84
3

The Relationships Among Commenting Style, Software Complexity Metrics, and Software Maintainability

by

Wilson K. Gibbins

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Masters of Science

in

Computer Science and Applications

APPROVED:

Sallie Henry

Dr. Sallie M. Henry (Chairperson)

James D. Arthur

Dr. James D. Arthur

Osman Balci

Dr. Osman Balci

Blacksburg, Virginia

The Relationships Among Commenting Style, Software Complexity Metrics, and Software Maintainability

by

Wilson K. Gibbins

Dr. Sallie M. Henry (Chairperson)

Computer Science and Applications

(ABSTRACT)

Programmers are encouraged to comment their source programs, yet the value of the comments is not easily verified. In this study, the relationships between comment quantity and software metrics are assessed to determine whether programmers increase comment quantity in complex modules. In addition, comment quantity and software metrics are related to software maintenance data. It was found that software complexity, as measured by software metrics, accounts for a substantial portion of the variance in comment quantity. Additionally, comment quantity has no statistically significant relationship to software maintainability for the task studied.

Acknowledgements

The longer one takes to get a degree, the more debts of gratitude one accumulates. First, a heartfelt thanks goes to my adviser, Dr. Sallie Henry. She had the difficult task of finding a compromise between giving me an appropriate amount of guidance without actually doing the work for me. In addition, she has given me inspiration and confidence at critical times during my academic career. A special thanks goes to Dr. Arthur, both for his contagious enthusiasm and valuable advice. I thank Dr. Balci, for agreeing to be on my committee at the last moment, in spite of the fact that he was familiar with the usual quality of my work.

In addition to my committee members, there were quite a few other individuals without which I would never have completed my research. Calvin Selig was instrumental in helping me successfully navigate the worlds of the software metrics system and student data collection. Geoff Vining and Cathy Underwood of the statistics lab were essential in allowing me to make some sense of the volumes of data that were produced. The staff of the Computer Science Department has always been helpful with administrative tasks during my extended tenure at Virginia Tech.

Finally, my family deserves more than I can ever repay for the support they have given me during these four years. My children, Karen and Jenny, have had to endure being without

their father for a few too many nights. My parents, Neil and Helen Gibbins, and my in-laws, Gilbert and Eloise Myers, have volunteered more financial and child-care support than I would ever have the courage to request. Most importantly, my wife, Ellen Silva, has been important in helping me in the presentation of ideas and in giving me the emotional support necessary for the ups and downs of academic life.

Table of Contents

List of Illustrations

List of Tables

1.0 Chapter 1: Introduction

1.1 Introduction

Most computer languages have some method of adding comments to source programs. The programmer has considerable freedom in the use of comments since the comments are ignored by language translators. This allows a wide variety of commenting styles. At one extreme, there is the behavior of the hypothetical programmer described by Feinberg [FEID 84] who did not comment his programs at all since he received no credit for comments from the automated performance evaluation system. At the other end of the spectrum is the attitude that "too much documentation is impossible" noted by Sheppard et al. [SHES 79].

There are three types of support for the use of comments. First, there are rational arguments in favor of commenting programs. It is not difficult to generate reasons to explain why commented programs are worth more than uncommented ones. Second, there are a number of computer science and data processing authorities who are advocates of commenting programs. There are also some who view commenting with disfavor. Third, there are empirical studies that have tested the value of comments. These studies have been controlled exper-

iments using small programs. More detailed descriptions of these three types of support follow.

1.2 Rational Arguments for Commenting

The generally accepted purpose of comments is to make programs easier to debug and maintain. This can be done by improving the readability of the source code or by directing the programmer to relevant parts of the program, such as by listing hardware assumptions or the location of system dependent code. Since testing and debugging comprise 1/3 to 1/2 of project development time [YOUE 75], and program maintenance is estimated to be as much as 60% of the total software budget [CONS 86], anything that makes software easier to debug or maintain has the potential to produce large economic returns. The supposed utility of comments is based on the following assumptions:

1. Programmers can determine when comments are needed and when they are not. Therefore, software developers can insert comments selectively to add information that helps debugging or maintenance without cluttering programs with excessive comments that distract maintenance programmers from more important aspects of the program.

2. Maintenance programmers are able to use the comments to improve their performance, whether correcting errors or making enhancements. These maintenance programmers have at least a rough idea of when the program comments should be read and when they should be ignored.

Neither assumption can be made as given. Assumption 1 is based on the programmer knowing where in the program there may be bugs or what parts of the program might need to be

changed. In this sense, comments are a sign of weakness, an uncomfortable idea for most programmers. When a programmer feels that a part of the program will probably be changed in the future, a better approach might be to rewrite the code to make it more flexible rather than merely noting the problem. Of course this is not always practical, as some changes cannot be foreseen. However, this begs the question of how the original programmer is to write comments to aid in the process of making unforeseen changes.

Assumption 2 is also questionable. Programmers generally do not leave bugs in their programs on purpose. Therefore, when errors occur, it is usually a sign that the program does not do what the original designer or programmer intended. One obvious approach for the maintenance programmer is to concentrate on what the program code does, rather than what the original programmer intended for it to do [ARNK 88]. Comments are a better reflection of the latter than the former.

1.3 Opinions of Authorities

Programming authorities are divided on commenting style and on the effectiveness of comments. The recommendations range from enthusiastic support for comments to skeptical disrespect.

Perhaps the most ardent proponent of comments is Yourdon [YOUE 75] who says

> In my opinion, there is nothing in the programming field more despicable than an uncommented program. A programmer can be forgiven many sins and flights of fancy, ...; however, no programmer, no matter how wise, no matter how experienced, no matter how hardpressed for time, no matter how well-intentioned, should be forgiven an uncommented and undocumented program.

Yourdon suggests that a programmer put four or five lines of comments at the beginning of each subroutine, and an average one comment for every two or three lines of source code.

Kernighan and Plauger [KERB 78a] are only lukewarm in their support of comments. They feel that a program needs only "a smattering of enlightening comments". One reason for their reluctance to endorse more voluminous commenting is that the comments do not always match the code, and can therefore be misleading. For this reason, Gerald Weinberg [WEIG 71] suggests covering the comments during debugging so as not to be mislead by them. Since one of the expected uses of comments is to aid in debugging, this technique implies that comments are not universally appreciated.

An approach to this problem is formal comments [KRIB 80]. The idea is to allow the programmer to express some comments as a special set of keywords and syntax that allows the comments to be compared with the program code for discrepancies. This approach cannot completely solve the problem however, since some comments are necessarily informal in nature.

Many authors are willing to offer commenting standards. They include lists of what information to include [PFLS 87], descriptions of how the commenting process should be performed [WOOS 77], and recommendations on the appropriate amount of comment density by source language [ARTL 85]. There is no description of how the debugging programmer or maintenance programmer is to use this information; it seems to be axiomatic that the modifier of a program will find reading the comments to be an effective use of time.

1.4 Experimental Support

The empirical support for the value of comments is mixed. Shneiderman [SHNB 77] compared the value of high level comments to low level comments. His high level comments are general descriptions of the program in a block header at the beginning of the program. His low level

comments are in-line descriptions of specific lines of the program. He found that the high level comments are more helpful than the low level comments as measured in both a program modification task and a program recall task. None of the experimental groups had uncommented programs, as Shneiderman was investigating the ability of programmers to understand programs in terms of "chunks" or groupings, rather than the value of comments per se.

In a study with contrasting results, Sheppard, Curtis, Milliman, and Love [SHES 79] had programmers modify programs with global (block header) comments, in-line comments, and no comments. They measured the time it took the programmers to make the modifications, and the number of errors that the programmers made. They found no significant effect for comments. They hypothesized that the mnemonic variable names in the programs gave the subjects sufficient guidance to the meaning of the program and comments could not add any significant effect beyond this.

Woodfield, Dunsmore, and Shen [WOOS 81] measured the interaction between the use of comments and program organization. To focus on the impact of comments, they used meaningless data names and removed all indentation from their experimental programs. Each program organization was one of four types: monolithic (one single module), functional modularization (organized by function), super modularization (broken up into very small functions), and abstract data type (organized by data objects). They tested subjects by asking them questions about the programs. Subjects with commented programs did significantly better on the test scores than subjects with uncommented programs. In addition, comments were of more help in the programs that were organized into modules than for monolithic programs.

Professional programmers are not required to take tests on program content; they write, debug, and modify programs. In a study by Dunsmore [DUNH 85], subjects were required to make modifications to commented and uncommented versions of programs. The mean mod-

ification scores of the subjects modifying commented programs were 20% better than those of the uncommented programs, but the difference was significant only at the .15 level.

In general, the experimental support for the advantages of comments is not impressive. However, the experimental programs were relatively short (39 to 192 lines). Perhaps the value of comments is only significant for larger programs, results that would be more interesting to industry.

Sheil gives the following analysis of the difficulty of experiments on commenting [SHEB 83].

> Although the evidence for the utility of comments is equivocal, it is unclear what other pattern of results could have been expected. Clearly, at some level comments have to be useful. To believe otherwise would be to believe that the comprehensibility of a program is independent of how much information the reader might already have about it. However, it is equally clear that a comment is only useful if it tells the reader something she either does not already know or cannot infer immediately from the code. Exactly which propositions about a program should be included in the commentary is therefore a matter of matching the comments to the needs of the expected readers. This makes widely applicable results as to the desirable amount and type of commenting so highly unlikely that behavioral experimentation is of questionable value.

1.5 Observations and Motivations

1.5.1 Academic Commenting Practices

One of the goals of computer science education is to prepare students for programming in industry. Industrial programs are frequently modified, placing a premium on documentation to support these changes. In educational environments, especially introductory classes, the emphasis is on commenting at the expense of other forms of documentation.

There are three main approaches to commenting that one sees in classes; program style sheets, general requirements to use comments, and no requirements at all. Professors who

hand out style sheets give students strong guidance on how to comment. In addition, style sheets are implicit instructions on what **not** to comment, discouraging students from cluttering their program with excessive comments. By taking the trouble to create a programming style sheet, the professor conveys a sense of importance to comments. A negative aspect of style sheets is that they tend to result in minimal acceptable behavior. Students rarely volunteer anything not on the style sheet, even if it would be valuable for the particular programming assignment.

Some instructors encourage commenting without clearly specifying what is expected. In some ways this is a good approach, as the student is given more responsibility in choosing the aspects of the program that require comments. Unfortunately, the typical result is the more dedicated students write volumes of comments in their anxiety over maximizing grades while the less dedicated students add little or no documentation to their programs.

Finally, there are some cases where the professor does not ask for any comments. In one, the professor acknowledged that realistically the assigned programs had little chance of actually being maintained after the first version was written. In another, the other professor was a part-time instructor who was currently working in industry with COBOL programs. He felt that comments are rarely changed along with changes in the code and soon become misleading. He suggested that our programs should be self-documenting. This is in agreement with Annino and Russell, who state

> The price of evolutionary change is that flowcharts, prose documentation, detailed descriptions of routines and variables, and program comments often become obsolete, incomplete, or incorrect shortly after they are written. ... For the purposes of computer program development, modification, and enhancement, the only dependable documentation in a changing environment is the source program listing.

Annino and Russell are encouraging the use of very high level simulation languages. Perhaps their approach is correct when the software development environment supports extremely readable source languages.

A basic problem with writing comments for academic programming assignments is that the comments are rarely tested. Most assignments are thrown away after the class is complete. Students have little experience with maintaining programs, either their own or those of their classmates. The comments are never read with the intent of actually maintaining the program. Therefore, the students do not get any feedback on the usefulness of their comments.

1.5.2 Personal Experience with Comments

As a former COBOL programmer at a bank, this author has seen industrial programs with very few comments. The only comments that my employer required of maintenance programmers was a line containing the programmer's name, the date, a project number for billing purposes, and a code that was put in columns 73-80 of each line changed or added.

For the novice programmer, the lack of comments is a burden. Some industry programs are quite unstructured. Comments are useful both for familiarizing new programmers with unstructured programming tricks and teaching the COBOL language.

For more experienced programmers, the lack of comments is less of a problem. As one gains experience, one learns to use other forms of documentation such as narrative program descriptions and identifier cross-references. COBOL is a relatively readable language, and data-processing problems are perhaps simpler than other applications. It seems that most maintenance problems have one of two extreme time frames. One type is extremely time critical so that the programmer has no time to read documentation other than a core dump (an extremely current form of documentation). The other type is not so time critical, allowing the programmer to schedule the production of automated forms of documentation such as identifier cross-references. In general, this author attempts to rely on such automated documentation, as it is more reliable than prose documentation.

There have been two incidents in this author's experience of programmers being unwilling to maintain programs with what they felt was an inadequate number of comments, in spite of the fact that both programs had some comments in them. Maintaining programs is generally not considered desirable work. In addition to not being creative work, it is very difficult to understand programs created by others whether they are commented or not. Therefore, it seems that there are political reasons for commenting programs. It eliminates an excuse for programmers who wish to avoid maintenance work.

