Nancy Currans

A Comparison of Counting Methods For
Software Science And Cyclomatic Complexity

APPROVED:

Curtis R. Cook

Professor of Computer Science in Charge of Major

Chairman of Department of Computer Science

Date research paper is completed 25th November 1985

A COMPARISON OF COUNTING METHODS FOR SOFTWARE SCIENCE AND CYCLOMATIC COMPLEXITY

Nancy Currans
Curtis Cook

Abstract

This paper reviews McCabe's cyclomatic complexity and Halstead's laws; it discusses studies in current literature relating the metrics to software. The studies are reproduced using data obtained from a large software project developed in a major electronics firm. Problems that occur when deriving the metrics are discussed; the result of computing the metrics different ways is investigated.

This study shows that when the counting method is varied, there is no significant difference in the bug-to-metric relationship. A strong relationship is shown to exist between Halstead's laws, McCabe's cyclomatic complexity, lines of code, and the number of bugs reported in the project.

Table of Contents

Chapter One: Introduction

The term "software complexity" means different things in the field of computer science. The more traditional meaning deals with computational complexity - the time and space complexity of an algorithm. Time complexity is the amount of time it takes an algorithm to compute. Space complexity is the amount of memory an algorithm requires.

A second more recent meaning of complexity, or "psychological complexity", relates to a human's performance when working with or using a piece of software. It studies the difficulty of a programmer working with (testing, modifying, debugging, etc.) a program, the difficulty a user has in understanding how to use the software, or the difficulty or ease of using the computer due to the physical layout of the hardware.

This paper investigates the difficulty a programmer has working with a program. The relationship between program characteristics and errors found in that program are studied. The data was obtained from an electronics firm in the Pacific Northwest: program characteristics were taken from the source code and error data was collected when engineers analyzed problem reports and made bug fixes.

Lines of code, along with two of the more popular software complexity metrics are reviewed: McCabe's cyclomatic complexity, and Halstead's software science. Current studies relating the metrics to software are summarized; they are then re-

peated using the data collected for this study. Problems that occur in computing the metrics are discussed, and the resolution to those problems for the purpose of this study is given.

Although it is dangerous to draw conclusions from one specific study and then generalize them to all software, these results support the conclusions of many published articles. McCabe's metrics and Halstead's laws are computed different ways; the results of the study show that the counting methods are strongly related to each other, and the statistical results are essentially the same no matter which counting method is used. The popular metric lines of code without comments has among the strongest relationship to errors when compared to either McCabe's or Halstead's metrics.

Further studies must be performed using code from other actual software projects for any conclusions to be drawn for all code developed and implemented in industry.

Chapter Two: McCabe and Cyclomatic Complexity

It has been shown that about half of software product development time is spent in the testing phase [2], and that most of the dollars spent on a product are spent maintaining it [5]. Thomas McCabe [22] was looking for a quantitative measure that would indicate the difficulty of testing a program or module.

The metric McCabe developed, the cyclomatic complexity v(G), is based on the program control graph. A program control graph is a directed graph that represents the flow of control through a piece of code. It has unique entry and exit nodes. Each node corresponds to a sequential block of code that can only be entered at the first statement, exited at the last statement, and has no internal transfer of control. Each arc represents the flow of control between blocks. It is assumed that each node can be reached from the entry node, and that the exit node can be reached from any other node.

A formal definition of v(G) follows:

$$v(G) = e - n + (2 \times p)$$

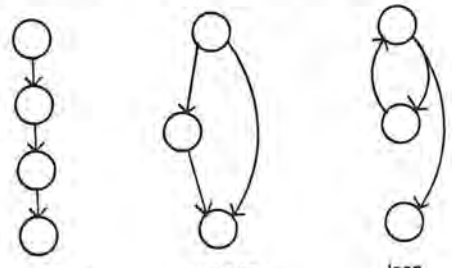
e: the number of edges in the graph

n: the number of nodes in the graph

p: the number of connected components

Berge's theorem [22] states that v(G) equals the maximum number of linearly independent paths in a single component graph G. V(g) measures the number of basic paths through G. Each path in a graph can be expressed as a combination of

basic paths. Below, v(G) is computed for the three basic control structures: sequential, decision, and loop.



McCabe showed that the cyclomatic complexity of a structured program equals the number of predicates in the code plus one. The cyclomatic complexity of a CASE statement adds the number of cases minus one to the computation. This is because any N-way CASE statement can be simulated by N-1 IF statements. N-way computed GO TO statements add N-1 to v(G). This is a much simpler method of computing v(G) than counting nodes, edges, and connected components in the program control graph. Hence, the calculation of v(G) can be made entirely from the syntactic characteristics of the program.

McCabe suggested that 10 is an acceptable upper bound on the level of complexity. Any module with a complexity greater than 10 should be rewritten, except possibly if it contains a CASE or computed GO TO statement.

Although McCabe does not cite any studies relating v(G) to bug rate, he analyzed 24 FORTRAN subroutines that were chosen because they were troublesome. v(G) for these subroutines ranged between 16 and 24. He reported a relationship between

v(G) and the reliability of the module. [22]

Myers [24] extended McCabe's work; he considered compound predicates. McCabe counted each predicate in computing v(G); Myers suggested counting each simple predicate or condition. "In practice compound predicates such as IF 'c1 AND c2' THEN are treated as contributing two to complexity since without the connective AND we would have IF c1 THEN IF c2 THEN which has two predicates". He states that "for this reason...it is more convenient to count the number of conditions instead of predicates when calculating complexity." [22]

Myers [24] does not cite any specific studies, but asserts that cyclomatic complexity "...is consistent with studies showing a high correlation between the number of decisions in a module and the module's complexity and error-proneness."

There are problems with McCabe's metric. It is based entirely upon the control flow graph. McCabe claims that two programs containing the same number of IF statements have the same level of complexity even if one has sequential IF statements while the other has several levels of nesting. The number of statements in the code has no impact on the measure of v(G). A 1000-line single-routine program with 10 IF statements and a 20-line single-routine program with 10 IF statements both have a cyclomatic complexity of 11. By defining the metric to be based enirely on the control flow graph, McCabe ignores such program characteristics as data structures, variable names,

comments, the algorithm used, and parenthesizing.

McCabe's cyclomatic complexity is of interest primarily because of its simplicity and ease of computation. His approach appeals strongly to intuition. It seems reasonable that a program module containing more paths through it would be harder to test and maintain than another containing a fewer number of paths. The control flow graph need not be drawn or analyzed. Several studies have found a high correlation between v(G) and the number of bugs in the code. The problems with computing cyclomatic complexity, such as how to handle a program with several subprograms, or counting predicates versus simple predicates, and the relationship between v(G) and the number of problems reported, will be discussed in chapter 5.

Chapter Three: Halstead and Software Science

Maurice Halstead [17] hypothesized that there are a set of invariant laws that may be applied to algorithms. These can be compared to the physical laws of science. His theories have become known as "software science". The laws of software science are said to hold true for all implementations of algorithms.

