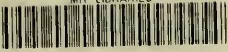
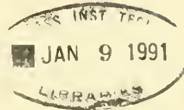

MIT LIBRARIES
3 9080 00701567 7

MASSACHUSETTS
INSTITUTE of TECHNOLOGY
LIBRARIES

DEWEY

HD28
.M414
no. 3222-
90



CYCLOMATIC COMPLEXITY METRICS REVISITED: AN EMPIRICAL STUDY OF SOFTWARE DEVELOPMENT AND MAINTENANCE

Geoffrey K. Gill
Chris F. Kemerer

October 1990

CISR WP No. 218
Sloan WP No. 3222-90

Center for Information Systems Research

Massachusetts Institute of Technology
Sloan School of Management
77 Massachusetts Avenue
Cambridge, Massachusetts, 02139

CYCLOMATIC COMPLEXITY METRICS REVISITED: AN EMPIRICAL STUDY OF SOFTWARE DEVELOPMENT AND MAINTENANCE

Geoffrey K. Gill
Chris F. Kemerer

October 1990

CISR WP No. 218
Sloan WP No. 3222-90

©1990 G.K. Gill, C.F. Kemerer

Center for Information Systems Research
Sloan School of Management
Massachusetts Institute of Technology

[illegible] [illegible]RIES
JAN 9 1990

Abstract

While the need for software metrics to aid in the assessment of software complexity for both development and maintenance has been widely argued, little agreement has been reached on the appropriateness and value of any single metric. McCabe's cyclomatic complexity metric, a measure of the maximum number of linearly independent circuits in a program control graph has been widely used in research. However, despite the widespread interest and popularity of this metric, it has not been without criticism, both analytical (the Myers and Hansen variants) and empirical (the high correlation of cyclomatic complexity with size measures). The current research tested both types of critiques on a newly collected dataset of real world software development and maintenance projects. The analytical research questions were tested on a set of 834 software modules from a number of existing real-time systems. Neither the Myers nor Hansen variants were found to be significantly different from the original value as computed by McCabe. Therefore, these particular theoretical criticism seem to have little or no practical impact, as represented by the data collected in this study. In regard to the empirical research questions, previous concerns were validated on this new dataset. However, the current research proposes a simple transformation of the metric whereby the cyclomatic complexity is divided by the size of system in source statements, thereby determining a "complexity density" ratio. This complexity density ratio is demonstrated to be a useful predictor of software maintenance productivity on a small pilot sample of actual maintenance projects.

CR Categories and Subject Descriptors: D.2.7 [Software Engineering]: Distribution and Maintenance; **D.2.8 [Software Engineering]**: Metrics; **D.2.9 [Software Engineering]**: Management; **F.2.3 [Analysis of Algorithms and Problem Complexity]**: Tradeoffs among Complexity Measures; **K.6.0 [Management of Computing and Information Systems]**: General - *Economics*; **K.6.3 [Management of Computing and Information Systems]**: Software Management

General Terms: Management, Measurement, Performance.

Additional Key Words and Phrases: Software Productivity, Software Maintenance, Software Complexity, McCabe Metrics, Cyclomatic Complexity.

Research support for the second author from the Center for Information Systems Research and the International Financial Services Research Center is gratefully acknowledged. Helpful comments were received from Bill Curtis on an earlier draft.

1. INTRODUCTION

A critical distinction between software engineering and other, more well-established branches of engineering is the shortage of well-accepted measures, or metrics of software. Without such metrics, the tasks of planning and controlling software development and maintenance will remain stagnant in a "craft"-type mode, wherein greater skill is acquired only through greater experience, and such experience cannot be easily communicated to the next system for study, adoption, and further improvement. With such metrics, software projects can be quantitatively described, and the methods and tools used on the projects to improve productivity and quality can be evaluated. These evaluations will help the discipline grow and mature, as progress is made at adopting those innovations that work well, and discarding or revising those that do not.