Finally, there is a certain amount of emotional value provided by comments. There are many maintenance problems for which the programmer does not actually need to understand the code to make the necessary change. Even in these cases comments can give the maintenance programmer some insight into the inner workings of the program and leave the programmer less anxious about the side effects of the change. In addition, there is occasional humor contained in comments (only some of which is intentional). In my background there was a programmer who expressed great delight whenever he came across a comment referring to the late night crisis working conditions under which the program was fixed or a comment identifying a section of code with no known purpose. Programming is frequently a stressful task. Anything that provides some relief from this stress may be beneficial.

1.6 Conclusion

The support for the practice of commenting programs is unclear. Programming style is an element of human behavior, and human behavior is not easily analyzed. There is much unsupported opinion with respect to commenting style. The empirical research has dealt only with small scale programs.

We analyze two relationships in our study. First, we determine whether programmers really do put comments in the places where they are likely to be beneficial. We did this by assessing the relationships between the number of comment words in a module and a number of software complexity metrics. Next, we look at the relationship between the number of comment words in a module and the ease of making an alteration within that module. This allows us to assess the value of comments with respect to debugging and maintaining programs.

The format of our investigation is a correlational study rather than a controlled experiment. It more closely resembles a field study than a laboratory study. This means that we are measuring the relationships in a more natural setting than we could in a controlled experiment. The subjects perform tasks that are familiar to them. It also allows us to work with larger programs, which can potentially show the value of comments for more complex programming problems.

The next chapter explains the software metrics that we use to assess the complexity of the code that the subjects generate. Chapter 3 describes the data that we collect from the subjects to determine the relationships of interest. Chapter 4 discusses the results of our data analysis. Finally, Chapter 5 draws conclusions from these results and suggests future directions for research.

2.0 Chapter 2: Software Metrics

2.1 Introduction

The structured programming revolution and the introduction of formal design methodologies have resulted from the simple idea that some solutions to programming problems are better than others. The goal of software metrics is to measure differences in source program quality or program complexity. Programmers can use such measures to generate feedback on the value of their design and coding decisions, and to locate potential problem areas in the existing systems.

In some sense, program complexity is a subjective concept. Program code that is difficult to understand or modify is complex by definition. This view of software complexity suggests that since complexity is subjective it should be measured subjectively, by human analysts, instead of by computer-generated analyses. This approach has two drawbacks.

First, subjective measures are notoriously inconsistent. This makes empirical studies difficult to replicate [CONS 86]. The result is that the value of research based on subjective measures is diminished.

Second, the cost of using human software analysts is quite high, both in terms of money and time, especially when one uses multiple analysts to gain some reliability. When measuring program quality, one wants the value of the information derived to be greater than the cost of deriving it. Some of the proposed software metrics can be generated by software analysis programs; others can only be produced by human software analysts. The cost of human analysis is higher than the cost of computer produced analysis. Except for occasions when software quality is extremely important and human analysis can add significant value to the software quality measurement process, automatable metrics have a more favorable cost/benefit ratio than human analysis.

Our study uses automatable metrics exclusively. They are produced by the software quality metric analysis tool, a system of programs that produces software metrics for a growing number of source languages. The system runs under the UNIX™ operating system. Without this software tool, large-scale software analysis such as this study would not be feasible.

The three classes of automatable metrics are code metrics, structure metrics, and hybrid metrics. Code metrics are counts of certain programming constructs. Structure metrics measure the relationships between programming constructs. Hybrid metrics combine code metrics and structure metrics to reflect the interaction of these complexity measures. In general, all code metrics tend to be highly correlated to each other, which suggests that they are measuring similar aspects of program complexity [HENS 81b] [BASV 83] [SELC 87]. Structure metrics have low correlations to other structure metrics, and to the code metrics, which suggests that they are measuring different aspects of program complexity [HENS 81b] [SELC 87].

2.2 Code Metrics

2.2.1 Lines of Code (LOC)

This is perhaps the most intuitive complexity measure. It is generally more difficult to work with long programs than with short programs. It is more difficult to remember all the details for a large program than for a short program. More lines of code mean more chances to make mistakes.

As simple as this measurement sounds, there are some still some choices about what gets included in this measurement [CONS 86]. Should blank lines, comments, variable declarations, and other non-executable lines be included? Should multiple statements on a single physical line add more than one to the count? How should the count be incremented for a single IF-THEN-ELSE statement that sprawls over four text lines? For this study, we use the following definition of line of code from [SELC 87].

> A line of code is counted as the line or lines between semicolons, where intrinsic semicolons are assumed at both the beginning and the end of the source file. This specifically includes all lines containing executable and non-executable statements, program headers, and declarations.

2.2.2 McCabe's Cyclomatic Complexity (CC)

McCabe's Cyclomatic Complexity metric is a measure of the ease of testing the alternative paths within a program or module [MCCT 76]. He feels that determining the total number of possible paths through a program is impractical, so he proposes an alternative measure of testability. This metric represents the decision structure of a program as a set of strongly connected graphs, one graph per procedure. The nodes in each graph represent blocks of

code with no branches in them and the edges represent branches in the program. McCabe adds an implicit edge from the termination vertex to the start vertex of each procedure to make it strongly connected. He borrows the concept of cyclomatic number from graph theory, which can be interpreted as the dimension of the circuit subspace of a graph, or alternatively as the number of regions in a planar graph [LIUC 68]. The formula for the cyclomatic number of a graph with n nodes, e edges, and p connected components is:

$$v(G) = e - n + p$$

When calculating the cyclomatic complexity for one procedure, p = 1, and there is one implicit edge added from the termination vertex to the start vertex to make the graph strongly connected, so the formula is:

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

The complexity of the entire program, with p procedures, is the sum of the complexity of all the procedures, or

$$\begin{aligned} v(G) &= \sum_{i=1}^{p} v(G_i) \\ &= \sum_{i=1}^{p} (e_i - n_i + 2) \\ &= e - n + 2p \end{aligned}$$

McCabe claims to have proved that the cyclomatic number is also equivalent to the number of predicates in a procedure plus one. He chooses to count compound predicates as two rather than one.

2.2.3 Halstead's Software Science

Maurice Halstead has created a collection of software metrics which he feels measures some natural laws of programs [HALM 77]. All of his metrics are based on the following quantities taken from a given implementation of an algorithm:

η_1 = number of unique or distinct operators appearing in that implementation.

η_2 = number of unique or distinct operands appearing in that implementation.

N_1 = total usage of all operators appearing in that implementation.

N_2 = total usage of all operands appearing in that implementation.

2.2.3.1 Program Length (N)

This metric is the total number of tokens in a program. It is defined as

$$N = N_1 + N_2$$

2.2.3.2 Program Volume (V)

This is a measure of program size. It is the number of bits required to store the program. First, we need the concept of program vocabulary η which is the sum of the unique operators and operands in the implementation.

$$\eta = \eta_1 + \eta_2$$

One needs $\log_2 \eta$ bits to specify a unique bit pattern for each of the η vocabulary elements. Since there are N tokens, each of which has a $\log_2 \eta$ bit pattern to represent it, the number of bits needed to specify the entire program is

$$V = N \log_2 \eta$$

2.2.3.3 *Program Level (L)*

Program level is an attempt to measure the ease of implementation of an algorithm in a given language. When implemented in the highest level language appropriate to the problem, an algorithm has the smallest possible volume, called the potential volume or V^*. The program level is equal to the potential volume divided by the actual volume.

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

However, in general V^* is not directly observable. Therefore, an estimator is needed. In theory, a higher level language would have built-in functions to do most of the processing and would not require the operands to be repeated frequently. With respect to a given algorithm, the highest level language would have a built-in function for the entire algorithm.

The fewest number of operators needed is a function designator and a grouping operator for the operands; a total of two. Therefore program level is proportional to two divided by the actual number of operands.

$$L \sim \frac{2}{\eta_1}$$

A higher level language allows the operands to be repeated less frequently. Therefore, the program level is proportional to the ratio of unique operands to total operands.

$$L \sim \frac{\eta_2}{N_2}$$

Combining these two measures of program level gives

$$L = \frac{2}{\eta_1} \times \frac{\eta_2}{N_2}$$

A program written in the highest level language appropriate for the problem has L = 1. As the language level decreases, L approaches 0.

2.2.3.4 *Programming Effort (E)*

Effort is dependent on program volume and program level. The formula for effort is

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

By substituting $\frac{V^*}{V}$ for L one obtains

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

From this we see that for an algorithm with a given potential volume, effort increases with the square of the volume.

2.3 Structure Metrics

2.3.1 McClure's Module Invocation Complexity

McClure's metric is in part a reaction to McCabe's Cyclomatic Complexity. McCabe gives equal weight to each predicate in a program. McClure suggests that this approach discards an important component of complexity, namely the number of different variables used to construct the predicates [MCCC 78].

McClure's metric is concerned with the calling relationships between subprograms (procedures, functions, subroutines, etc.) which she refers to as modules. A programmer attempting to understand a module must contend both with the module's processing and the circumstances under which it can be called, or invoked. For each place in a program where a module may be invoked, there are zero or more variables that determine whether it will be invoked or not. These variables are called the **invocation control variable set**. McClure gives the following example of a module X being invoked by another module:

```
IF v1 = 3 and v2 > v3
   Call X while v4 = 0
```

Module X has the control variable set $\{v_1, v_2, v_3, v_4\}$ for this invocation. Other invocations may have different control variable sets. The complexity of an invocation control variable set is

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

where

b	=	1 if the invocation is from a selection structure or nested selection structure.
b	=	2 if the invocation is from a repetition structure
$C(v)$	=	the Control Variable Complexity Function [MCCC 78]
e	=	the number of variables in the invocation control variable set

The representation of control structure in the software quality metrics analysis tool does not discriminate between selection structures and repetition structures [HENS 88]. Therefore, the implementation of the McClure metric in the software quality metrics analysis tool always sets the value of b equal to 1.

The complexity of module p, denoted $M(p)$, is determined by two factors. The first is the complexity contributed by each invocation of module p. The second is the complexity contributed by each module that p invokes. The formula for $M(p)$ is

$$M(p) = [f_p \times X(p)] + [g_p \times Y(p)]$$

where

f_p	=	the number of modules that directly invoke p
$X(p)$	=	the average complexity of all invocation control variable sets used to invoke module p
g_p	=	the number of modules directly invoked by p
$Y(p)$	=	the average complexity of all invocation control variable sets used by module p to invoke other modules.

2.3.2 Henry and Kafura's Information Flow Metric

Henry and Kafura's metric [HENS 81a] is a measure of the effects that procedures have on each other due to calling relationships and the sharing of data. The metric is based on information flow between procedures. The following definitions are essential to understanding the metric.

local flow There is a local flow of information from module A to module B if one or more of the following conditions hold:

1. if A calls B,

2. if B calls A and A returns a value to B, which B subsequently utilizes, or

3. if C calls both A and B passing an output value from A to B.

fan-in The fan-in of procedure A is the number of local flows into procedure A plus the number of data structures from which procedure A retrieves information.

fan-out The fan-out of procedure A is the number of local flows from procedure A plus the number of data structures which procedure A updates.

The Henry-Kafura Information Flow metric is calculated as follows:

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

where

C_p = the complexity of procedure p

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

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

Fan-in and fan-out reflect the strength of connections between a procedure and the rest of the program, a measurement of the coupling concept [STEW 74]. The product of fan-in and fan-out is equal to the total number of pairs of input sources and output destinations in a given procedure. By squaring this product, large penalties are imposed on procedures with excessive connections to the remaining sections of the program.

2.4 *Hybrid Metrics*

2.4.1 Woodfield's Syntactic Interconnection Model

Woodfield's metric is a measure of how the calling structure of a program determines the amount of time a programmer must spend in each procedure to understand the program as a whole [WOOS 80]. The calling structure of a program can be represented as a graph, with the procedures represented as nodes and the calling relationships represented as edges. Ideally, this graph would be a tree. Each procedure would have only one parent; it would be called from only one place in the program. Therefore, a programmer would have to examine a procedure only once during an analysis of the program. In practice, some procedures are called from multiple places in the program. This requires that these procedures be examined more than once. One would expect these additional examinations to take less time.

A key determinant of Woodfield's metric is the number of times a procedure must be examined which he calls fan-in. Fan-in is the number of connections a procedure has to the rest of

the program. This is the sum of control connections, data connections, and implicit connections. Note that this is **not** the same as Henry and Kafura's concept of fan-in [HENS 81a]. A control connection occurs when a procedure calls another procedure. A data connection occurs when one procedure updates a data item and another uses the updated value. An implicit connection occurs when an assumption is used in two or more procedures. Implicit connections cannot be determined by an automated system, so they are not included in our implementation of this metric.