Halstead's premise was that regardless of the language used, implementations of algorithms have physical characteristics that can be measured and calculated from four basic program measurements:

n1: the number of distinct operators in a program
n2: the number of distinct operands in a program
N1: the total number of occurrences of operators in a program
N2: the total number of occurrences of operands in a program

The source code can be parsed into "tokens", which are divided between operators and operands. Operands include constants and variables. Operators are those tokens in the program that affect the ordering or value of the operands.

The laws that Halstead proposed based on n1, n2, N1, and N2, are as follows:

VOCABULARY (n) is defined as the total number of distinct tokens in a program:

$$n = n1 + n2 \quad \text{[equation 1]}$$

LENGTH (N) is defined as the total number of tokens in a program:

$$N = N1 + N2 \quad \text{[equation 2]}$$

Halstead's theory includes a length estimator that is strongly dependent on the distinct number of tokens in the program (n1 and n2). He suggested that N can be estimated using the following equation:

$$Nhat = [n1 \times \log2\ (n1)] + [n2 \times \log2\ (n2)] \quad \text{[equation 3]}$$

There is no mathematical derivation known for this equation [2,11,27].

VOLUME (V) is defined as the number of bits needed to represent the program. The equation is as follows:

$$V = N \times \log2\ (n) \quad \text{[equation 4]}$$

The derivation of program volume is straightforward. The vocabulary (n) of the program is defined as the number of distinct tokens in the implementation; log2 (n) is the number of bits needed to uniquely represent every element of the vocabulary in binary. The length (N) is defined as the number of total occurrences of all tokens in the source program. It follows that the volume of a program is the total number of occurrences of tokens in the program, multiplied by the number of bits necessary to represent the number of distinct tokens found in the program.

There are many ways to implement a given algorithm. Therefore, POTENTIAL VOLUME (V*) is defined as the most efficient implementation of an algorithm in a given language.

PROGRAM LEVEL (L) relates the volume (V) to the potential volume (V*) of a program implementation in the following way:

$$L = V^* / V \quad \text{[equation 5]}$$

L ranges between 0 and 1. A program level of one indicates that the implementation of the algorithm is the most efficient possible. The program level metric was intended to measure the effort incurred writing the program, error-proneness, and the understandability of the code. Halstead theorized that the program level should increase with the number of distinct operands (n2) and should decrease with both the number of occurrences of operands (N2) and the number of distinct operators (n1). [11] This led him to propose the PROGRAM LEVEL ESTIMATOR, Lhat:

$$\text{Lhat} = (2 / n1) \times (n2 / N2) \quad \text{[equation 6]}$$

As the program volume increases, Halstead theorized that the "difficulty" of the program implementation increases. [27] DIFFICULTY (D) is defined to be inversely proportional to the program level:

$$D = 1 / L \quad \text{or} \quad D = V / V^* \quad \text{[equation 7]}$$

Halstead theorized that the increase in program volume tends

to introduce poor programming practices such as redundant operands; or it may indicate lack of higher-level programming constructs. The result causes an increase in both program volume and difficulty.

LANGUAGE LEVEL (lambda) was developed to measure the power of a given language. Halstead hypothesized that the language level is constant over all algorithms implemented in a given language. It is defined as:

$$lambda = L \times V^* \quad \text{or,} \quad lambda = L \times L \times V \quad \text{[equation 8]}$$

Assuming the same language, or a constant language level, the program level decreases as the smallest possible program implementation increases. A more powerful language would require a smaller minimum volume to implement a given algorithm. As the potential volume decreases, the level of the program increases proportionately.

Halstead defined the EFFORT (E), as the effort needed to implement an algorithm:

$$E = V / L \quad \text{or} \quad E = [N \times \log2\ (n)] / L \quad \text{[equation 9]}$$

He hypothesized that the difficulty of program implementation, and hence the amount of effort needed to write the program, increases as the volume of the program increases, and decreases as the program level increases. Effort was more specifically defined as the number of mental discriminations made by the programmer during the implementation phase.

Theoretically, it can be used to predict the amount of time needed to implement an algorithm. Using Stroud's 18 mental discriminations per second [11], the estimated programming time is calculated as Ep = E / 18.

The laws that Halstead proposed were intended to be tested, somewhat like the physical laws of science. He suggested many relationships between the metrics themselves, and between the metrics and actual implementation of algorithms. His theories have not gone without criticism.

Shen, Conte, and Dunsmore [6] mentioned the following:

1) Halstead incorrectly inferred that because two sets of data were highly correlated, one can be used as a substitute for the other.

2) The sample sizes used in the experiments should have been larger for his statistical results to be valid.

3) The programs used in the study were too small, usually single modules of less than fifty statements, making it highly debatable whether results can be generalized to multi-module industrial programs.

4) Some experiments used college students. It is not clear whether these findings can be generalized to the programmer population as a whole.

Although the theoretical basis of software science may be

weak, many empirical studies have shown close relationships between his laws and both errors and programming effort.

Halstead [11] studied the correlation between N and Nhat using polished programs. He contended that Nhat was a good estimator of N, and that that any deviance between N and Nhat is due to impure code. If the code being analyzed is impure, Nhat will not be a good estimator of N. Impurites include cancelling of operators, ambiguous operands, synonymous operands, common subexpressions, unnecessary replacements, and unfactored expressions. Elshoff [9] pointed out that if it is true that Nhat best estimates N for well-structured programs, then Nhat is an easy measure of structuredness. He measured 154 programs written at General Motors Corporation. They were divided into two sets: 34 were written with "structured programming" techniques, the other 120 were written using "conventional techniques". The study encompassed measures of volume ranging over several orders of magnitude. He found that the correlation between the actual and estimated length was higher for the structured programs.

There is much evidence suggesting a strong relationship between the length and the length estimator. [27] This might seem surprising because it is easy to create a program where Nhat is not a good estimator of N. Nhat significantly overestimates N if each unique token is used only once or twice. Other studies using FORTRAN, COBOL, and PL/1 have indicated that Nhat is sensitive to program length; it estimates N best

for programs in the 2000 to 4000 lines of code range. It tends to be larger for smaller programs, and smaller for larger programs [27].

One of the most comprehensive studies to date was performed on 429 FORTRAN programs that were obtained from the Purdue University Library. The total sample contained a combined count of operators and operands exceeding 240,000. The lengths of the programs ranged between 7 and 10,611. The correlation between the actual and predicted lengths of the programs was 0.95. [4]

In a similar study [11] testing the relationship between L and Lhat, Halstead found that Lhat tended to overestimate the true value of L by 18%, but that the correlation coefficient was 0.90.