Of particular concern is the need to improve the software maintenance process, given that maintenance is estimated to consume 40-75% of the software effort [Vessey and Weber, 1986]. What differentiates maintenance from other aspects of software engineering is, of course, the constraint of the existing system. The existing system constrains the maintenance work in two ways, the first through the imposition of a general design structure that circumscribes the possible designs of the system modifications, and second, and more specifically, through the complexity of the existing code which must be modified.

A measure, or metric of this latter, more specific type of system influence that has been proposed is the McCabe's cyclomatic complexity metric, a measure of the maximum number of linearly independent circuits in a program control graph [McCabe, 1976]. As described by McCabe, a primary purpose of the metric is to "...identify software modules that will be difficult to test or maintain" [McCabe, 1976, p.308], and therefore is of particular interest to researchers and practitioners concerned with maintenance. The McCabe metric has been widely used in research [Zuse and Bollmann, 1989], and a recent article by Shepperd cites some 63 articles that are directly

or indirectly related to the McCabe metric [Shepperd, 1988]. However, despite the widespread interest and popularity of this metric, it has not been without criticism, with an early critique appearing in 1982 [Evangelist], and most recently the comprehensive aforementioned critique by Shepperd [1988].

Shepperd's review highlights two types of criticisms. The first, which he labels "theoretical criticisms," have to do with analytical criticisms of the algorithm by which the cyclomatic complexity metric is computed for software. Most prominent among these are the refinements proposed by Myers [Myers, 1977] and Hansen [Hansen, 1978]. Shepperd also notes that, among the numerous empirical validations or uses of the metric, that the most consistent single result is the high degree of correlation between McCabe's metric and a count of source lines of code (SLOC). As evidence of this, four studies cited by Shepperd had Pearson correlation coefficients of .9 or greater for these two metrics.[1] This "empirical criticism" suggests that the additional effort required for computing and understanding the McCabe cyclomatic complexity metric may not be worthwhile in practice.

Therefore, given the need for a complexity metric such as McCabe's in general, but also the well-formed criticism of the McCabe metric in particular, the need for additional research in this area is clear. The purpose of the current research is to empirically test the two main criticisms summarized or raised by Shepperd, and to suggest needed improvements. In regard to the theoretical criticisms of Myers and Hansen, Shepperd does note that it is "arguable whether [they] represent much of an improvement over that of McCabe" (p. 32), since "The majority of modifications to McCabe's original metric remain untested." (p.35) [Shepperd, 1988] The current research directly addresses this call for an empirical test of these variations.

The second research question revolves around the practical usefulness of the metric, given the high correlations noted by some previous research. As noted by Shepperd, concerns about the

---

[1] See [Basili and Perricone, 1984], [Feuer and Fowlkes, 1980], [Paige, 1980], and [Woodward, Hennell, and Hedley, 1979].

external validity of the data and analyses in some of the previous studies can be used raised to mitigate some of the results, particularly those using data on small programs, often using student subjects. Therefore, these results bear validation on data from actual systems, and, in particular, from data on maintenance projects, since use of the metric for testing and maintenance was one of its author's main stated purposes. In particular, this research seeks not to determine whether Cyclomatic complexity captures all aspects of complexity in one figure of merit, but rather to answer the question raised by Shepperd, as to whether cyclomatic complexity can serve as a "useful engineering approximation" [Shepperd, 1988].

Briefly summarized, the results of this research were as follows. In a test on 834 software modules from a number of existing real-time systems, neither the Myers nor Hansen variants were found to be significantly different from the original value as computed by McCabe. However, in regard to the second set of research questions, the empirical criticism of Shepperd and others was validated, in that the correlation between Cyclomatic complexity and SLOC was found to be .95 for these real world systems. However, a simple transformation involving the cyclomatic complexity metric is proposed, whereby the cyclomatic complexity is divided by the size of system in source statements, thereby determining a "complexity density" ratio. This complexity density ratio is demonstrated to be a useful predictor of software maintenance productivity. on a small sample of actual maintenance projects.

The rest of this paper is organized as follows: Section 2 details the main research questions, and provides references to previous research literature where appropriate. Section 3 outlines the data collection procedures and summarizes the data set used in the research. Results are presented in Section 4, and a concluding remarks are presented in Section 5.