Another important determinant of this metric is the expected decrease in time needed to understand a procedure on each subsequent examination. This is reflected in the review constant, RC. Traditionally, RC is set to $\frac{2}{3}$ based on a suggestion of Halstead [HALM 77].

The formula for Woodfield's metric is

$$C_B = C_{iB} \times \sum_{k=2}^{fan\_in-1} RC^{k-1}$$

where

C_B	=	the complexity of module B's code
C_{iB}	=	the internal complexity of module B's code
fan_in_i	=	the sum of the control and data connections to B
RC	=	the review constant.

Woodfield uses Halstead's effort metric E for the internal complexity C_{iB}. He states the need for C_{iB} to come from ratio scale; E is a metric that qualifies.

2.4.2 Henry and Kafura's Information Flow Metric

Henry and Kafura's metric can be combined with a code metric to measure the interaction between the internal complexity of a procedure and its coupling to the rest of the program. The formula for this hybrid version of the information flow metric is

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

where

C_p	=	the complexity of procedure p
C_{ip}	=	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

The internal complexity of procedure p, C_p , is measured by using one of the code metrics. Henry and Kafura use lines of code as the internal complexity measure C_{ip} in a study that demonstrated the validity of their metric on the UNIX™ operating system [HENS 81a]. In this study we also use lines of code as the internal complexity measure. Since the different code metrics are highly correlated with each other [HENS 81b] [BASV 83] [SELC 87], the code metric chosen should not be relevant.

2.5 *The Software Quality Metric Analysis Tool*

All software metrics in this study are produced by the software quality metric analysis tool. This system of programs produces the software metrics mentioned above for the programming languages Pascal, C, Fortran, THLL, and Ada, and a program design language (PDL).

It has been developed by software engineering students at Virginia Tech under the direction of Dr. Sallie Henry and Dr. Dennis Kafura. The system has already been used in a number of software quality studies.

The system is organized into three stages, as shown in Figure 1 on page 25. The first two stages are command line programs; the third stage is interactive. Only the first stage is source language dependent; the second and third stages work with intermediate representations of the source code. The first stage is responsible for translating the given source language into a common form called relation language code [HENS 88], and for generating code metrics (lines of code, McCabe's Cyclomatic Complexity, and the Halstead Software Science metrics). This relation language is a generic procedural language that has all differences in control structure and constants abstracted away. There is a version of the first stage for each source language. The second stage takes relation language code and derives the relations among procedures that are required to calculate structure and hybrid metrics. The third stage takes the code metrics produced by the first stage and the relations produced by the second stage and uses this information to generate each of the metrics discussed in this chapter. This program allows the user to create output files containing the metrics for use in statistical analysis systems. It also allows the user to organize subprograms into modules to control the granularity of the analysis.

2.6 Conclusion

The discipline of software metrics gives software professionals and researchers the means to analyze programs objectively. The large number of different metrics, a fraction of which have been mentioned here, is a testimony to the many different aspects of software that make its

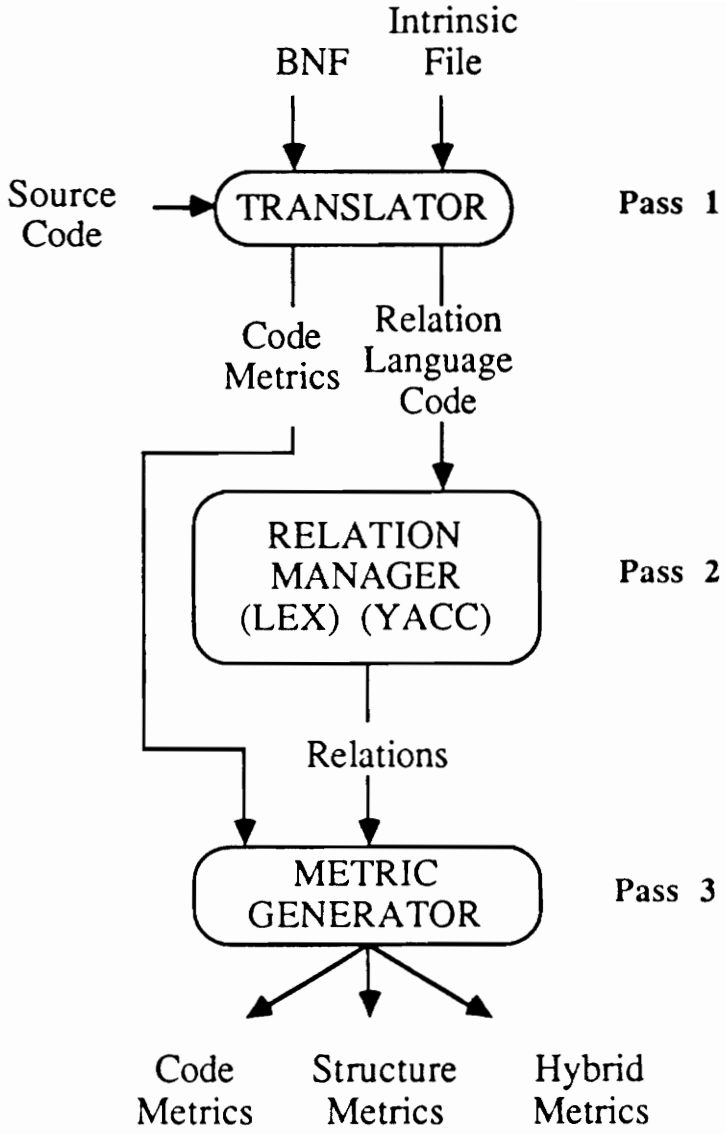

Figure 1. Software Metric Analyzer

Figure 1. Logical Structure of the Software Quality Metric Analyzer

construction and maintenance a difficult and expensive process. The next chapter describes the data collected to assess the software relationships of interest.

3.0 Chapter 3: Empirical Data

3.1 Introduction

The objective of this study is to assess the use of comments in an environment as demanding as that which software professionals face. Data pertaining to software metrics, comments, and program change descriptions is collected. Interviews with the subjects provide additional insight into the use of comments.

3.2 Subjects

The subjects are all senior or graduate level computer science students taking part in a software engineering class. The students participate in team project work that includes program specification, design, coding and integration. Therefore, the students are exposed to a large portion of the software life cycle. While they have some choice over the computer hardware

and software that they use, all students are given accounts on the Virginia Tech cluster of VAX 11/785's. All students use this system for some of their design work and for software change reporting.

The class is divided into as many teams of three students as possible, with one or two teams absorbing any additional students if necessary. In this study the class is divided into ten teams of three students. The team members are assigned by an automated team generator program that attempts to create teams that have approximately the same ability and are compatible. The team generator algorithm uses the following data to assign the students to 'optimal' teams [HENS 83]:

1. Computer Science courses previously taken and the grades received.

2. Computer Science courses in which the student is currently enrolled (more difficult courses imply that the student might not have much time to spend on the course).

3. Whether or not the student is interviewing (interviewing students have less time).

4. Whether or not the student is graduating (graduating students occasionally have motivation problems).

5. The times during the day and on weekends when the student is free.

6. Two people with whom the student would like to work (used only if the team is still 'optimal').

7. One student with whom the student will not work (There are always personality clashes, the algorithm guarantees that this will be considered).

There are no particular roles assigned to the team members. The students determine their own team organization.

Each team is required to design a program that they expect to be 3000-5000 lines long. They must specify their design using a program design language (PDL) [SELC 87]. The design is verified to be syntactically correct. The teams give presentations on their designs to their classmates. These presentations include a description of the project, the data structures used, the modules that comprise the project, and the hierarchical relationships among the modules.

The teams then contract five or more classmates to produce code that implements the specifications. No student may code for his own team. Each team has 300 points to award to classmates who work for them, giving an average of 100 points available per student. The accumulation of these points determines a portion of each student's class grade. The team members must specify the basis for allotting these points before the programmers contract to work for them, using such factors as difficulty, quality of result, and meeting deadlines. However, since the modules are assigned by auctioning them to classmates, the actual number of points contracted on a module may be different from the original value. The teams are encouraged to assign negative points to work that is not completed to discourage students from contracting for more points than they need and ignoring some of their obligations. Each student must work for at least three teams, insuring that the student will view at least four different design specifications, including the one produced by his team.

Next, the team members integrate, test, and debug the code fragments they receive. The designers and coders are required to log any changes to their programs using an on-line logging system available on the Virginia Tech cluster of VAX 11/785's. Finally, the students present the working programs for approval.

The students choose a variety of different projects. Naturally games are popular. Seven of the projects are game-related, including two adventure games, backgammon, cribbage, Clue™, a collection of casino games, and a gambling game simulator for testing gambling strategies in poker and blackjack. Two of the projects are academic in nature, a class scheduler and an improved team generator for software engineering classes. The remaining project is a custom editor for writing Pascal programs.

The actual programs produced range from 6000-10,000 lines long, including comments and blank lines. Therefore, the programs are representative of the complexity found in some industry programs.

The design teams are allowed to set coding standards for their programmers, including the option of no standards at all. These standards describe the desired form of the contracted code. They specify such aspects as maximum module size, data item naming conventions, expectations for unit testing, requirements for code walkthroughs, commenting requirements, and penalties for not following standards. The length of these standards range from slightly less than one page to two full pages. All but one of the teams (team 5) require comments of some sort. Of the teams that require comments, all require block headers and descriptions of local variables. The coding standards for teams one, two, four, seven, eight, and nine appear in Appendix A.

The scope and complexity of the projects makes the class quite demanding. The students are forced to work with large-scale software projects. They must coordinate their efforts with other designers and programmers; therefore they endure the problems associated with differences in programmer abilities, personalities and motivations. Since the work must be completed in a ten-week academic quarter, the students are under time pressure from the first week of classes. The students face many of the problems of software professionals.

Data derived from such a class can be expected to have characteristics similar to those found in industry [HOLR 87]. While the subjects are not professional software developers, they are experienced programmers. They are all well versed in modern approaches to program structure, unlike some professional programmers. They all use the Pascal programming language. By using subjects with roughly similar training and the same programming environment, we intend to reduce the variance in software produced based on these factors to allow us to concentrate on the effects of program complexity, commenting style, and software maintainability.

3.3 Assessment

3.3.1 Software Complexity Metrics

The following eight software complexity metrics are used to assess the performance of the team: lines of code, the Halstead Software Science metrics N, V, and E, McCabe's Cyclomatic Complexity, Henry and Kafura's Information Flow Metric, Woodfield's Syntactic Interconnection Model, and McClure's Module Invocation Complexity measure. The logarithm of Henry and Kafura's metric is taken before using it in the data analysis. This is necessary because of the large values generated (mean of 3,521,072; largest value 1,254,528,000). The effect is to turn the metric into a linear combination of the log of the metrics lines of code, fan-in, and fan-out, with fan-in and fan-out weighted twice as heavily as lines of code. All complexity metrics are generated automatically by an automated software metrics analysis system. The output of the metrics system consists of eight metric scores for each of the 839 modules (procedures and functions) coded by the teams involved in the study [HENS 88].

3.3.2 Comment Measurements

Commenting style is measured by counting the number of comment words in each module (procedure or function) of each program. It is currently impossible to measure the amount of usable information in comments or the value of what is there. However, by counting the number of words, one can assess how much of an effort the designer or programmer made when commenting the program code. Also, one can evaluate the idea that more documentation is always better as reflected in the "too much documentation is impossible" attitude mentioned in the introduction [SHES 79]. In the remainder of this paper, we refer to the count of comment words as "comment quantity".

The algorithm on page 18 of *The C Programming Language* [KERB 78b] is used to count comment words. It counts the number of strings of contiguous non-blank characters. The output of this algorithm matches that of the UNIX filter "wc". All '*' and '-' characters are removed from the comments before counting words.

There are some alternative comment measurements that were tested and discarded. One of them is a count of non-blank characters within comments. This did not provide much additional information beyond the number of words, as the correlations between non-blank characters and words within comments are 0.97 or greater. See Appendix B for these correlation coefficients. Another is the number of words before discarding '*' and '-' characters. This count of the number of words correlates 0.96 or greater with the count that was used. See Appendix C for these correlation coefficients.