Shen, Conte, and Dunsmore [27] reported on a study relating D, the "difficulty" measure, to the number of bugs found in the module. Their study involved 197 PL/1 programs where error data was collected for two functionally equivalent programs, one having a larger measure of difficulty. The program with the larger measure of D had more errors associated with it. Another study from IBM, using 30 program modules, where errors were reported after software release showed D to be a good measure of relative error-proneness. [27]

The language level metric was intended to serve as a method of ranking the power of languages on a linear continuum. It

could be used to select a new language for a new application, or for testing its power potential. Halstead computed the language level for several languages: [12]

	lambda	std. dev.
PL/1	1.53	0.92
Algol	1.21	0.74
FORTRAN	1.14	0.81
CDC Assembly	0.88	0.42

Table 1

The language rankings appeal to our intuitive notion of how expressive each language is. The values lend only weak support to Halstead's theories because the standard standard deviations are so large. [27]

C. T. Bailey and W. L. Dingee [1] did not believe that the language level is constant for all algorithm implementations across a given language. They believed that Halstead's work was invalid because 1) his sample sizes were too small (sample sizes for his studies ranged between 7 and 34), and 2) his program volume range was too limited. (Program volumes in his studies ranged between 200 and 64000; the majority of the programs studied fell into the 200 to 2000 range.) They conducted studies of their own using larger sample sizes of ESS programs with a wider volume range. (ESS is a medium-level programming language providing program control of machine registers.) Their programs ranged in volume between 600 and 68000. The results from their studies showed that language level is high-

ly dependent upon program length. This became apparent when they grouped the modules by length. Bailey and Dingee believed that the results from their studies are better representative of real development environments.

They went one step further by analyzing the language level equation. At first glance, it appears to be dominated by the program level L (see equation 8). On closer inspection, however, we find that this is not always the case. The larger the program, the greater the number of distinct keywords. This is shown in equation 6 by an increase in n1. Intuitively, we can reason that for larger programs, the number of repetitions of operands (N2) will increase. As n1 and N2 increase with the volume of a program, the program level will tend to decrease (see equation 6). As program level decreases, program volume (V) will dominate the language level equation. In short, the language level will decrease with program volume.

Several studies have shown E to correlate well to the number of bugs found in a program [27,29]. Funami and Halstead [12] counted the number of bugs discovered during a 100 man-month software development project. The software science metrics were collected for the nine modules that constituted the project, and correlated to E. The correlation coefficient was 0.982, showing a strong relationship between the number of bugs found and effort. If, in fact, E is related to the number of mental discriminations a programmer makes, then the study also suggests a relationship between the number of men-

tal discriminations and the number of bugs.

Another study [13] looked at 46 programs written in FORTRAN, COBOL, Algol, and PL/1. Each was re-written using good programming techniques, which resulted in 46 program pairs. Effort was calculated for each program implementation. It decreased in 40 out of 46 cases when the programs were rewritten. The conclusion of the study was that effort is related to good programming practices.

In a Purdue University study, [28] 48 programmers were asked to study eight versions of the same program for a fixed time. Subjects then answered a 20-question quiz, designed to measure comprehension. Those who studied programs with the lowest effort measure had the highest quiz scores. This implies a relationship between effort and comprehensibility.

J. D. Gould [16] studied the effort metric related to the ability of a programmer finding a seeded bug in a program. He asked ten experienced programmers to debug the same 12 FORTRAN listings. Each subject was given 45 minutes to find one non-syntactical bug. If he incorrectly identified the problem, he made an "error". For each listing, Gould determined the average number of "errors" made by the programmers locating the bug. The mean number of "errors" was correlated to E; the correlation coefficient was 0.78.

One of the criticisms of E is that it measures the number of mental discriminations made by a "fluent, concentrating pro-

grammer". It does not consider the experience level of the programmer, either in the implementation language, or in programming itself.

Other criticisms center around the theory behind the derivation of Halstead's theories [7]. For example, effort equals volume divided by program level. To derive the numerator of the equation, Halstead assumed that the human mind uses a binary search to locate each token used in the algorithm implementation. That implies that a programmer keeps a sorted list of tokens in his mind. This is questionable; no research has been done to substantiate this theory. Studies have investigated how a person searches through memory to find a fact. It is currently thought is that the search is sequential, probably not exhaustive, but terminated when the information sought is found. The complaint is that Halstead tried to apply computer science theory to the human mind.

Halstead confused short-term memory and long-term memory. [7] Can we assume that a programmer searches short-term, rather than long-term memory for the information he needs to implement a program? While short-term memory has been studied, long-term memory is still an active area of research. These problems make the theoretical derivation of E hard to accept.

Not enough studies have been performed to substantiate Halstead's theories. [1] Although empirical studies have shown some positive results, there are too many underlying

problems with Halstead's assumptions and the theories that he proposed. While continued empirical investigations may be worthwhile, including testing across a broad range of languages and sample sizes, real progress will not be made until his basic theories are reformulated. [19]

We will compute Halstead's metrics several different ways and compare our results with those in recently published studies. The problems with Halstead's laws, and their computation, such as how pairs of operators (2-word key words, parenthesis, brackets), and global versus local variables should be counted, along with the relationship between the metrics and the number of bugs reported will be discussed in chapter 5.

Chapter Four: The Field Study

The code obtained for this study was from a software project developed by a large electronics firm in the Pacific Northwest, written in C to run on the UNIX (tm) operating system. The modules of code were broken down according to user functionality; some examples included I/O, graphics, mass storage, and math functions.

The study began approximately three months prior to the start of the testing phase, while the code was still under development. This left a couple of months to plan and begin implementing the tools that would be used to obtain and analyze the data (figure 1).

The software complexity metrics applied in this study assume that 1) modules receive the same amount of initial testing or debugging prior to entering the testing phase, and 2) all modules receive the same amount of testing while error data is being collected. This is consistent with the requirements suggested by Ottenstein when performing this type of study. [25] Modules not meeting both criteria were not considered in this study.

Collecting the Error Data

Prior to this project, the electronics firm handled bug reporting manually. This can be detrimental to the data-gathering process because it is cumbersome and difficult to

tabulate the data. Valid information can easily be lost.

To obtain valid data, bug reporting had to be practically effortless. An automated system that would lend itself well to inspection by both the development and test engineers was established. It allowed anyone to easily peruse the database of bugs reported.

Time was of the essence, so the best solution was to utilize a tool already in existence. It was decided to use UNIX notes files to maintain bug reports. They can be organized and titled however the programmer wishes. The basic structure consists of base notes and responses. (see figure 2) At the time a base note is entered into the system, a title directly associated with the base note itself is entered. An index page can be displayed listing the titles of each base note entered.

Next on the priority list was to define what error data needed to be collected. Questions were outlined: 1) Is the problem a new bug, or is it one that was previously reported? 2) Is the problem a legitimate bug, or did the user misunderstand how the product was designed to function? 3) Does the problem reported need to be fixed so the product fits the external requirements? Or, is it an enhancement request, that could be either ignored or satisfied later. 4) If the problem reported requires a change to the code (definition of a bug), how severely would it have impacted the user? [The definition of

"severity" was established by the corporation, and could not be changed for the purpose of this study.] 5) Which modules were changed to fix the problem? 6) Where in the product life-cycle was the problem introduced? Was it something that was overlooked in the requirements, design, or coding phase?