2. CYCLOMATIC COMPLEXITY METRICS

2.1 Analytic Critiques

McCabe [1976] proposed that a measure of the complexity is the number of possible paths through which the software could execute. Since the number of paths in a program with a backward branch is infinite, he proposed that a reasonable measure would be the number of independent paths. After defining the program graph for a given program, the complexity calculation would be:

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

where $V(G)$ = the cyclomatic complexity.
e = the number of edges.
n = the number of nodes.

Analytically, V(G) is the number of predicates plus 1 for each connected single-entry single-exit graph or subgraph.

From the beginning, a number of researchers have proposed variations on the original metric. The two most prominent variations are those of Hansen (1978) and Myers (1977) [Shepperd, 1988]. Hansen [1978] gave four simple rules for calculating McCabe's Complexity:

1. Increment one for every IF, CASE or other alternate execution construct.
2. Increment one for every Iterative DO, DO-WHILE or other repetitive construct.
3. Add two less than the number of logical alternatives in a CASE.
4. Add one for each logical operator (AND, OR) in an IF.

Myers [1977] demonstrated what he believed to be a flaw in McCabe's metric. Myers pointed out that an IF statement with logical operators was in some sense less complicated than two IF

statements with one imbedded in the other because there is the possibility of an extra ELSE clause in the second case. For example:

```
IF ( A and B ) THEN
    ...
 ELSE
      ...
```

is less complex than:

```
IF ( A ) THEN
   IF ( B ) THEN
       ...
      ELSE
       ...
 ELSE
   ...
```

Myers [1977] noted that calculating V(G) by individual conditions (as suggested by McCabe) made V(G) identical for these two cases. To remedy this problem, Myers suggested using a tuple of (CYCMID,CYCMAX) where CYCMAX is McCabe's measure, V(G), (all four rules are used) and CYCMID uses only the first three. Using this complexity interval, Myers was able to resolve the anomalies in the complexity measure.

Hansen [1978] raised a conceptual objection to Myers's improvement. Hansen felt that because CYCMID and CYCMAX were so closely related, using a tuple to represent them was redundant and did not provide enough information to warrant the intrinsic difficulties of using a tuple. Instead he proposed that what was really being measured by the second half of the tuple (CYCMAX) was really the complexity of the expression, not the number of paths through the code. Furthermore, he felt that rule #3 above also was just a measure of the complexity of the expressions. To provide a better metric, he suggested that a tuple consisting of (CYCMIN, operation count) be used where CYCMIN is calculated using just the first two rules for counting McCabe's complexity.

All of the above tuple measures have a problem in that, while providing ordering of the

complexity of programs, they do not provide a single number that can be used for quantitative comparisons. It is also an open research question whether choosing one or the other of these metrics makes any practical difference. While a number of authors have attempted validation of the CYCMAX metric (e.g. Li and Cheung, 1987; Lind and Vairavan, 1989) there does not appear to be any empirical validation of the practical significance of the Myers or Hansen variations [Shepperd 1988]. Therefore, one question investigated in this research is a validation of the CYCMIN, CYCMID, and CYCMAX complexity metrics.

2.2 Empirical Tests

In Shepperd's comprehensive review of previous research on the metric, he notes that, among the numerous empirical validations or uses of the metric, that the most consistent single result is the high degree of correlation between McCabe's metric and a count of source lines of code (SLOC). As evidence of this, four studies cited by Shepperd had Pearson correlation coefficients of .9 or greater for these two metrics. This "empirical criticism" suggest that the additional effort required for computing and understanding the McCabe cyclomatic complexity metric may not be practically worthwhile. However, as noted by Shepperd, concerns about external validity of the data and analyses in some of the previous studies can be raised to mitigate some of the results, particularly those using data on small programs from student subjects. Therefore, these results bear validation.on data from actual systems. In particular, applied research in this area does not seek to determine whether cyclomatic complexity captures all aspects of complexity in one figure of merit, but rather looks to answer the overriding question raised by Shepperd, as to whether cyclomatic complexity can serve as a "useful engineering approximation" [Shepperd, 1988].