The most sophisticated attempt to measure comment style that was considered is the use of text readability scores [GUNR 52] [SMIE 70]. The program "style", available under UNIX, provides estimates of reading level in terms of academic grade level. Unfortunately, readability scores are dependent on sentence length for the calculations, the assumption being that

```
procedure plot_cursor(cursoron, numchips : integer);
{*****************************************************
 PLOT_CURSOR          puts cursor at the top of the stack
                  of pieces indicated by the CURSORON variable

 parameters:
   CURSORON          point on board that cursor should be on
   NUMCHIPS          number of pieces on a point on the board

 calls:
   find_row_column, moveto
*****************************************************************}

var
  row,              { row where cursor is to be placed }
  column              { column where cursor is to be placed }
  :integer;

begin { plot_cursor }
  { get the base of the row and column }
  find_row_column(CURSORON,row,column);

  { adjust the column to the number of chips on the point
    places them to the left and right accordingly }

  if (cursoron < > THEHOME) then
    begin
      if ((numchips < = two_column) and (numchips > one_column)) then
        column := column - 1
      else
        if (numchips > two_column) then
          column := column + 1;

      { adjusts the column so that the cursor is always at the piece
        on the point that is to be used. }

      if (numchips < > 0) then
        if (cursoron < = mid1_board) then
          row := row - ((numchips-1) mod column_size)
        else
          if (cursoron < BAR) then
            row := row + ((numchips-1) mod column_size);
    end;
  moveto(row,column);
end; { plot_cursor }
```

Figure 2. A Procedure from Group Nine

longer sentences are more difficult to understand. Programmers are under no obligation to add punctuation to their comments, and so have negligent punctuation habits. The style analysis program considers such strings of unpunctuated comments to be one long sentence. The readability scores were erratic; they varied from a low grade level of -3.4 to a high grade level of 76.7 and were therefore discarded.

The comment quantity measurements are divided into three categories: block, data, and code. Block comment quantity words are contained in the block header comments at the beginning of the program. Data comment quantity words are extracted from comments annotating constants, types, and variables. They reflect an explanation of local data for the function or procedure. Code comment quantity words are extracted from "in-line" comments between the first begin and the newline character on the line of the last end within a procedure or function. Therefore, each of the 839 modules has four comment scores: the total number of comment words within the procedure or function, the number of comment words in the block header comment, the number of comment words in the constant, type, and variable declaration area, and the number of comment words in the code area.

A sample procedure from team nine (backgammon) is shown in Figure 2 on page 33. After removing all '*' and '-' characters one derives a count of 39 block comment words, 14 data comment words, and 49 code comment words, giving a total of 102 module comment words.

3.3.3 Program Change Analysis

The subjects are required to log each change they make to their programs. They use an interactive change logging program that was written by Calvin Selig and is available on the Virginia Tech cluster of VAX 11/785's. The program prompts the user for the desired information. The information includes the following

- project team
- person making the log
- person who made the program alteration
- the name of the procedure or function
- the life-cycle phase (design, coding, integration, or maintenance)
- the time it took to make the change
- the subjective difficulty of making the change, from one (lowest) to 7 (highest)
- the type of change made (parameter, local variable, routine call, routine creation, logic, or syntax)
- the number of lines changed
- a one to 5 line explanation of the change

Each log data item consists of a team and procedure number (to match it to the metric and comment scores associated with that procedure) the number of lines changed, and the time it took to make the change.

3.3.4 Interview

At the end of the class, an interview was held with each team to discuss their experiences with the effectiveness of comments. This interview contributed one third of their final exam

grade. In general, the students were enthusiastic about sharing their opinions. Four themes occurred during these interviews.

First, many of the students felt that their coders did **not** adhere to the team coding standards. Also, many of the students admitted that they ignored the standards given to them. In general, programmers seem to be rather inflexible with respect to coding style. They have strong opinions about the nature of a "good" program, though these opinions vary widely among individuals. Two of the programmers mentioned a running feud that they had on the proper use of blank lines within programs. Many programmers considered style a matter of personal preference, and had no objective way of labeling one style as better than another. One team (team 5) decided that programmers would resist having a coding style dictated to them and chose not to enforce any coding standards. In addition, the time constraints on the project encouraged some programmers to cut corners as deadlines drew near. This behavior is in agreement with Scott Costello's model of the deadline response decision [COSS 84], in which he predicts that as the deadline approaches, behavior shifts toward making the software "minimally functional" and away from "software engineering activities, such as design review, code review, **documentation**, and rigorous testing" [emphasis added].

Second, the integrators differed in their strategies for using comments during the integration, debugging, and testing phases. Some integrators ignored all comments. Some integrators only read block header comments, and ignored all in-line comments. Some integrators only read in-line comments and ignored all block comments. Differences in comment reading strategies appeared even within teams, in spite of the team members agreeing on appropriate commenting standards.

Third, many of the subjects admitted that the comments were not updated along with program changes. This is alleged to be a problem found in industry also. Martin and McClure state that "documentation ... soon becomes out of date due to changes to the software" [MARJ 83]. The reasons for this are pointed out by Fjeldstad and Hamlen [FJER 79]:

> Programmers are not technical writers and do not want to be. If proper documentation would take significant additional time over that spent on making the change, and another job waits to be started, they find it hard to justify the cost or establish the value of doing the documentation.

Fourth, many of the students defended the use of comments for providing information that could be derived from alternative sources of documentation. For example, the students felt that comments should be used to provide the calling relationships within a program, even though some of this information can be derived from hierarchy charts and identifier cross-references. They supported their opinion by pointing out that hardcopy data lags behind the current program during the testing and debugging phases of software development. They prefer information that is available immediately at the terminal (comments) to information that comes from a printer or manual. There is nothing that logically precludes having a hierarchy chart or identifier cross-reference available on-line. However, this was not the case in the software development environment being used; the students' approach to keeping documentation on-line is to keep it in the form of comments. Since most of the logs had short time intervals (80% were 20 minutes or fewer), and the turnaround time for printouts was frequently over an hour, the students were reluctant to use any software tool that required a printout.

3.4 Conclusion

The software environment being investigated is sufficiently demanding to approximate the challenges faced by professional programmers. There are a wide variety of measurements to assess commenting behavior and its value. The student interviews provide insight into the human aspects of programming behavior that are difficult to quantify, yet perhaps are the most important.

4.0 Chapter 4: Results of Analysis

4.1 Introduction

There are two major relationships to be assessed. The first is the relationship between comments and module complexity, with module being defined as either a function or procedure. The second is the relationship between comments and the ease of making alterations to modules. In addition, the differences between teams and the influence this has over the results is observed.

4.2 Analysis of Project Teams

Seven of the teams chose to implement their project under VAX/VMS on the Virginia Tech cluster of VAX 11/785's. The other three teams implemented their programs on microcomputers; two under Turbo Pascal on IBM PCs, and one under Lightspeed Pascal on an Apple

MacIntosh. These three teams (teams 3, 6, and 10) were eliminated from analysis, due both to differences in program organization forced upon these teams by the microcomputer compilers, and because their software change logs were of questionable value since they were not made on-line.

The projects vary a considerable amount, as shown in Table 1 on page 40. Note that the number of lines refers to the number of text lines in the source file, not the software metric LOC. Also note that the number of comment lines is the number of source lines that have a comment on at least part of the line.

Since the students produce formal PDL designs in addition to program source code, it is possible to assess complexity through software metrics on both the designs and the final products. The differences between the design metrics and program metrics indicate the kinds of designs produced by of the teams. Notice the similarity in design and program metrics for teams five and nine, as shown in Table 3 on page 42. This implies that these teams produced detailed designs, while the other teams produced high level or architectural designs. The abbreviations in this table and all other tables in this chapter are expanded in Table 2 on page 41.

4.3 Analysis of Change Logs

There are 1129 entries to the logging system. Many of these are inappropriate to the study, for the following reasons. First, some of the logs are entries that pertain to the design phase of the software life-cycle. This is not a phase of the software life-cycle being studied. Second, some of the logs are entries describing the creation of a module. This study is only concerned with changes to software. Third, some of the logs have explanations indicating that changes

Table 1. General Project Statistics

Project	Modules	Lines	Comment Lines	Characters	Comment Characters
1	154	10,249	3,805	344,711	175,648
2	141	8,594	3,029	275,546	151,065
4	163	10,610	3,363	343,731	171,428
5	61	6,151	1,880	220,412	89,427
7	118	7,833	3,044	329,829	165,644
8	110	6,802	3,008	238,688	139,987
9	92	6,493	2,294	214,454	108,496

Table 2. Abbreviations

Metric	Abbrev
Lines of Code	LOC
McCabe	CC
Halstead Length	N
Halstead Volume	V
Halstead Effort	E
McClure	MC
Log of Henry-Kafura	LOGHK
Woodfield	WOOD
Module Comment Quantity	MCQ
Block Comment Quantity	BCQ
Data Comment Quantity	DCQ
Code Comment Quantity	CCQ
statistical significance	sig

Table 3. Software Metrics by Team

Team	LOC	N	V	E	CC	WOOD	Fan-in	Fan-out
1 Design	1,345	6,371	29,242	516,260	299	651,631	522	667
1 Program	3,319	26,313	142,954	7,269,142	990	10,941,418	860	864
2 Design	895	5,355	26,270	672,303	218	793,383	291	310
2 Program	3,175	21,601	112,016	6,297,943	858	7,708,571	629	663
4 Design	1,003	5,162	24,815	567,515	257	573,079	277	310
4 Program	3,992	29,128	145,460	14,629,702	1,130	18,756,640	702	731
5 Design	2,612	27,540	112,084	134,888,298	198	134,996,255	226	259
5 Program	3,168	32,001	129,920	144,267,206	416	144,575,861	213	223
7 Design	1,123	5,669	27,575	697,329	157	826,597	393	449
7 Program	3,263	22,829	125,794	7,774,959	987	11,394,059	703	716
8 Design	722	4,081	21,443	365,371	132	378,072	305	315
8 Program	1,928	14,194	75,514	3,287,319	683	3,671,946	673	680
9 Design	1,565	11,384	61,949	2,224,960	389	2,459,734	400	436
9 Program	1,882	14,815	78,716	4,103,286	486	4,767,457	468	473

to more than one module are being described only in the given log. There is no method of distributing the number of minutes spent and number of lines changed to the appropriate modules. All three of these types of logs are eliminated, leaving 604 logs remaining for analysis.

4.4 Statistical Methods

We assess the relationships of interest in our study using two statistical tools. Correlation coefficients are used to measure the relationship between a pair of variables. As pointed out by Edwards [EDWA 84], "when the variables are linearly related, the correlation coefficient is a measure of the degree of relationship present". Linear regression is used to measure the degree of relationship between one response variable and a collection of predictor variables. All statistical analyses are performed by the statistical package SAS™.

All correlation coefficients are represented by R, the Pearson product-moment correlation [SASB 85]. The formula for the correlation coefficient between the variables x and y is

$$R_{xy} = \frac{\Sigma(x-\bar{x})(y-\bar{y})}{\sqrt{\Sigma(x-\bar{x})^2\Sigma(y-\bar{y})^2}}$$

The numerator of this formula suggests that when the deviations from the mean of both the x and y variables tend to have the same sign, the result will be a positive correlation. This case occurs when large values for one variable tend to be paired with large values for the other variable and small values for the first variable tend to be paired with small values for the second variable. If the deviations tend to have different signs, the result will be a negative correlation. This case occurs when large values of one variable tend to be paired with small values of the other variable and small values of the first variable tend to be paired with large

values of the second variable. If the signs of the deviations have no consistent pattern, the correlation coefficient is near zero, suggesting an unsystematic relationship between the variables. The terms in the denominator restrict the correlation coefficient to values between -1 and +1.

Linear regression is the process of choosing a parameter for each of a number of variables to create a model that predicts the value of another variable. These parameters are chosen using the principle of least squares [SASS 85]. In general, the main result that we use from the linear regression analyses is the value of R^2 which is the percentage of the variation of the response variable that the model explains [DRAN 81]. Using Y to represent the predicted variable in question, then Y_i represents the ith value of Y, $\overline{Y}$ represents the mean value of Y, and $\hat{Y}_i$ represents the value that the model predicts for the Y_i. The total sum of squares is equal to the sum of squared deviations of each Y_i value from the mean value of Y.

$$SS_{tot} = \Sigma(Y_i - \overline{Y})^2$$

Each predicted value of Y, $\hat{Y}_i$, is rarely exactly equal to the value of Y_i that it is trying to predict. Therefore, there is some error in most models. The error sum of squares is equal to the sum of the squared differences of each predicted value of Y_i and the actual value of Y_i.

$$SS_{err} = \Sigma(Y_i - \hat{Y}_i)^2$$

The sum of squares due to regression is equal to the total sum of squares minus the error sum of squares.

$$SS_{reg} = SS_{tot} - SS_{err}$$

The value of R^2 is equal to the ratio of sum of squares due to regression divided by the total sum of squares.

$$R^2 = \frac{SS_{reg}}{SS_{tot}}$$
$$= \frac{SS_{tot} - SS_{err}}{SS_{tot}}$$

Note that if the error sum of squares is small with respect to the total sum of squares, then the sum of squares due to regression is large relative to the total sum of squares and the value of R^2 is close to 1. Also note that if the error sum of squares is large with respect to the total sum of squares, then the sum of squares due to regression is small relative to the total sum of squares and the value of R^2 is close to 0. Therefore, high values (close to 1) of R^2 indicate a model that predicts the values of Y well, while low values (close to 0) of R^2 indicate a model that predicts the values of Y poorly.