Applying the base note concept to bug tracking, the bugs were posted to the notes file as individual base notes. Each base note contained the following information: a unique number identifying the bug report, the name of the engineer reporting the problem, the date the problem was reported, a description of the system being used when the problem occurred, the configuration of that system, the version of the software being used, and a detailed description of the problem. Figure 3 is a copy of the problem report form used.

Bugs that were fixed had a response attached to the base note. The information in the response that was utilized in this study is as follows: the name of the engineer fixing the problem (or the name of the engineer posting the response), the names of the modules affected by the change, and the severity of the problem, categorized as critical, severe, moderate, or trivial. Figure 4 is a copy of the problem resolution report form, which includes the corporate definitions for each level of severity.

To enhance the clarity and ease of the reporting scheme, several notes files were created to log the bugs found in the

project. Basically, each functional area was given a notes file: mass storage, I/O, etc. In addition, the catch-all "general" was created for errors that did not fit in any other category. The choice to divide the bugs into categories by functionality was good for the tester and design engineer. The tester could easily peruse the file for a current bug to see if it had been reported and/or fixed. The design engineer could quickly browse the file for bugs reported against specific modules of code.

The error data collected is largely self-explanatory. (See figure 4) However, there are a couple of interesting things to note. Choice number 4, "duplicate of reported problem", under "PROBLEM CLASSIFICATION" may encompass two types of problems. The bug reported may be identical to one already in the tracking system, or it may be that one bug fix corrects two seemingly different problems. If two bugs are reported, and the engineer makes one fix which corrects them both, he was instructed to resolve one bug appropriately (answering all of the questions), and the other as a duplicate of an already-reported bug. This is important, because without before-hand instruction to the engineers, it is quite conceivable that the same bug resolution information could be posted several times, which would cause inflated error data.

Once the data was posted to the notes files, a simple way to extract all pertinent information was needed. The solution was to utilize the UNIX tool LEX, which allows the programmer

to define extraction rules based on simple pattern-matching. LEX takes the set of rules and creates a C-program. The program is then compiled and run against the data in the notes files. The output was a simple ASCII data file. Each line of the file represented one bug report. This information was input into a dBASE II database structure and a dBASE II program was written to create a report summarizing the information. The report included all of the information in figure 4, tallied by module. This is the data that was ultimately correlated to the complexity data.

Collecting the Complexity Data

The correlation of error data to any complexity metric required that there be an extraction technique to obtain the appropriate program characteristics. The program characteristics were obtained using a method known as the Reduced Form, developed by Harrison and Cook. [20] It includes information for each routine in the code system. Essentially, it includes five parts:

1) An identification line which consists of the name of the subprogram.

2) A declaration table, which lists the number of times each type of declaration was used in the subprogram.

3) An operand table which lists an alias for each unique

string, constant and variable, and the number of times the item was used.

4) An operator table which lists the operators used and the function calls made, plus the number of times each was used.

5) A length line, which indicates the number of source lines and the number of non-commentary source lines in the subprogram.

A program was written to take the list of tokens as input, and compute the metrics for the purpose of this study.

Chapter Five: Analysis of Data

This study repeated software complexity metric studies found in recent publications. The data obtained was analyzed and compared with the published results. The uniqueness centered around the use of the C programming language. No previous publications studied the relationship between Halstead's laws or McCabe's metrics and C, and most of the studies used either FORTRAN or COBOL programs from a text book or controlled experimentation, rather than code from industry.

Before discussing any results, a short description of bug rates is needed. The bug rates in each module can be calculated in several different ways. Recall that there are four categories of bugs: critical, serious, moderate, and trivial. It is possible to perform statistical analysis using the total number of bugs found in each module, or to group them by category. Rather than choose one particular method of grouping the bugs, we used several different groupings. We consider the total number of bugs, the total number of non-trivial bugs, the number of critical and serious bugs taken together, and then each category by itself.

The table headings require an explanation. The following abbreviations are used: "C" for critical, "S" for severe, "M" for moderate, and "T" for trivial bugs. (For a review of the bug classifications, refer to figure 4.) Table 1 shows the total number of errors reported against each of the thirty modules by severity:

module	C	S	M	T	total
mod1	1	3	1	2	7
mod2	3	6	20	6	35
mod3	5	9	8	5	27
mod4	0	1	0	0	1
mod5	1	2	1	6	10
mod6	0	1	3	2	6
mod7	0	0	0	0	0
mod8	0	0	1	0	1
mod9	0	1	1	0	2
mod10	3	8	5	5	21
mod11	1	3	6	3	13
mod12	1	4	11	5	21
mod13	3	3	6	1	13
mod14	0	1	0	0	1
mod15	0	3	0	0	3
mod16	1	0	2	0	3
mod17	3	5	6	1	15
mod18	4	8	11	3	26
mod19	0	0	0	0	0
mod20	0	0	0	0	0
mod21	0	0	0	0	0
mod22	7	17	18	2	44
mod23	0	3	2	0	5
mod24	0	0	0	0	0
mod25	3	2	2	2	9
mod26	1	2	0	0	3
mod27	0	1	8	5	14
mod28	0	0	6	10	16
mod29	1	2	0	0	3
mod30	0	1	3	0	4

Table 1

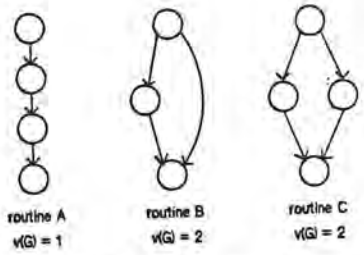
McCabe's Cyclomatic Complexity

It is not clear how to compute v(G) for a module with several routines. Should, as McCabe suggested, v(G) be the sum of the v(G)'s for each routine within a module, or should routine boundaries be ignored?

Partial v(G)s can be computed for each routine, and then summed across the module to compute the module's cyclomatic complexity. The alternative is to compute the module's v(G)

without regard to routine boundaries. These two alternative approaches are demonstrated below.

Consider a module with a main routine that makes calls to two procedures, but has sequential code otherwise. The program control graph looks like diagram A below. Assume that the two procedures called can be represented by control graphs B and C:



Using the first counting method, v(G) is computed for each routine: v(G)[A] = 1, v(G)[B] = 2, v(G)[C] = 2. v(G) = v(G)[A]+v(G)[B]+v(G)[C]. v(G)=5. Computing v(G) as specified in the second method ignores routine boundaries. v(G)=3. (There are a total of two decisions in the module, add one, and v(G) equals 3.)

If v(G) is calculated for each routine in the module, and the partial v(G)s are added together, the cyclomatic complexity of the module may be larger than if it had been computed over the entire module, without consideration to routine boundaries. This is because one is added to each partial v(G) computation. When the partial v(G)s are added together, the cyclomatic complexity is increased by one for each routine included in the summation.