2.3 Productivity Applications

As suggested by McCabe, a likely use of a complexity metric is to "...identify software modules that will be difficult to test or maintain" [1976, p. 308]. The emphasis on maintenance is appropriate, given that software maintenance is an area of increasing practical significance. It is estimated that from 40-75% of information systems resources are spent on software maintenance [Vessey and Weber, 1986]. However, according to Lientz and Swanson [1980] relatively little research has been done in this area. One area that is of considerable practical importance for research is the effect of existing system complexity on maintenance productivity. It is often assumed that a) more complex systems are harder to maintain, and b) that systems suffer from "entropy", and therefore become more chaotic and complex as time goes on [Belady and Lehman, 1976]. These assumptions raise a number of research questions, in particular, when does software become sufficiently costly to maintain that it is more economic to replace (rewrite) it than to repair (maintain) it? This is a question of some practical importance, as one type of tool in the Computer Aided Software Engineering (CASE) marketplace is the so-called "restructurer", designed to reduce existing system complexity by automatically modifying the existing source code [Parikh, 1986; US GSA, 1987]. Knowledge that a metric such as CYCMAX accurately reflects the difficulty in maintaining a set of source code would allow management to make rational choices in the repair/replace decision, as well as aid in evaluating these CASE tools. The empirical evidence linking software complexity to software maintenance costs has been criticized as being relatively weak [Kearney, 1986]. However, studies have shown that a significant fraction of the staff resources used in maintenance are spent in understanding the existing code [Fjeldstad and Hamlen, 1979]. Therefore, a metric that measures complexity should prove to be a useful predictor of maintenance costs.

2.4 Summary of Research Questions

Based upon the above review of the previous literature, most especially that of Shepperd, three research questions are the subject of this empirical study. They are:

1. What is the empirical relationship between the original McCabe formulation of the cyclomatic complexity metric and that of the later variants proposed by Myers and Hansen? The practical implications of these questions are that, if the variants are shown to be significantly different than the original formulation in practice, then they merit collection and analysis by both researchers and practitioners.

2. What is the empirical relationship between the cyclomatic complexity metric and simpler size metrics such as source lines of code? The practical implication of this question is whether cyclomatic complexity should be used as the numerator in ratios to determine the productivity of software development and maintenance efforts, or whether simpler to compute measures, such as SLOC would suffice.

3. How can cyclomatic complexity metrics be used to assess the impact of existing code complexity on maintenance project productivity? If a relationship can be shown between some form of the metric and productivity, then the metric can be used as a component in management planning and control of software maintenance, an activity of considerable economic significance.

3. DATA COLLECTION

3.1 Data Overview and Background of the Datasite

The approach taken to addressing the research questions in this study was to collect detailed data about a number of completed software projects at one firm, hereafter referred to as Alpha Company. All the projects used in this study were small, customized, and primarily real-time defense applications undertaken in the period from 1984-1989, with most of the projects in the last three

years of that period. The company considers the data contained in this study proprietary and therefore requested that its name not be used and any identifying data that could directly connect it with this study be withheld. The numeric data, however, have not been altered in any way. The company employed an average of about 30 professionals in the software department.

One benefit of this approach was that because all the projects were undertaken by one organization, neither inter-organizational differences nor industry differences should confound the results. A possible limitation is that this concentration on one industry and one company may somewhat limit the applicability of the results. At a minimum, however, the defense industry is a very important industry for software[2] and the results will clearly be of interest to researchers in that field. More generally, these results may provide suggestions for research at other sites.

3.2 Data Sources

The data for this study were derived from three sources [Gill, 1989]:

1. **Source Code** -- A copy of the source code developed or modified by each project was obtained.

2. **Accounting Database** -- This database contained the number of hours every software engineer worked on each project for every week of the study period.

3. **Activity Reports** -- A description of the work done by each engineer for each week for every project.