4.5 Analysis by Module

4.5.1 Approach

A record is built to describe each of the 839 modules (procedures and functions) in the study. Each module description consists of 15 numbers:

- the 8 software complexity measures
- the 4 comment quantity measures (module, block, data, code)
- the number of changes logged to the procedure
- the number of lines changed in the procedure
- the number of minutes spent changing the procedure

These values allow the assessment of the relationships between the variables through correlation coefficients and regression analysis.

4.5.2 Correlational Results

4.5.2.1 Comment Quantity and Software Metrics

Module comment quantity shows a significant and positive correlation with each software metric except Halstead's E and Woodfield's metric, as shown on Table 4 on page 47. This may have been an artifact of the technique of using linear correlation to assess a non-linear measurement (E). Halstead's E and Woodfield's metric are correlated 0.9995 with each other. Woodfield's metric is equal to a calculated factor times Halstead's E. For the most common case of a module being called from only one other module, this factor is equal to one. The highest correlations of module comment quantity are with the logarithm of Henry and Kafura's metric (0.4598) and with McCabe's Cyclomatic Complexity (0.4007).

Block comment quantity also shows a significant and positive correlation with each software metric except Halstead's E and Woodfield's metric, as shown on Table 5 on page 48. The significant correlations are lower than those of module comment quantity, with the exception of the McClure metric. Again, the highest correlations are with the logarithm of Henry and Kafura's metric (0.3734) and with McCabe's Cyclomatic Complexity (0.2952). The relatively high correlation of Henry and Kafura's metric to comment quantity may reflect the practice of listing calling relationships between modules and describing each of the module parameters in the block comments. Calling relationships and number of parameters are measures of module coupling, as is Henry and Kafura's metric.

Table 4. By Module: Correlation of module comment quantity with software metrics

	LOC	N	V	E	CC	WOOD	MC	LOGHK
R	0.2478	0.1538	0.2721	0.0023	0.4007	0.0054	0.0815	0.4598
sig	0.0001	0.0001	0.0001	0.9474	0.0001	0.8752	0.0199	0.0001

Table 5. By Module: Correlation of block comment quantity with software metrics

	LOC	N	V	E	CC	WOOD	MC	LOGHK
R	0.0945	0.0862	0.1359	0.0140	0.2952	0.0169	0.1814	0.3734
sig	0.0061	0.0125	0.0001	0.6849	0.0001	0.6242	0.0001	0.0001

Data comment quantity shows the same pattern of significant relations as the previous two, with the exception that the correlation between data comment quantity and the McClure metric is not significant, as shown on Table 6 on page 50. The correlation of data comment quantity with McCabe's Cyclomatic Complexity is much lower than that of any of the other three comment measures, though it is still significant. It is most highly correlated with the logarithm of Henry and Kafura's metric (0.3360). A possible explanation of this is the number of local variables may be indirectly related to module coupling. The number of module parameters may be correlated with the number of local variables, as data intensive modules may have many parameters and many local variables. The number of module parameters is correlated with module coupling. Henry and Kafura's metric is a measure of module coupling.

Code comment quantity shows the same pattern of significant relations as data comment quantity (no significant relationship with McClure, E, or Woodfield's), as shown on Table 7 on page 51. Its highest correlation is with McCabe's Cyclomatic Complexity (0.4465). This is no surprise, as some programmers automatically put comments on case statements, end statements, if statements, and else clauses. Team 2 had standards that required such comments. In effect, programmers are pairing a comment with many of the predicates in the program. McCabe's metric is equal to the number of predicates plus one.

4.5.2.2 Comment Measures with Comment Measures

The comment measurements are all significantly correlated with each other. However, the correlations are low (below 0.20) with the exception of module comment quantity with each of the other three. This is to be expected, as module comment quantity is equal to the sum of the other three. See Appendix D for these correlation coefficients.

Table 6. By Module: Correlation of data comment quantity with software metrics

	LOC	N	V	E	CC	WOOD	MC	LOGHK
R	0.2429	0.1070	0.2255	-0.0078	0.1006	-0.0073	-0.0243	0.3360
sig	0.0001	0.0019	0.0001	0.8213	0.0035	0.8330	0.4888	0.0001

Table 7. By Module: Correlation of code comment quantity with software metrics

	LOC	N	V	E	CC	WOOD	MC	LOGHK
R	0.1383	0.1111	0.1676	-0.0020	0.4465	0.0012	-0.0051	0.1571
sig	0.0001	0.0013	0.0001	0.9549	0.0001	0.9731	0.8848	0.0001

4.5.2.3 *Software Metrics and Comment Measures with Change Log Data*

It is intended to use comment quantity as an addition to more traditional software metrics in the prediction of software problems. It is not obvious whether one can expect changes to have a positive or negative correlation with comment quantity. A large number of comments in a module can suggest that the programmer has put considerable effort into a module. It could also suggest that the programmer realizes that there is a lot of complexity within the module, perhaps more than traditional software metrics are able to detect.

The measurements having the strongest relationships with the change log data are the logarithm of Henry and Kafura's Metric, module comment quantity, and McCabe's Cyclomatic Complexity. The correlation of Henry and Kafura's Metric with the number of change logs is 0.3357 (see Table 8 on page 53) with the number of minutes spent making changes is 0.2728 (see Table 10 on page 55), and with the number of lines changed is 0.2457 (see Table 12 on page 57). The correlation of module comment quantity with the number of change logs is 0.2782 (see Table 9 on page 54), with the number of minutes spent making changes is 0.2232 (see Table 11 on page 56), and with the number of lines changed is 0.1988 (see Table 13 on page 58). The correlation of McCabe's Cyclomatic Complexity with the number of change logs is 0.2070, with the number of minutes spent making changes is 0.1806, and with the number of lines changed is 0.1724. All software metrics and comment measures show significant positive correlations with each of the change log statistics, with the exceptions of E, Woodfield, and McClure. None of these had significant correlations with any of the change log statistics.

4.5.2.4 *Software Metrics with Software Metrics*

In general, the relationships between the pairs of software metrics are similar to those found in other studies [HENS 81b] [BASV 83] [SELC 87]. Code metrics have high correlations to other

Table 8. By Module: Correlation of software metrics with number of change logs

	LOC	N	V	E	CC	WOOD	MC	LOGHK
R	0.1217	0.0864	0.1517	0.0025	0.2070	0.0055	-0.0014	0.3357
sig	0.0004	0.0123	0.0001	0.9435	0.0001	0.8732	0.9692	0.0001

Table 9. By Module: Correlation of comment quantity with number of change logs

	Module	Block	Data	Code
R	0.2782	0.2274	0.1554	0.1603
sig	0.0001	0.0001	0.0001	0.0001

Table 10. By Module: Correlation of software metrics with number of change minutes

	LOC	N	V	E	CC	WOOD	MC	LOGHK
R	0.1094	0.0733	0.1301	-0.0012	0.1806	0.0024	-0.0162	0.2728
sig	0.0015	0.0339	0.0002	0.9727	0.0001	0.9446	0.6450	0.0001

Table 11. By Module: Correlation of comment quantity with number of change minutes

	Module	Block	Data	Code
R	0.2232	0.1360	0.1512	0.1526
sig	0.0001	0.0001	0.0001	0.0001

Table 12. By Module: Correlation of software metrics with number of lines changed

	LOC	N	V	E	CC	WOOD	MC	LOGHK
R	0.1403	0.0908	0.1428	0.0043	0.1724	0.0116	-0.0118	0.2457
sig	0.0001	0.0085	0.0001	0.9009	0.0001	0.7374	0.7376	0.0001

Table 13. By Module: Correlation of comment quantity with number of lines changed

	Module	Block	Data	Code
R	0.1988	0.1296	0.1170	0.1494
sig	0.0001	0.0002	0.0007	0.0001

code metrics. Structure and hybrid metrics have low correlations to each other and to code metrics. See Appendix O for the correlation values. Lines of code and the three Halstead metrics all have correlations of at least 0.89 with each other. The McClure metric has no correlation with any other metric that was higher than 0.21. The logarithm of Henry and Kafura's metric has no correlation with any other metric that was higher than 0.35. The are two exceptions to the general rules. One is Woodfield's metric, which is highly correlated with Halstead's E and the other code metrics for reasons mentioned earlier. The other is McCabe's Cyclomatic Complexity, which is usually highly correlated with the other code metrics. This effect is not observed in our study, as the Cyclomatic Complexity has no correlation with a code metric above 0.24, and is most highly correlated with the logarithm of Henry and Kafura's metric, at 0.35.

4.5.3 Linear Regression Models

The intent is to create linear regression models to explain the variance in the comment quantity measures. This allows one to see how strongly the comment quantity measures are related to software complexity. The eight software metrics are used to create all $2^8 - 1$ possible models for a given comment quantity measure. Next, candidate models are chosen based on Mallow's C_p, a statistic that helps modelers make trade-offs between including too few variables in the model, called underspecifying the model, and including too many variables, called overspecifying the model [MALC 73]. Underspecifying the model introduces bias into the parameter estimates. Overspecifying the model introduces variances that are larger than the simpler model [MYER 86]. Finally, best candidate models are selected based on the partial t values associated with the parameter values for each of the software metric variables within the model. These partial t values are deviations from the mean of the Student t distribution. They indicate the probability that the parameter estimate is significantly different from zero. We looked for partial t values of 0.05 or less.

4.5.3.1 Comments

The software metrics are used to create linear regression models to account for the variance in comment quantity in each module. In general, the metrics are able to explain a substantial portion of the variance of comment quantity in each category (module, block, data, and code). Most of the parameter estimates are positive, suggesting that programmers are putting more comments in more complex modules. Negative parameters are caused by the use of parameters to offset each other. They do not imply that some software metrics suggest decreasing the amount of comments, as all significant correlations of metrics with comments are positive.

Of the four comment measures models, the model of data comment quantity is the strongest, with an R-Square of 0.6087 (see Table 14 on page 61). This is probably due to the tendency for programmers to add a brief comment description to every variable. Six out of seven of the teams had standards specifying this practice. The most influential variables in this model were Halstead's N and V, based on partial t values (see Appendix H).

Next in strength is the model of module comment quantity, with an R-Square of 0.4436. This is largely due to the influence of the data comment quantity component, as N and V have the strongest influence on the model (see Appendix F). The log of Henry and Kafura's metric has a stronger influence in this model than in the data comment quantity model.

The models for block comment quantity and code comment quantity are weaker, with R-Squares of 0.1951 and 0.2616 respectively. The log of Henry and Kafura's metric has the strongest influence on the block comment quantity model, based on partial t values (see Appendix G). Once again this reflects the fact that programmers list calling relationships and describe parameters in the block comment section, a measurement of the coupling between a module and its environment. McCabe's Cyclomatic Complexity has the strongest influence on the code comment quantity model (see Appendix I).

Table 14. By Module: R-square values for models of comment quantity

	Module	Block	Data	Code
R^2	0.4436	0.1951	0.6087	0.2616
F value	107.35	32.65	179.32	47.71
sig	0.0001	0.0001	0.0001	0.0001

The teams varied quite a bit in the amount of comment quantity variance explained (R^2) by linear models, as shown in Table 15 on page 63. Note that group nine, one of the teams with a detailed design, also has one of the highest amounts of comment quantity explained in each of the categories, except code comment quantity. Note also that most of the teams had higher R^2 values than the overall values. Each team seems to have somewhat different commenting styles. Dividing the analysis by teams tends to reduce some of this variance. Finally, note that for all but two of the teams (one and nine), the R^2 values for the block comment quantity model is the lowest of the three.

4.5.3.2 Change Log Data

The models of change log data are weaker than the models of comment data. These models of change log data are based on software metrics only. The models of the number of change logs, the number of lines changed, and the number of minutes spent changing them had R-Square values of 0.1698, 0.1294, and 0.1161 respectively, as shown in Table 16 on page 64. The low R-Square values suggest the inherent difficulty in predicting changes. All models were statistically significant. The models are shown in appendices J, K, and L.

Once again, the R^2 values varied widely among teams, as shown in Table 17 on page 65. Teams five and nine, which had the most detailed designs, also seem to have stronger models of change prediction based on software metrics. This may reflect the dedication of these teams, both to high quality designs and conscientious use of the logger.