Myers [24] extended McCabe's original work for complex condi-

tions. Should v(G) be computed as the number of predicates in the code, or as the number of simple predicates and conditions? Consider a module with entirely straight-line code except for the following line: "IF a<b AND b>c THEN...". If v(G) is defined as the number of decisions plus one, then the cyclomatic complexity equals 2. If, on the other hand, it is defined to include the number of conditions, v(G) equals 3.

Four different schemes for counting complex conditionals were studied and computed from the program modules:

1) The number of predicates in the module plus one. Each of the following keywords occurring in a module added one to the complexity count: WHILE, IF-THEN, IF-THEN-ELSE, and FOR. N-way CASE statements increased the count by N-1.

2) The number of simple predicates or conditions, plus one, where the predicates are as specified in 1) above. (CASE statements were also handled as described above.)

3) A "partial" v(G) was computed for each routine in the module, as described in 1) above. A summation was taken to obtain the final v(G) value.

4) "Partial" v(G)'s were computed for each routine as described in 2) above. A summation was taken to obtain the final v(G) value.

The first two methods give no consideration to routine boundaries. The last two calculate v(G) for each routine, and use

a summation to compute the cyclomatic complexity.

McCabe did not specifically state any studies relating v(G) to bug rate, although he did suggest that a simple ranking exists between the cyclomatic complexity metric and the error-proneness of 24 FORTRAN subroutines. [24] We compared the four counting methods with various bug groupings. The correlations are given in Table 2.

metric	CSMT	CSM	CS	C	S	M	T
v1(G)	.763	.771	.839	.755	.854	.599	.330
v2(G)	.776	.784	.840	.766	.851	.623	.340
v3(G)	.745	.752	.828	.751	.840	.574	.329
v4(G)	.767	.773	.835	.763	.844	.607	.346

Table 2

The correlation coefficients suggest a strong relationship between the bug rate and v(G). The relationship between v(G) and the moderate or trivial bugs is not strong. The relationship between the bug rate and the critical and serious bugs taken together, or the serious bugs alone is very strong.

Does any one method of counting cyclomatic complexity provide additional information which is helpful in establishing a relationship between the bug rate and the metric? As a statistical test, correlation coefficients were computed between the four metrics themselves with the following results:

metric	v2(G)	v3(G)	v4(G)
v1(G)	.995	.996	.993
v2(G)	.---	.993	.998
v3(G)	.---	.---	.996

Table 3

The high correlation coefficients suggest that the four variations of computing the cyclomatic complexity are very strongly related. Table 2 shows that the correlations for all of the counting methods with the bug rate are very nearly the same. Table 3 shows that a high correlation exists between all methods of computation. Together, the tables indicate that there is no difference between the four methods of computation, and therefore, if v(G) is to be computed, the easiest method should be used.

Halstead's Software Science

The most basic problem with Halstead's software science computations [26] is the ambiguity of the counting rules when deriving n1, n2, N1, and N2. He relates his method of counting to implementations of algorithms. For example, declarations have nothing to do with the algorithm itself, and hence do not impact the calculations. He worked with FORTRAN, which has implicit declarations, so in his experimentation, this question was not serious.

Another question concerns counting local versus global variables. If a variable is declared global and then used locally within a routine, or set of routines, is it counted as the same variable, or as a distinctly different variable?

It is not clear if counting should be done by functional

module or by individual routine. Should n1, n2, N1, and N2 be calculated for each routine in each module? Potentially, each routine uses the same operators; should they be counted as unique for each routine? How does one combine measures to obtain one set of values for each module?

Even more elementary problems exist in defining an unambiguous counting strategy. How are operators that occur in pairs, such as IF-THEN, {}, and () counted? Is each half counted as a distinct operator, or are they grouped together as one? Should delimiters be counted, such as the semi-colon?

Elshoff investigated different ways of computing Halstead's laws [8], varying the computations by addressing some of the questions stated above. The conclusions from his study did not include a statement about which computation method was best, but only that the results from a study may depend upon the counting method utilized.

To solve the ambiguity problem for the purpose of this study, several counting methods were defined. The list of rules followed for all counting strategies are:

1) Count only executable statements. This counting rule was used primarily because Halstead based his theories on implementations of algorithms. No consideration was given to language-dependent overhead of implementing the algorithm in a given language. (i.e. declaration of variables.)

2) Any pairs of symbols, such as parenthesis, are counted together as one. They function as a single operator.

3) Count the semi-colon as a unique operator.

4) Count the tokens (operators and operands) the same in any context. For example, the parenthesis pair in a decision statement is no different from one used in an arithmetic calculation.

5) In the GO TO <label> statement, count the GO TO as an operator, and the <label> as an operand. This is different from Halstead's original treatment of each GO TO <label>. He counted each occurrence of GO TO <label> as a unique operator.

6) Function calls are counted as operators.

Several variations of counting methods were used in this study:

1) n1 and n2 were calculated by the summation of the n1 and n2 values calculated by routine, over the entire module. (see figure 5) n1 includes the distinct number of functions called, keywords, and arithmetic or logical operators. n2 includes all global and local variables, labels, and constants. N1 and N2 were calculated by counting the total number of occurrences of operators and operands, respectively, over the entire module.

2) Method 2 is similar to method 1 except that it does not

consider routine boundaries. If a variable is global, it is counted only once in n2. (In method 1, a global variable occurring in routines "a", "b", and "c" would add one to "n2a", "n2b", and "n2c" (see figure 1). Ultimately, 3 would be added to n2.) Operators are handled similarly. A minus sign adds one to n1, regardless of how many routines it occurs in. Notice that N1 and N2 are identical for methods 1 and 2. They represent the total number of occurrences of tokens, regardless of how "distinctness" is defined.

3) Method 3 is an attempt to remove one variable from the calculations. Prior to any coding, if the number of operators used in the program is estimated, the only unknown needed to calculate Nhat is the number of distinct operands that will be used in the code. After the design phase of the project, an estimation of the number of local and global variables, labels, and constants needed can be made Method 3 tests this theory by calculating n2, as per the description in method 2; it uses the constant 40 for n1. Forty was an estimate obtained by counting the number of distinct operators used in random samples of C-code.

Halstead [11] studied the relationship between his length estimator (Nhat) and his length metric (N). Using polished code, he asserted that Nhat is a good estimator of N, providing the code is "pure". Following are the results of correlating N with Nhat for each of the three counting methods:

metric	N1	N2	N3
Nhat	.976	.971	.951

Table 4

The results from this study support the assertion that there is a high correlation between Halstead's length and his length estimator, no matter which counting method is used.

To test for a relationship between the length or length estimator and the bug rate, the following correlation coefficients were computed:

metric	CSMT	CSM	CS	C	S	M	T
N1	.761	.750	.796	.749	.794	.604	.407
N2	.761	.750	.796	.749	.794	.604	.407
N3	.761	.750	.796	.749	.794	.604	.407
Nhat1	.717	.703	.779	.719	.784	.532	.398
Nhat2	.702	.677	.736	.692	.734	.527	.435
Nhat3	.698	.675	.743	.685	.749	.516	.424

Table 5

A strong relationship exists between both the length and bug rates, and the length estimator and the bug rates. There is a strong relationship between the metrics and the more significant bugs; the metrics are not strongly related to the trivial or moderate bugs taken by themselves. The relationship between all variations of computing length N and the bug rates are identical. This is to be expected from the counting rules. The number of tokens (N1 and N2) in the code is the same for each counting method.