Of the source code developed for the projects, 74.4% was written in Pascal, and the other 25.6% in a variety of other third generation languages, primarily FORTRAN. Following Boehm [1981], only *deliverable* code was included in this study, with deliverable defined as any code which was placed under a configuration management system. This definition eliminated any purely

[2] Fortune magazine reported in its September 25, 1989 issue that the Pentagon spends $30 billion annually on software.

temporary code, but retained code which might not have been part of the delivered system, but was considered valuable enough to be saved and controlled. (Such code included programs for generating test data, procedures to recompile the entire system, utility routines, etc.) The source code was the main data source for addressing the first two sets of research questions.

In order to address the maintenance productivity question, two additional types of data were required. The first of these was the total number of hours worked on the project and the person charging the project. These were obtained from the firm's accounting database. Because the hourly charges were used to bill the project customer and were subject to United States Government Department of Defense (DOD) audit, the engineers and managers used extreme care to insure proper allocation of charges. For this reason, these data are believed to be accurate representations of the actual effort expended on each project. A third data source was a weekly summary called an activity report of the work done by each software engineer every week as part of their standard record-keeping for DOD reporting. An example of an activity report appears in Appendix A. These activity reports were used to isolate coding and debugging activities when examining the effect of complexity density on maintenance productivity.

3.3 Data Definitions

For the purposes of this research, a software module was defined as the source code that was placed in a single physical file (which was often compiled separately). The size of the software modules was measured in non-comment source lines of code (NCSLOC). The definition of a line of code adopted for this study is that proposed by Conte, Dunsmore and Shen [1986, p. 35]:

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

This set of rules has several major benefits. It is very simple to calculate and it is very easy

to translate for any computer language. Because of its simplicity, it has become the most popular SLOC metric [Conte, et al. 1986, p. 35]. Through consistent use of this metric, results become comparable across different studies.[3]

In addressing the first two research questions, this study included a total of 834 modules of which 771 were written in Pascal and 63 in FORTRAN. They averaged 176.2 NCSLOC (Non-Comment Lines of Code) with a standard deviation of 257.9. In total, approximately 150,000 lines of code (NCSLOC) from 21 software systems were analyzed.

The definition of software maintenance adopted by this research is that of Parikh and Zvegintzov [1983], namely "work done on a software system after it becomes operational." Lientz and Swanson [1980, pp. 4-5] include three tasks in their definition of maintenance: "...(1) corrective -- dealing with failures in processing, performance, or implementation, (2) adaptive -- responding to anticipated change in the data or processing environments; (3) perfective -- enhancing processing efficiency, performance, or system maintainability." The projects described below as maintenance work pertain to all of the three Lientz and Swanson types.

To address the third research question, project data were gathered for seven maintenance projects in order to test the hypothesis concerning the effect of complexity on maintenance productivity. The average size of a maintenance project was 1006 added (standard deviation, 1158), and required an average of 1059 work hours (standard deviation, 982).

---

[3] As Boehm [1987] has described, SLOC have method has many deficiencies as a metric. In particular, they are difficult to define consistently, as Jones [1986] has identified no fewer than eleven different algorithms for counting code. Also, SLOC do not necessarily measure the "value" of the code (in terms of functionality and quality). However, Boehm [1987] also points out that no other metric has a clear advantage over SLOC. Furthermore, SLOC are easy to measure, conceptually familiar to software developers, and are used in most productivity databases and cost estimation models. For these reasons, it was the metric adopted by this research.

4 RESULTS

4.1 Empirical Tests of Analytic Critique

For the 834 modules, three complexity metrics were computed:

CYCMAX — the original McCabe cyclomatic complexity.

CYCMID — the Myers variation.

CYCMIN — the Hansen variation.

Table 4.1.1 gives the average value for each metric. Table 4.1.2 gives the Pearson correlation coefficients for the three complexity metrics used in this study.

CYCMIN	CYCMID	CYCMAX
20.3	22.2	24.0

Table 4.1.1 Means of Complexity Metrics

	CYCMIN	CYCMID	CYCMAX
CYCMIN	1.000 (0.0000)	0.9849 (0.0001)	0.9861 (0.0001)
CYCMID		1.000 (0.0000)	0.9980 (0.0001)
CYCMAX			1.000 (0.0000)

Note: The number below the correlation is the statistical significance. (n=834).