The comment quantity measures can also be used as software complexity metrics. They can be added to multiple regression models in an attempt to use comments to aid in the prediction of change log data. The only change log model that is improved by the addition of of comment quantity variables is the model of number of logs. The R^2 value is improved to 0.1795. See

Table 15. By Module: R-square values for models of comment quantity, by team

Team	Module	Block	Data	Code
1	0.4517	0.4214	0.6078	0.2863
2	0.7442	0.3938	0.8677	0.4915
4	0.4820	0.1372	0.2962	0.7121
5	0.6496	0.3511	0.4603	0.8543
7	0.6855	0.3601	0.8044	0.5723
8	0.7776	0.2123	0.9521	0.4782
9	0.7273	0.4426	0.9541	0.3398

Table 16. By Module: R-square values for models of change log data

	Logs	Lines Changed	Minutes Spent
R^2	0.1698	0.1294	0.1161
F value	33.084	25.189	26.589
sig	0.0001	0.0001	0.0001

Table 17. By Module: R-square values for models of change log data, by team

Team	Logs	Lines Changed	Minutes Spent
1	0.1306	0.0783	0.0936
2	0.5653	0.5174	0.2312
4	0.3481	0.1918	0.3697
5	0.7599	0.4656	0.7716
7	0.1343	0.1802	0.2173
8	0.4618	0.4109	0.1471
9	0.5502	0.5177	0.4622

Appendix M for the specification of this model. Note that the block comment quantity has a positive parameter, while data comment quantity has a negative parameter.

4.6 Analysis by Change Log

4.6.1 Approach

A record is built to describe each of the 604 change logs in the study. Each change log description consists of 14 numbers:

- the number of lines changed
- the number of minutes taken changing them,
- the 8 software complexity measures
- the 4 comment measures.

4.6.2 Correlational Results

None of the software metrics or comment measures has a correlation with the number of minutes that is significant at the 0.05 level (see Table 18 on page 68, Table 19 on page 69) The decrease in significance of relationships, compared to the analysis by module, seems to be caused by the fact that not all of the modules are considered in the change log analysis. Only modules with at least one change are included in this analysis. These tend to be the

more complex modules, as the mean values for the the software metrics for change modules are higher than the mean values of all the modules (see appendix E). This suggests that software metrics can be used to divide modules into complexity classes, but within a class of roughly equivalent complexity, the metrics' ability to predict diminishes. Note that some of the modules occur more than once in the list of changed modules.

Lines of code, the Halstead metrics, and Woodfield's metric have low but significant correlations with the number of lines changed in the module, as shown in Table 20 on page 70. For the most part these correlations are similar in magnitude to the values in the analysis by module, with the exception of the correlation with Halstead's E (not significant in the analysis by module), as shown in Table 12 on page 57. McCabe's metric, McClure's metric, and Henry and Kafura's metric do not show significant relationships to the number of lines changed. In the analysis by module, the metrics may be predicting whether or not a change is made to a module, which allows McCabe's metric and Henry and Kafura's metric to have a significant relationship to the number of lines changed. Given the fact that one knows there was a change made to the module in the analysis by change log, the length metrics (LOC and Halstead's metrics) seem to have a stronger relationship to the number of lines changed. See Table 20 on page 70, and Table 21 on page 71 for these correlation coefficients.

The logarithm of Henry and Kafura's metric shows a slight positive (0.0712) correlation with the number of minutes spent during a log, significant at the 0.10 level. Code comment quantity shows a slight positive (0.0754) correlation with the number of minutes spent during a log, significant at the 0.10 level. Block comment quantity shows a slight negative (-0.0693) correlation with the number of lines changed during a log, significant at the 0.10 level.

Table 18. By Log: Correlation of software metrics with number of change minutes

	LOC	N	V	E	CC	WOOD	MC	LOGHK
R	0.0212	0.0089	0.0205	-0.0061	0.0548	-0.0047	-0.0472	0.0692
sig	0.6052	0.8294	0.6178	0.8827	0.1817	0.9094	0.2523	0.0915

Table 19. By Log: Correlation of comment quantity with number of change minutes

	Module	Block	Data	Code
R	0.0413	-0.0275	0.0413	0.0754
sig	0.3100	0.5000	0.3107	0.0642

Table 20. By Log: Correlation of software metrics with number of lines changed

	LOC	N	V	E	CC	WOOD	MC	LOGHK
R	0.1462	0.1282	0.1228	0.1116	0.0193	0.1159	-0.0231	-0.0335
sig	0.0003	0.0017	0.0027	0.0064	0.6383	0.0046	0.5759	0.4139

Table 21. By Log: Correlation of comment quantity with number of lines changed

	Module	Block	Data	Code
R	-0.0225	-0.0693	-0.0036	0.0450
sig	0.5803	0.0888	0.9306	0.2700

4.6.3 Linear Regression Results

The approach is to create a linear model to predict the number of minutes spent on a log, based on the number of lines changed and all of the software metrics. Each of the comment quantity measures can then be added to the model. The unique contribution of the comment quantity measure, the variance that is uncorrelated with all of the other variables in the model, can then be isolated. If the parameter of the comment quantity measure is negative, and the partial t value is significant, then there exists some support for the notion that greater comment quantity within a module implies that the module is easier to change.

This condition never occurred, either for the body of change data as a whole (see Table 22 on page 73), or divided by team (see Table 23 on page 74). Therefore, we found no empirical support for the value of comments. This could have been due to the high correlation of the comment quantity measures to the other software metrics. Their unique contribution may be too small to be detected by the statistical methods being used. Naturally, it could also be due to a lack of value of comments in the environment being studied.

4.7 Conclusion

One sees the influence of module complexity on comment quantity throughout the data. Most of the software metrics have significant positive relationships to comment quantity in a module. The linear models are able to account for a sizable portion of the variance in comment quantity in each module, especially within some of the project teams.

However, in general, the relationship of software metrics to block comment quantity is the weakest. Shneiderman's study [SHNB 77] suggests that this is where comments are most

Table 22. By Log: Parameter, Partial t, and significance values for comment quantity

	Module	Block	Data	Code
param	-0.0140	-0.0414	-0.00339	0.0635
t	-0.586	-1.284	-0.082	1.048
sig	0.5581	0.1997	0.9349	0.2952

Table 23. By Log: Parameter, Partial t, and significance values for comment quantity, by team

Team	Module	Block	Data	Code
1 param	0.0430	0.1209	-0.1022	-0.0988
1 t	1.215	2.904	-1.164	-1.291
1 sig	0.2276	0.0047	0.2476	0.2005
2 param	-0.0137	0.1859	-0.0741	-0.1809
2 t	-0.129	1.057	-0.383	-0.973
2 sig	0.8976	0.2930	0.7027	0.3327
4 param	0.1311	0.1126	-0.1187	0.2849
4 t	2.265	1.555	-0.849	2.601
4 sig	0.2368	0.1247	0.3987	0.0114
5 param	-0.0371	-0.0546	-0.1299	0.0679
5 t	-0.674	-0.731	-1.213	0.769
5 sig	0.5023	0.4673	0.2287	0.4443
7 param	0.0156	-0.0337	-0.2129	1.4592
7 t	0.083	-0.168	-0.516	2.082
7 sig	0.9343	0.8683	0.6107	0.0482
8 param	0.0382	-0.1524	-0.2185	0.3562
8 t	0.140	-0.310	-0.370	0.798
8 sig	0.8890	0.7578	0.7125	0.4278
9 param	-0.0176	-0.0304	0.1623	-0.0321
9 t	-0.446	-0.460	0.776	-0.338
9 sig	0.7301	0.6467	0.4396	0.7364

useful. Based on casual viewing of the source code, one rarely sees more than two lines devoted to actual description of the module. The variance in block comment quantity is mainly related to the number of called and calling modules listed, and the number of parameters described.

A significant beneficial effect of having greater comment quantity in a module is never observed. Since comment quantity is positively correlated with most of the software metrics, one would expect to frequently see more comment words in the more troublesome modules. However, after correcting for module complexity there were no positive effects of more comments over fewer, either in terms of shorter times to correct a module or fewer corrections to a module. The idea that more internal documentation is generally better is not supported.

5.0 Chapter 5: Conclusions

5.1 Analysis of Results

Before the study was complete, we hypothesized that students would put lengthier internal documentation in the parts of their code that they understood; they would comment the simpler modules heavily, leaving the more difficult ones with less of an explanation. Another hypothesis was that students would take a "minimal acceptable behavior" approach, adding a fixed number of comments to each module. Neither of these hypotheses were supported. Students actually did put more comments in more complex parts of their programs. Comment quantity correlates more highly with the more sophisticated metrics such as Henry and Kafura's, than with lines of code.

Unfortunately, there was no evidence for the value of these well-placed comments. No advantage (or disadvantage) ever appeared for having more comments in a module. There are a number of reasons for this lack of a significant result.

First, the adherence to commenting standards was weak. An example of this can be seen in Appendix N, a block comment from group nine. It does not list the calls relationships, required by group nine. It does list the programmer's name, not required by group nine. Some program debuggers could have easily gotten discouraged after finding key information missing, and given up on reading comments. A comment can't help if it is not read.

Second, many of the errors may have been low-level logic errors, rather than semantic errors. Comments may not be of much use for such errors. Since most of the logs had short times (an average time of 19 minutes to make a modification), it does not seem that the subjects were obtaining a deep understanding of the program before making the change.

Third, the changes were made by people who were already familiar with the module. All the changes were made either by the program designers or the program coders. This means that comments were not used to gain a general understanding of the program, or to figure out the overall design of the program. Perhaps this is the main contribution that comments can make.

Fourth, the block comment quantity has the weakest relationship to module complexity. The block comments may have been superficial; they may have been more a reflection of programmers mechanically following the commenting rules rather than actually conveying genuine insight. The comments may have been merely "echoing the code" [KERB 78a].

Finally, the subjects may not be proficient in their task. They may not have had much experience modifying code written by others, and therefore used unsystematic strategies for finding the errors. They probably did not have experience working with the exact programming standards that were in effect. Perhaps programmers can exploit the benefits of programming standards only after they have considerable experience working with them.

In summary,

- Software complexity metrics can account for a large portion of the variance in comment quantity.

- The relationship between software complexity metrics and comments may be due more to programmers mechanically following commenting rules than actually explaining the functioning of their programs.

- For low-level debugging in a high level language by programmers familiar with the modules concerned, the value of comments is small or nonexistent.

5.2 *Future Work*

Comments may be more important in a lower level language such as assembly language or even 'C'. The study should be repeated using such languages.

Programmers have different attitudes toward the value of comments. Though we measured comments by counting words, the value of the words vary with the effort that the programmer put into them. In any future work, the name of the programmer who wrote each module should be recorded, to measure these differences.

In this study we were unable to measure the readability of comments using traditional text analysis algorithms. This implies that the grammar and style of comments is not English prose. It would be useful to develop a theory of commenting which could lead to a better understanding of the readability of comments.

The program integrators/debuggers/maintainers in this study were all familiar with the code. The value of comments might show more clearly if the people doing the integration and debugging were not the same people as the designers or coders.

Finally, comments did not prove useful in the task that we measured. A study is needed to determine the kinds of problems that can be helped by well-written comments, and the types of comments that can be of use in such situations. This would allow comment writers to concentrate on their intended audience, minimizing the time they spend and maximizing the value of their work.

Bibliography

[ANNJ 79] Annino, J. S. and E. C. Russell, "The ten most frequent causes of simulation analysis failure - and how to avoid them!", *Simulation*, 32(6), pp. 137-140, 1979.

[ARNK 88] Arnold, K., "C Advisor: Debugging Tricks for Butterfingered Programmers", *UNIX Review*, Vol. 6, No. 6, June, 1988, pp. 80-85.

[ARTL 85] Arthur, L. J., *Measuring Programmer Productivity and Software Quality*, John Wiley & Sons, New York, 1985.

[BASV 83] Basili, V.R., R.W Selby, and T.Y. Phillips, "Metric Analysis and Data Validation Across Fortran Projects", *IEEE Transactions on Software Engineering*, Vol. SE-9, No. 6, Nov. 1983, pp.652-663.

[CONS 86] Conte, S. D., H. E. Dunsmore, and V. Y. Shen, *Software Engineering Metrics and Models*, The Benjamin/Cummings Publishing Company, Inc., Menlo Park, California, 1986.

[COSS 84] Costello, S. H., "Software Engineering Under Deadline Pressure.", *ACM SIGSOFT Software Engineering Notes*, Vol. 9, no. 5 (October 1984), pp. 15-19.

[DRAN 81] Draper, N. R., and H. Smith, *Applied Linear Regression*, John Wiley & Sons, Inc., New York, New York, 1981.

[DUNH 85] Dunsmore, H. E., "The Effect of Comments, Mnemonic Names, and Modularity: Some University Experiment Results", from *Empirical Foundations of Information and Software Science* edited by Agrawal, J. C., and Zunde, P., Plenum Press, New York, 1985, pp.189-196.

[EDWA 84] Edwards, A. L., *An Introduction to Linear Regression and Correlation, Second Edition*, W. H. Freeman and Company, 1984.

[FEID 84] Feinberg, D. A., "One Day in the Programming Department...",*Computer*, November 1984, 17 (11), p. 94.

[FJER 79] Fjeldstad, R. K., and W. T. Hamlen, "Application Program Maintenance Study-Report to Our Respondents", IBM Corporation, DP Marketing Group, 1979, reprinted in *Tutorial*

on Software Maintenance, Girish Parikh and Nicholas Zvegintzov, editors, IEEE Computer Society Press, Silver Spring, Maryland, 1983.