Correlation coefficients were calculated between Halstead's volume and the bug rate. Table 6 shows the correlation coefficients.

metric	total bugs	CSM	CS	C	S	M	T
V1	.763	.753	.803	.756	.802	.601	.406
V2	.763	.750	.799	.754	.796	.601	.410
V3	.763	.752	.802	.754	.800	.601	.601

Table 6

Although no specific studies were found relating volume to the bug rate, it is interesting to note that these include the highest correlations found in our study. The critical and serious bugs taken together have the strongest relationship to volume. The moderate and trivial bug count is not related to the volume metric.

Shen, Conte, and Dunsmore [27] studied the relationship between difficulty and the bug rate. Their studies ranked modules according to their error-proneness, and observed whether the difficulty measure increased accordingly. Correlation coefficients in Table 7 were computed to study the relationship between the three difficulty measures and the bug rate.

metric	CSMT	CSM	CS	C	S	M	T
D1	.724	.712	.769	.740	.759	.559	.397
D2	.575	.563	.556	.591	.522	.495	.321
D3	.268	.280	.238	.249	.225	.287	.042

Table 7

The counting method matters when computing the difficulty. The first method shows the strongest relationship to the bug rate. The other two methods do not indicate any relationship at all.

Halstead proposed a language level metric that was intended to be used as a mechanism to rank languages on a linear continuum, according to their power. He computed the language level for several languages; C was not among those studied. The language level was computed three times, using each of the proposed counting methods. The results are as follows:

method	average	standard deviation
Method 1	0.49	1.30
Method 2	0.97	1.05
Method 3	9.08	12.17

Table 8

The standard deviations computed are quite large. This supports the Dingee and Bailey [1] contention that language level is not constant for all implementations of algorithms in a given language. Shen, et. al. [27] would contend that the results from this study lend weak support to Halstead's hypothesis.

Of all Halstead's metrics, effort (E) is most often correlated to bug rate. The correlation coefficients suggest a strong relationship between effort and bug rate. (Table 9)

metric	CSMT	CSM	CS	C	S	M	T
E1	.722	.712	.778	.742	.772	.551	.389
E2	.696	.689	.691	.686	.673	.596	.367
E3	.760	.761	.784	.753	.775	.636	.364

Table 9

The strongest relationships exist between effort and the more significant bugs (critical and serious). Strong relationships do not exist between the moderate or trivial bugs and the effort.

There is a strong relationship between the counting methods as shown in the following table of correlation coefficients.

metric	E2	E3
E1	.912	.922
E2	.---	.968

Table 10

Comparative Study Between Complexity Metrics

Lines of code is a simple metric that has performed well in many studies. Therefore, we investigated its relationship to bug rate. Below is a table of correlation coefficients that show this relationship. Lines of code has been computed two ways. "LOCw" is the number of lines of code including comments, and "LOCwo" is the number of lines of code excluding comments. For the purpose of this study, if a line of code has comments on it, that line is counted in both the measures. The only line of code that is not considered in the "LOCwo" count is source code that only has a comment on it.

metric	CSMT	CSM	CS	C	S	M	T
LOCw	.787	.781	.770	.716	.773	.687	.424
LOCwo	.826	.819	.830	.772	.833	.699	.439

Table 11

Lines of code, counted with and without the comments, correlated well to the number of bugs in the module. Lines of code without comments has among the strongest relationship to bug rate when compared with any of Halstead's or McCabe's metrics.

Going one step further, the relationship between all of the metrics, computed using method one was investigated. Table 12 shows the correlation coefficients:

metric	LOCwo	v(G)	N	Nhat	V	D	E
LOCw	.982	.896	.907	.883	.909	.890	.975
LOCwo	.---	.818	.903	.799	.832	.820	.810
v(G)	.---	.---	.973	.960	.972	.950	.911
N	.---	.---	.---	.976	.998	.989	.948
Nhat	.---	.---	.---	.---	.983	.974	.959
V	.---	.---	.---	.---	.---	.990	.962
D	.---	.---	.---	.---	.---	.---	.964

Table 12

There is a strong relationship between all three metrics: lines of code, Halstead's laws, and McCabe's cyclomatic complexity.

Chapter Six: Conclusions

The counting rules for software science and cyclomatic complexity seem obvious. This paper shows that there are many choices to make. McCabe's cyclomatic complexity and Halstead's software metrics are each computed different ways. The relationship between these metrics and the bugs reported against modules of code are analyzed in the body of this paper.

All of the metrics studied in this report correlate well to the number of bugs reported against the code: lines of code and variations on both McCabe's cyclomatic complexity and Halstead's laws. The relationship between the metrics and the bugs reported is strongest for the severe and critical bugs.

Comparing the correlations between the variations of the metrics with the bug rates, we found that there was very little difference. No one counting method stood out as being more strongly related to the bug rate than any other.

This paper shows that for the metrics studied, lines of code and cyclomatic complexity have the strongest relationship to bug rate. Since lines of code is so easy to compute, we suggest that it should be used over the other metrics as a bug rate indicator.

Data Collection:

