performance Evaluation Review*, Vol. 10, No. 1, Spring 1981.

[HENS88] Henry, S.M., "A Technique for Hiding Proprietary Details While Providing Sufficient Information for Researchers", *Journal of Systems and Software*, 1988 (to appear).

[KAFD82] Kafura, D., Henry, S., "Software Quality Metrics Based on Interconnectivity", *Journal of Systems and Software*, Vol. 2, 1982.

[KAFD87] Kafura, D., Reddy, G.R., "The Use of Software Complexity Metrics in Software Maintenance", *IEEE Transactions on Software Engineering* , March 1987.

[LIEB78] Lientz, B.P., Swanson, E.B., Tompkins, G.E., "Characteristics of Application Software Maintenance", *Communications of the ACM*, June 1978.

[MCCT76] McCabe, T., "A Complexity Measure", *IEEE Transactions on Software Engineering*, December 1976.

[MCCC78] McClure, C., "A Model for Program Complexity Analysis", *Proceedings Third International Conference on Software Engineering*, Atlanta, GA, May 1978, 149-157

[MILH76] Mills, Harlan D., "Software Development", *IEEE Transactions on Software Engineering*, December 1976.

[MUNJ78] Munson, J.B., "Software Maintainability: A Practical Concern for Life-Cycle Costs", *Proceedings of the IEEE Computer Science and Applications Conference*, 1978.

[MYER87] Myers, R.H., *Classical and Modern Regression with Applications*, Duxbury Press, 1987.

[OVEC86] Overstreet, C.M., Nance, R.E., Balci, O., Barger, L.F., "Specification Languages: Understanding Their Role in Simulation Model Development", *Technical Report SRC-87-001, Systems Research Center, Virginia Tech, Blacksburg, VA*, December 1986.

[OTTL84] Ott, L., *An Introduction to Statistical Methods and Data Analysis*, Duxbury Press, 1984.

[RAMC84] Ramamoorthy, C.V., Prakash, A., Tsai, W., Usuda, Y., "Software Engineering: Problems and Perspectives", *IEEE Computer*, October 1984.

[SHEV85] Shen, V.Y., Yu, T., Thebaut, S.M., Paulsen, L.R., "Identifying Error-Prone Software -- An Empirical Study", *IEEE Transactions on Software Engineering*, April 1985.

[WOOS80] Woodfield, S., *Enhanced Effort Estimation by Extending Basic Programming Models to Include Modularity Factors*, Ph. D. Disertation, Purdue University, Computer Science Department, 1980.

[YAUS78] Yau, S.S., Collofello, J.S., MacGregor, T., "Ripple Effect Analysis of Software Maintenance", *Proceedings of the IEEE Computer Science and Applications Conference*, 1978.

[YAUS80] Yau, S.S., Collofello, J.S., "Some Stability Measures for Software Maintenance", *IEEE Transactions on Software Engineering*, November 1980.

The vita has been removed from the scanned document