[GUNR 52] Gunning, R., *The Technique of CLEAR WRITING*, McGraw-Hill Book Company, Inc., New York, 1952.

[HALM 77] Halstead, M. H., *Elements of Softare Science*, Elsevier North-Holland, Inc., New York, 1977.

[HENS 81a] Henry, S., and D. Kafura, "Software Structure Metrics Based on Information Flow", *IEEE Transactions on Software Engineering*, Vol SE-7, September, 1981.

[HENS 81b] Henry, S. J., D. Kafura, and K. Harris, "On the Relationships Among Three Software Metrics", *Performance Evaluation Review*, Vol. 10, No. 1, 1981.

[HENS 83] Henry, S., "A Project Oriented Course on Software Engineering", *Proceedings of the 1983 ACM SIGCSE Symposium on Computer Science Education*, 1983, pp. 57-61.

[HENS 88] Henry, S., "A Technique for Hiding Proprietary Details While Providing Sufficient Information for Researchers; or Do You Recognize This Well-Known Algorithm?", *The Journal of Systems and Software*, Vol. 8, No. 1, January 1988, pp.3-11.

[HOLR 86] Holt, R., D. Boehm-Davis, and A. Shultz, "Mental Representations of Programs for Student and Professional Programmers", Psychology Department, George Mason University, Fairfax VA 22030.

[KERB 78a] Kernighan, B. W. and P. J. Plauger, *The Elements of Programming Style*, McGraw-Hill Book Company, New York, New York, 1978.

[KERB 78b] Kernighan, B. W. and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1978.

[KRIB 80] Krieg-Brucker, B., and D. C. Luckham, "ANNA: towards a language for annotating Ada programs", *Proc. ACM, SIGPLAN Symposium on the Ada Programming Languages, ACM SIGPLAN Notices*, November 1980, 15 (11), pp.128-138.

[LUIC 68] Lui, C. L., *Introduction to Combinatorial Mathematics*, McGraw-Hill Book Company, New York, 1968.

[MALC 73] Mallows, C. L., "Some Comments on C_p", *Technometrics*, Vol 15, No. 4, November 1973, pp. 661-675.

[MARJ 83] Martin, J., and C. McClure, *Software Maintenance: The Problem and Its Solutions*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1983.

[MCCC 78] McClure, C. L., "A Model for Program Complexity", *Proceedings: 3rd International Conference on Software Engineering*, May 1978, pp. 149-157.

[MCCT 76] McCabe, T. J., "A Complexity Measure", *IEEE Transactions on Software Engineering*, Vol. SE-2, No. 4, December 1976.

[MYER 86] Myers, R. H., *Classical and Modern Regression with Applications*, PWS Publishers, Boston, Massachusetts, 1986.

[PFLS 87] Pfleeger, S. L., *Software Engineering, The Production of Quality Software*, Macmillan Publishing Company, New York, 1987.

[SASB 85] *Sas Users Guide: BASICS, Version 5 Edition*, SAS Institute Inc., Cary North Carolina, 1985.

[SASS 85] *Sas Users Guide: STATISTICS, Version 5 Edition*, SAS Institute Inc., Cary North Carolina, 1985.

[SELC 87] Selig, C., *ADLIF - A Structured Design Language for Metric Analysis*, Masters Thesis, 1987.

[SHEB 83] Sheil, B. A., "The Psychological Study of Programming", *Computing Surveys*, Vol. 13, No. 1, March 1981, pp. 110-120.

[SHES 79] Sheppard, S., B. Curtis, P. Milliman, and T. Love, "Modern Coding Practices and Programmer Performance", *Computer*, December 1979, pp.41-49.

[SHNB 77] Shneiderman, Ben, "Measuring computer program quality and comprehension", *International Journal of Man-Machine Studies*, 9, 1977, pp. 465-477.

[SHNB 80] Shneiderman, B., *Software Psychology : Human Factors in Computer and Information Systems*, Winthrop Publishers, Inc., Cambridge, Massachusetts, 1980.

[SMIE 70] Smith, E. A., and P. Kincaid, "Derivation and Validation of the Automated Readability Index for Use with Technical Materials", *Human Factors*, 12, 1970, pp. 457-464.

[STEW 74] Stevens, W. P., G. J. Myers, and L. L. Constantine, "Structured design", *IBM Systems Journal*, Vol. 13, No. 2, 1974.

[WEIG 71] Weinberg, G., *The Psychology of Computer Programming*, D. Van Nostrand Co., 1971.

[WOOS 77] Wooldridge, S., *Systems and Programming Standards*, Petrocelli/Charter, New York, 1977.

[WOOS 80] Woodfield, S. N., "Enhanced Effort Estimation by Extending Basic Programming Models to Include Modularity Factors", *Ph.D. Dissertation*, Purdue University, Computer Science Department, 1980.

[WOOS 81] Woodfield, S. N., H. E. Dunsmore, and V. Y. Shen, "The Effect of Modularization and Comments on Program Comprehension", *Proceedings of the Fifth International Conference on Software Engineering*, (March 1981) pp. 215-223.

[YOUE 75] Yourdon, E., *Techniques of Program Structure and Design*, Englewood Cliffs, New Jersey: Prentice-Hall 1975.

Appendix A. Programming Standards

Group One	: Pascal_Based_Editor
Steve Kramer	: 552-9038
Adil Rasheed	: 951-0238
Joan Trimbur	: 552-7782/951-0770

Programming Standards

All programmers must follow the programming standards given below. Any deviation from these standards will result in deduction of points (as described in programming rules).

I. DOCUMENTATION

1. Each module must have a header.

Programmer	: your name
Module Name	: as on the contract
Test Cases	: specify
Comments/Suggestions/Problems ...	

2. All Procedures/Functions must have header documentation.

Programmer	: your name.
Purpose	: what does it do?
Algorithm	: how does it work?
Input	: document input variables
Output	: document output variables
Calls	: what procs/funcs called?
Called by	: what procs/funcs call it?

3. Describe the purpose of variables used.

4. Add in-line documentation with blocks of code.

5. Indent (three spaces) each block of code. Be consistent!

II. IDENTIFIER USAGE

1. Procedure, function, variable names must be the same as in our ADLIF. If a new identifier name is used enter the following comment:

```
newvar (* NEW VARIABLE *)
```

2. Get team permission before using a new global variable.

Figure 3. Team 1 Programming Standards

Figure 4. Team 1 Programming Standards: Page 2

III. PROGRAMMING LANGUAGE/SYSTEM

All code must compile and run on VM PASCAL. You may work on any system but it's *your responsibility to convert all code to VM PASCAL* Therefore, use standard pascal !!!

Note: No strings, no '_', pointer is @ on VM, need filedef for I/0.

IV. TESTING

1. You must test your module and provide us with the test data.

2. Try to scedual [sic] one walkthroug [sic] per week. You must have at least one walkthrough with our team.

3. Send mail to CS498101 if you have design problems during testing.

V. LOGGED CHANGES

You must log all the changes during the coding phase. *You*

will lose points for not logging changes.

VI. LATE POLICY

ALL CODE MUST BE MAILED TO CS498101 BEFORE MIDNIGHT ON

THE DUE-DATE.

ONE DAY LATE	: 50 %
TWO DAYS LATE	: 0 %
AFTER TWO DAYS	: -100 %

GOOD LUCK !

Figure 5. Team 2 Programming Standards

Programmer Rules:

Documentation/Format of Code:

1. Actual code for a procedure should not exceed 60 lines. If more lines are needed, create new procedures and nest them inside the long procedure. (The same holds for functions)

2. Indent two spaces for a new level of indentation.

3. Use (* and *) for comments.

4. Put the expression after the corresponding 'end'.
   EXAMPLE:

```
while (i<0) do begin
 |
 |
end; (* i<0 *)
```

5. Follow formats on the next page for lining up constructs.

6. The following must be in the procedure header:

   Name, Phone #, Name of Procedure. Description of Procedure, Parameters, What the Procedure calls.

7. Line-document any variables.

Design Related:

8. Names of Procedures/Functions must correspond to those in the design. (See Rule 1 if length is beyond 60 lines.)

9. Names of Variables/Parameters must correspond to those in the design.

General

10. LATE POLICY: 1 day late -1/2 off; 2 days -0 points; any later- (-n) points.

11. Programmers should expect/take calls from 8am-2am (slightly shorter on the weekends) concerning code.

12. All graphics procedures are subject to approval before due date.

Figure 6. Team 2 Programming Standards: Page 2

13. Constants should be used inside procedure for often used numbers/symbols.

14. 'Graphics.pas' will be used for graphics and random number generator.

15. Changes to types/parameters/variables must be cleared.

16. The logger must be used for changes to code/design.

17. Programmers will sign a contract consenting to these rules.

18. There may be additional rules/requirements for specific modules.

Constructs:

```
If (exp)
then begin
|
end
else begin
|
end;
```

```
If (exp)
then
|
else if (exp2)
        then begin
        |
        end;
```

```
While (exp) do begin
|
end;
```

```
For i:=1 to x do begin
|
end;
```

```
case (variable) of
|
|
end; (* case *)
```

```
[sometype]=record
          |
          end;
```

Figure 7. Team 4 Programming Standards

Programmer Specifications:

I. Coding Requirements:

1. All code must be written in VAX-11 Pascal.
2. All changes must be logged. Failure to comply will result in a maximum of a 25% point penalty.
3. All variable names must be meaningful.
4. Each constant, type, and variable declaration must be accompanied by a short explanation of its purpose.
5. Use form feeds before each procedure.
6. All procedures must be no more than one page in length, whenever possible.
7. Use of Goto's or global variables will not be accepted except where specified by design.
8. The bodies of all procedures and functions must be followed with a comment giving the name of the procedure or function.
9. Consistent indentation and blank lines must be used for clarity.
10. All procedures and functions should be thoroughly tested by the author (see testing policies).
11. Author's name must be at the top of each module for which he contracted.
12. Any deviation from the adlif design must be first consulted with the designer. (This does not include expanding code.) Procedures and functions must be named at they were in adlif.
13. Each procedure and function must contain the following documentation in a header;

Purpose	: purpose of the routine
Input	: description of input parameters
Output	: description of output parameters and screen output
Calls	: listing of all routines called by this routine
Called by	: listing of all routines which call this routine

14. Final modules must be sent to group 4's account, CS498104, by midnight of the agreed due date. A hard copy must be delivered to a group member the next day.

15. All input/output must be performed using the direct I/O functions given in the file graphics.pas.

Figure 8. Team 4 Programming Standards: Page 2

II. Testing Requirements

1. Walkthroughs will be used to verify interim progress and the final program correctness.

2. Programmers will interactively test for all specific error cases given in the design.

III. Late Policy

Unless approval from a group member is given ahead of time, the following late policy will hold.

One day late	: a maximum of 50% of the points will be awarded
Two days late	: no points will be awarded
Three days late	: 75% negative points will be awarded
Four days late	: 100% negative points will be awarded

Figure 9. Team 7 Programming Standards

PROGRAMMER RULES

1) Submission

Each module contracted must be sent to account CS498107 on VAX 3/5 by midnight of the specified completion date for full credit to be awarded.

Modules submitted on (1) day late will be worth 1/2 credit.

Modules submitted two (2) days late will be worth 1/4 credit.

Modules submitted more that two (2) days late will receive negative credit.

Saturday and Sunday each count as a separate day.

Modules that are submitted ahead of their completion dates and meet the design specifacations [sic] will receive extra credit.

2) Progress Reporting

Progress should be reported to one of the designers twice a week. Satisfactory progress is a condition of employment.

3) Documentation Requirements

Each procedure/function must be contain [sic] a header block comment containing the following information:

a) Programmer Name
b) Date
c) Procedure Name
d) Description of what the procedure does
e) Description of each parameter
f) The names and purposes of any files that are accessed
g) Description of any VAX/Pascal Extensions used
h) What external modules must be linked to run the procedure
i) Description of any assumptions made
j) List of procedures that are called
k) List of calling procedures (say MAIN for main program)

All local variables must be accompanied by a comment describing their uses in the procedure.

Type and constant declarations must also be accompanied by a comment describing their use.

Figure 10. Team 7 Programming Standards: Page 2

In-line comments should be used whenever the purpose of a code statement is not blatantly obvious. Comments describing a section of code should be placed in blocks before the code. In-line comments should be placed to the right of the statement they describe. Use the next line if the comment won't fit to the right of the statement.

4) Indentation, Nesting, Etc.

Modules should have proper indentation to indicate nesting of statements.

Procedures that are called only within another procedure should be nested inside that procedure.

Use white space to separate sections of code.

Put a form-feed in front of each procedure. Procedures should fit on one page.