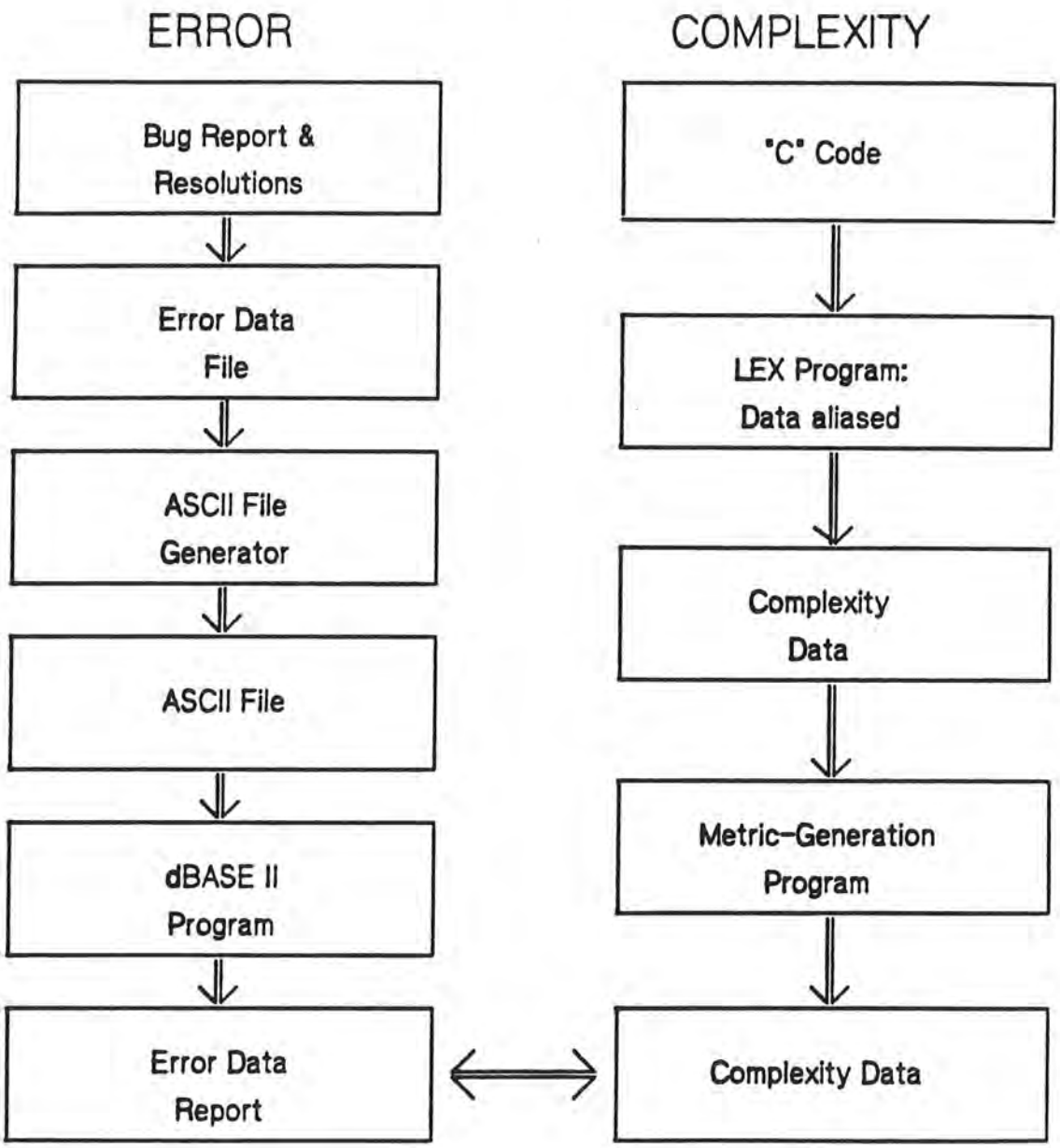


figure 1

Diagram of a Notesfile

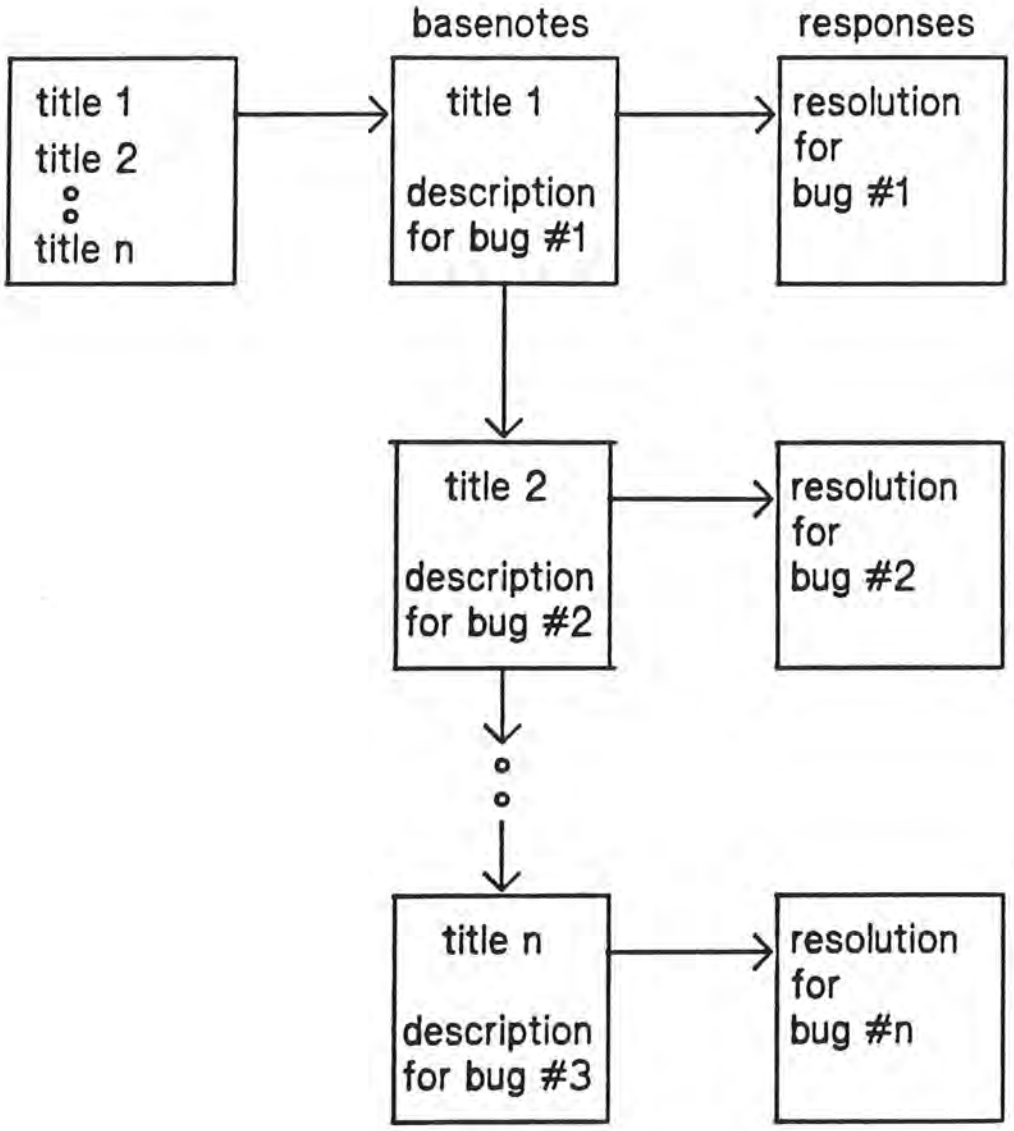


figure 2

PROBLEM REPORT FORM

<< PROBLEM ID >>

Unique identification for the problem report.
Set automatically when the problem is reported.

<< REPORT SUBMITTER >>

Identification of the person who discovers and reports the problem.

<< REPORT DATE >>

Date that the problem report is posted to the system.

<< SYSTEM IDENTIFICATION >>

Hardware identification. (Used to weed out problems due to a flakey hardware system.

<< SYSTEM CONFIGURATION >>

System configuration was being used when the problem was discovered: includes plugins, peripherals, and the amount of memory in the system.

<< SOFTWARE VERSION >>

Version of the software that was being used when the problem was discovered.

<< PROBLEM DESCRIPTION >>

Description of the problem in sufficient detail so that an attempt could be made at duplication.

figure 3

PROBLEM RESOLUTION REPORT FORM

<< LAB ENGINEER ASSIGNED >>

Identification of lab engineer assigned to resolve the problem.

<< PROBLEM CLASSIFICATION >>

Identification of problem type by selecting an item from the following list:

0 - new problem (requirements)
1 - new problem (design)
2 - new problem (code)
3 - side-effect of problem fix
4 - duplicate of reported problem (problem id _____)
5 - documentation problem
6 - enhancement request
7 - could not duplicate
8 - not a problem (user misunderstanding)

<< PROBLEM SEVERITY >>

Indication of the severity of the problem taking into account the destructiveness of the problem and its probability of occurrance.

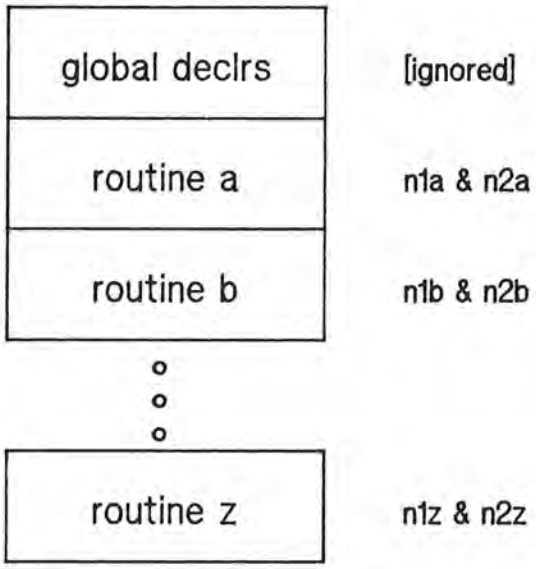
1 - critical (severe impact, occurs in normal use)
2 - serious (degrades operation, occurs in normal use)
3 - moderate (degrades operation, special circumstances)
4 - trivial (minor impact, unlikely to occur)

<< AFFECTED MODULE(S) >>

Identification of all modules that require a change to fix problem.

figure 4

Method One



$n1 = n1a+n1b+...+n1z \quad n2 = n2a+n2b+...+n2z$

figure 5

BIBLIOGRAPHY

[1] Bailey, C. T. and W. L. Dingee, "A Software Study Using Halstead Metrics," ACM Sigmetrics, pages 189-197, 1981.

[2] Boehm,B.W. "Software and it's Impact: A Quantitative Assessment," DATAMATION, Vol. 19, pp. 48-49, May 1983.

[3] Brooks, Ruven E., "Studing Programmer Behavior Experimentally: The Problems of Proper Methodology," COMMUNICATIONS OF THE ACM, Vol. 23, No. 4.

[4] Bulut, N., M. Halstead, and R. Bayer, "The Experimental Verification of a Structural Property of FORTRAN Programs," in Proceedings of the ACM Annual Conferernce, 1974, New York.

[5] Cammack, W.B. and H.J. Rogers, "Improving the Programming Process," IBM TECHNICAL REPORT TR 00.2483.

[6] Conte, Samuel D., "The Software Science Language Level Metric," CSD TR 373, September 1981.

[7] Coulter, Neal S. "Software Science and Cognitive Psychology," IEEE Transactions on Software Engineering, Vol. SE-9, No. 2 March 1983.

[8] Elshoff, James L. "An Investigation into the Effects of the Counting Method Used on Software Science Measurements".

[9] Elshoff, J. L. "Measuring Commercial PL/I Programs Using Halstead's Criteria," ACM SIGPLAN Notices, May 1976.

[10] Elshoff, James L. and Michael Marcotti, "On the Use of the Cyclomatic Number to Measure Program Complexity," SIGPLAN Notices, December 1978.

[11] Fitzsimmonds, Ann and Tom Love, "A Review and Evaluation of Software Science," COMPUTING SURVEYS, Vol. 10, No.1, March 1978.

[12] Funami Y. and Halstead, M. H. "A Software Physics Analysis of Akiyama's Debugging Data", CSD-TR144, Purdue University, Lafayette, Indiana, May 1975.

[13] Gordon, Ronald D. "A Measure of Mental Effort Related to Program Clarity," PhD Thesis, Department of Computer Science, Purdue University, Lafayette, Indiana, 1977.

[14] Gordon, Ronald D. "Measuring Improvements in Program Clarity", IEEE Transactions on Software Engineering, Vol. SE-5, No. 2, March 1979.

[15] Gordon, R. D. and M. H. Halstead, "An Experiment Comparing FORTRAN Programming Times with the Software Physics Hypothesis," in 1976 Fall Joint Computer Conference, AFIPS Conference Proceedings, Vol. 45, Montvale, New Jersey: AFIPS Press, 1976, pages 936-937.

[16] Gould, J. D. "Some Psychological Evidence on How People Debug Computer Programs," International Journal of Man-Machine Studies", 1975.

[17] Halstead, Maurice H. ELEMENTS OF SOFTWARE SCIENCE, Elsevier North- Holland. Inc., New York, 1977.

[18] Halstead, Maurice H., "Natural Laws Controlling Algorithm Structure?," SIGPLAN Notices, February 1972.

[19] Hansen, Wilfred J., "Measurement of Program Complexity by the Pair", SIGPLAN Notices, March 1978.

[20] Harrison, Warren, and Curtis Cook. "A Method of Sharing Industrial Software Complexity Data," SIGPLAN Notices, Volume 20 Number 2, February 1985.

[21] Lassez, J. L., D. van der Knijff, J. Shepherd, and C. Lassez, "A Critical Examination of Software Science," The Journal of Systems and Software 2, 1981.

[22] McCabe, Thomas, "A Complexity Measure", IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, Vol. SE-2, No. 4. December 1976.

[23] Myers, Glenford J. THE ART OF SOFTWARE TESTING, John Whiley, 1979.

[24] Myers, Glenford J. "An Extension to the Cyclomatic Measure of Program Complexity", SIGPLAN Notices, October 1977.

[25] Ottenstein, Linda M. "Quantitative Estimates of Debugging Requirements", IEEE Transactions on Software Engineering, Vol. SE-5, No. 5, September 1979.

[26] Salt, Norman F. "Defining Software Science Counting Strategies", Department of Measurement, Evaluation and Computer Applications, The Ontario Institute for Studies in Education, SIGPLAN Notices, March 1982.

[27] Shen, Vincent Y, Samuel D Conte, and H. E. Dunsmore, "Software Science Revisited: A Critical Analysis of the Theory and its Empirical Support," Transactions on Software Engineering, Vol. SE-9, No. 2, March 1983.

[28] Woodfield, S.N., H.E. Dunsmore, V. Y. Shen. "The Effect of Modularization and comments on Program Comprehension ", Proceedings of the 5th International Conference of Software Engineers, San Diego, CA., March 1981, pp. 215-223.

[29] Zislis, P. M., "An Experiment in Algorithm Implementation," Department of Computer Science, Purdue University, West Lafayette, Indiana, Technical Report 96, June 1973.