5) Don'ts

a) Don't use global variables
b) Don't use GOTOs.
c) Don't change the names of procedures or functions if they are specified in the ADLIF design.

6) Testing

Modules should be thoroughly tested. Programmers should be able to show the designers some type of written test plan. We expect multiple condition coverage for each module.

7) Logging of Changes

To receive full and possibly additional credit all changes made should be logged using Cal's LOGGER program.

8) Designers

If you have questions, concerns, ideas, etc., please contact one of use in class or by calling before midnight (unless it is crucial). We are:

Steve Clark	951-8944
Bob Francis	951-8775 or 953-2880
Zoran Mladen	961-2221

Figure 11. Team 8 Programming Standards

BSR ASSOCIATES (a.k.a GROUP 8)

PROGRAMMING STANDARDS

All programmers for BSR Associates must conform to the following programming standards:

1. No variable/procedure/function names may be changed--you MUST use the names that have been specified in ADLIF.

2. If a module has not been completely specified (i.e. you need to create new procedures/functions within the module), the programmer must nest the unspecified procedure(s)/functions(s) within the existing module. The new procedure/module name must be unique--check the ADLIF code to make sure you are not using a name that is already in use.

3. Each control statement (WHILE DO, REPEAT UNTIL, FOR LOOP, CASE STATEMENT, IF THEN (ELSE)) must have a comment preceding it explaining the purpose.

4. All declarations must be commented.

5. Each procedure/function must have a header in the following format:

```
(*************************************************************)
(*                                                           *)
(* NAME:                                                     *)
(*                                                           *)
(* DESIGNED BY:                                              *)
(*                                                           *)
(* DESCRIPTION:                                              *)
(*                                                           *)
(* CALLED BY:                                                *)
(*                                                           *)
(* CALLS:                                                    *)
(*                                                           *)
(* MODIFIER NAME    DATE      COMMENTS                       *)
(* ______________   ____   ____________________             *)
(*                                                           *)
(*************************************************************)
```

6. Each parameter in the procedure/function declaration must be commented.

```
ex.  FOO (VAR Larry : INTEGER;          (* stooge #1 *)
          Moe   : CHAR;                 (* stooge #2 *)
          Curly : BOOLEAN);             (* stooge #3 *)
```

Figure 12. Team 8 Programming Standards: Page 2

7. All changes must be logged in LOGGER as well as noted in the procedure heading--this will count for 10% of points awarded.

8. 10% of points awarded will be based on participation in walkthroughs, as specified per module.

9. The programmer must meet with the designer of his/her module to discuss design specifications before s/he starts coding.

10. LATE POLICY: 1 day (up to 24 hours) late: 50% off
    2 days (24 - 48 hours) late: 0%
    Anytime later : -100%

11. TESTER ONLY: Test cases must be determined from specifications given. Degree of testing will be specified for each module.

12. Bonus points will be awarded for creativity--i.e. if your code makes our design better, we will give you extra points.

****

13. All programmers who abide by the above specifications are invited to Betsy's for an attitude-adjustment hour (a.k.a. PARTY!!!!!) the last week of class--we provide the necessary beverages.

Figure 13. Team 9 Programming Standards

Programmer Rules

1. All code must be written using standard PASCAL except when using graphics or files.

2. Each procedure/function should have a header using the following format:

```
(*************************************************************************

 Procedure/Function name:

 Description:

 Returned function value: (functions only)

 Parameters:

 Calls:
 *************************************************************************)
```

All local variables must be documented in line.

3. Full credit is awarded for modules that meet the specifications. Partial credit may be awarded for code that does not meet design specifications completely.

4. The programmer is required to use the logger.

5. Each programmer is required to schedule a minimum of one walkthrough per module. The walkthrough must be at least three days before a module is due.

6. Late policy:
   50% is deducted for modules 1 day late.
   0 credit is given for modules 2 days late.
   There is a negative penalty for the amount contracted for modules not delivered 2 days after the assigned date.

7. If no progress is made within one week of when the programmer receives the specifications we reserve the right to withdraw the module with no points awarded.

8. Modules are due at midnight of the due date specified.

9. Code must follow the specifications laid out by the adlif code. No changes can be made without first consulting the assigned designer.

Appendix B. Within Comments: Correlations of Non-Blank Characters with Number of Words

Table 24. Correlations of Non-Blank Characters with Number of Words

	Module	Block	Data	Code
R	0.9878	0.9776	0.9962	0.9917
sig	0.0000	0.0001	0.0001	0.0001

Appendix C. Correlations of Comment Words, With and Without * and -

Table 25. Correlations of Comment Words, With and Without '*' and '-'

	Module	Block	Data	Code
R	0.9874	0.9646	0.9979	0.9998
sig	0.0001	0.0001	0.0001	0.0001

Appendix D. Correlations of comment quantity measures with each other

Table 26. Correlations of comment quantity measures with each other

	Module	Block	Data	Code
Module sig	1.0000 0.0001	0.6965 0.0001	0.7142 0.0001	0.5170 0.0001
Block sig		1.0000 0.0000	0.1929 0.0001	0.1097 0.0015
Data sig			1.0000 0.0000	0.0822 0.0173
Code sig				1.0000 0.0000

Appendix E. Mean software metric values for all modules and error modules

Table 27. Mean software metric values for all modules and change modules

	LOC	N	V	E	CC	WOOD	MC	LOGHK
All	24.7	192	966	223,635	6.62	240,543	15.4	7.48
Change	42.6	359	1865	712,141	10.1	748,373	17.0	9.81

Appendix F. By Module: Multiple regression model of module comment quantity

Table 28. By Module: Multiple regression model of module comment quantity

	Intercept	LOC	N	V	E	CC	LOGHK
Param	23.74	0.6471	-0.4663	0.0942	1.1×10^{-5}	0.9288	3.6862
t	4.622	2.737	-7.353	8.940	2.615	3.464	5.836
Sig	0.0001	0.0063	0.0001	0.0001	0.0091	0.0006	0.0001

Appendix G. By Module: Multiple regression model of block comment quantity

Table 29. By Module: Multiple regression model of block comment quantity

	Intercept	LOC	N	CC	WOOD	MC	LOGHK
Param	21.07	-0.549	0.0926	0.5400	-7.06×10^{-6}	0.0672	3.171
t	7.154	-3.735	4.045	3.167	-3.448	2.620	7.948
Sig	0.0001	0.0002	0.0001	0.0016	0.0006	0.0090	0.0001

Appendix H. By Module: Multiple regression model of data comment quantity

Table 30. By Module: Multiple regression model of data comment quantity

	Intercept	LOC	N	V	E	CC	WOOD	LOGHK
Param	1.151	1.502	-0.6872	0.1017	5.7×10^{-5}	-.6817	-2.7×10^{-5}	0.7271
t	0.454	13.045	-22.101	19.657	6.426	-5.226	-3.007	2.355
Sig	0.6498	0.0001	0.0001	0.0001	0.0001	0.0001	0.0027	0.0187

Appendix I. By Module: Multiple regression model of code comment quantity

Table 31. By Module: Multiple regression model of code comment quantity

	Intercept	LOC	N	V	CC	WOOD	MC
Param	-1.965	-0.191	0.173	-0.0166	1.141	-1.50×10^{-5}	0.0388
t	-1.306	-1.716	5.893	-3.448	9.123	-7.163	-2.118
Sig	0.1918	0.0865	0.0001	0.0006	0.0001	0.0001	0.0345

Appendix J. By Module: Multiple regression model of change logs

Table 32. By Module: Multiple regression model of change logs

	Intercept	LOC	N	V	MC	LOGHK
Param	-0.0983	-0.0127	-0.0025	9.99×10^{-4}	-0.00214	0.09024
t	-0.894	-2.464	-5.361	6.465	-2.346	6.122
Sig	0.3715	0.0139	0.0001	0.0001	0.0192	0.0001

Appendix K. By Module: Multiple regression model of changed lines

Table 33. By Module: Multiple regression model of changed lines

	Intercept	N	E	WOOD	MC	LOGHK
Param	-1.243	0.0151	-1.76×10^{-5}	1.53×10^{-5}	-0.0139	0.436
t	-1.443	5.707	-5.123	4.366	-1.886	3.797
Sig	0.1495	0.0001	0.0001	0.0001	0.0596	0.0002

Appendix L. By Module: Multiple regression model of total minutes spent making changes

Table 34. By Module: Multiple regression model of total minutes spent making changes

	Intercept	N	V	MC	LOGHK
Param	-5.43	-0.0706	0.0196	-0.0523	1.922
t	-1.762	-5.326	5.620	-2.015	4.543
Sig	0.0784	0.0001	0.0001	0.0442	0.0001

Appendix M. By Module: Multiple regression model of error logs including comment quantity "metrics"

Table 35. By Module: Multiple regression model of error logs including comment quantity "metrics"

	Intercept	N	V	E	MC	LOGHK	BCQ	DCQ
Param	-0.192	-0.00650	0.00145	2.02×10^{-7}	-0.00252	0.0798	0.00409	-0.00521
t	-1.606	-3.936	5.032	1.928	-2.780	5.265	3.344	-3.589
Sig	0.1088	0.0001	0.0001	0.0542	0.0056	0.0001	0.0009	0.0004

Appendix N. Block Comment from a Team 9 Procedure, Not Conforming to Standards

Figure 14. Block Comment from a Team 9 Procedure, Not Conforming to Standards

```
{*****************************************************************}
{                                                                 }
{                                                                 }
{   COMPUTER_MOVE                                                 }
{                                                                 }
{   programmed by Jay Clayton                                     }
{   223-15-7592                                                   }
{   951-2837                                                      }
{                                                                 }
{   This routine has the computer decide on a move                }
{   given the board, stat of the board, the dice                  }
{   roll and the strategy determined by POINTS.  2                }
{   or 4 moves are passed back depending on whether               }
{   doubles were rolled.  DETERMINE_MOVES is called               }
{   to get all possible legal moves.  EVALUATE_MOVES              }
{   is called to determine the "best" move.  Note:                }
{   the best move is determined by the point system               }
{   defined in POINTS.  Doubles are processed by                  }
{   determining two separate sets of moves.  This is              }
{   done because of speed problems with processing                }
{   all possible combinations of moves.                           }
{                                                                 }
{   PARAMETERS                                                    }
{         DIE1,DIE2       dice roll                               }
{         MY_stat       stat record for player                    }
{                       moving                                    }
{         OPP_stat      stat record for                           }
{                       opponent                                  }
{         MY_BOARD        Board of player moving                  }
{         OPP_BOARD       Board of opponent                       }
{         POINTS          Point record for                        }
{                       strategy                                  }
{         MOVE            array of moves as output                }
{                                                                 }
{                                                                 }
{*****************************************************************}
```

Appendix O. Correlations of software complexity metrics with each other

Table 36. Correlations of software complexity metrics with each other

	LOC	N	V	E	CC	WOOD	MC	LOGHK
LOC	1.00	0.972	0.983	0.891	0.175	0.895	-0.012	0.184
N		1.00	0.980	0.960	0.142	0.962	0.003	0.120
V			1.00	0.892	0.236	0.8960	0.016	0.214
E				1.00	-0.005	0.999	-0.013	0.002
CC					1.00	0.001	0.138	0.346
WOOD						1.00	-0.011	0.008
MC							1.00	0.210
LOGHK								1.00

Vita

Wilson Kirt Gibbins was born in Mount Gilead, Ohio on January 2, 1957. While attending Barboursville High School, he won the West Virginia High School Chess Championship three times and graduated as tri-valedictorian of his class.

Like his uncle before him, Wilson went to college seeking enlightened conversation about philosophy, religion, and the future of the world. Like his uncle before him, he discovered that none of his roommates had similar interests, as they generally concerned themselves with drinking beer and chasing women. Like his uncle before him, Wilson eventually conceded that his roommates had a good point. Eventually, at The Ohio State University, he met and later married Ellen Silva, a chemical engineer who he hoped could support him in the manner to which he was accustomed. Along the way to matrimonial bliss, Wilson picked up a B.S. in Psychology, Summa Cum Laude, an M.B.A. with an emphasis in Finance, and a Phi Beta Kappa key.

In 1983, Wilson's chemical engineer became a mother, making him the father of Karen Jayne. Having a baby wasn't challenge enough, so in 1984 Wilson embarked on a stormy voyage through the computer science department at Virginia Polytechnic Institute and State Univer-

sity. Having one child while in graduate school was not enough of a challenge for Wilson, and so in 1985, with the help of his wife, he added Jennifer Marie to his family.

When Jenny started eating in quantity, Wilson realized he had to make a serious effort to complete a degree so he could buy chess books as well as food. Therefore, he came to Dr. Sallie Henry for guidance, and in August 1988 completed the requirements for a Master's degree in computer science.

Wilson is now employed by James Madison University as a microcomputer consultant. He intends to begin spending more than one evening a week with his family.

Wilson K. Gilbins