Table 4.1.2 Correlations Metrics for Code Complexity

The most striking result is the extent to which similar metrics are highly correlated with each other. These results indicate that the metrics proposed by Myers (CYCMID) and Hansen (CYCMIN) measure complexity in a very similar manner to McCabe's metric, CYCMAX. In fact,

the empirical data suggest that there are unlikely to be any practically significant different results using CYCMIN or CYCMID instead of CYCMAX.

4.2 Test of Empirical Criticism

As suggested by previous research, the length metric, NCSLOC, and the complexity measure, CYCMAX also turned out to be highly correlated. Table 4.2.1 gives the Pearson correlation coefficients for the complexity metrics with the SLOC metrics. Similar to the results found in some previous research, the cyclomatic complexity metric is highly correlated with length in NCSLOC.

	NCSLOC
CYCMIN	0.934 (0.000)
CYCMID	0.949 (0.000)
CYCMAX	0.949 (0.000)

Table 4.2.1: Correlations of Complexity and SLOC

4.3 Pilot Test of the Impact of Complexity on Maintenance Productivity

Given the above results, it would be inappropriate to test the impact of complexity on productivity by using McCabe's or derivative complexity metrics directly since they are so closely related to program length. If McCabe's complexity were used unnormalized, the results would be dominated by the length effect, i.e., the tested hypothesis would actually be that maintenance productivity is related to program length. While this is a possible hypothesis, there are two reasons why it is less interesting than the impact of cyclomatic complexity. The first is that managers typically

do not have a great deal of control over the size of a program as it is intimately connected to the size of the application. However, by measuring and adopting complexity standards, and/or by using CASE restructuring tools, they can manage unnecessary cyclomatic complexity. The second reason is that there are valid intuitive reasons why the length of the code may not have a large effect on maintenance productivity. When a software engineer maintains a program, each change will tend to be localized to a fairly small number of modules. He will, therefore, only need detailed knowledge of a small fraction of the code. The size of this fraction is probably unrelated to the total size of the program. Thus the length of the entire code is less relevant.[4] Therefore, a transformed metric, complexity density, is defined as the ratio of cyclomatic complexity to thousand lines of executable code (KNCSLOC). These *complexity density* measures are similar to Gilb's logical decisions per statement [Gilb, 1977], but they have not often been published. One exception is a by-product of Selby's research on reused code [Selby, 1988]. In his study of 25 NASA projects, his mean was 81.4, with a standard deviation of 57.2. CMAXDENS, the average McCabe complexity per thousand lines of code was 121.3 for our sample (standard deviation of 62.7).

Since both Pascal and FORTRAN code was analyzed in this research, it is important to determine whether there exists any effect of programming language used. The average complexity density of FORTRAN modules was compared with Pascal modules (see Table 4.3.1).

[4] In fact, the correlation of productivity with *initial* code length was not significant (Pearson coefficient, -0.21; p=0.61).

Variable	FORTRAN (n=63)	Pascal (n=771)
CMINDENS	0.10	0.10
CMIDDENS	0.11	0.11
CMAXDENS	0.12	0.11

Table 4.3.1: Comparison of FORTRAN and Pascal Average Complexity Density

No statistical difference between the means of the modules from the different languages was found. These results show that there is little difference that can be attributed to the different languages, at least for the data in this sample. It is also an indication of the general robustness of the complexity density metric.

The value of such a complexity metric lies in whether it can predict how difficult a piece of software will be to maintain. Such a prediction can be used either in estimating resources required, or in trying to restructure the code so that the maintainability is improved. The current best measure of maintainability is the productivity of the maintenance project on a piece of software. In order to test the complexity density metric, its average value was computer for seven application systems before the were maintained. Logically, this initial (pre-maintenance) complexity density should only affect those phases that are directly related to maintaining the code. User support, system operation, and management functions should not be directly affected by the complexity of the code. Therefore, maintenance productivity was defined as the total number of lines added divided by the time spent in coding and testing upgrades to the software. To perform this analysis the pre-maintenance cyclomatic complexity density for the maintenance projects was computed and plotted against the

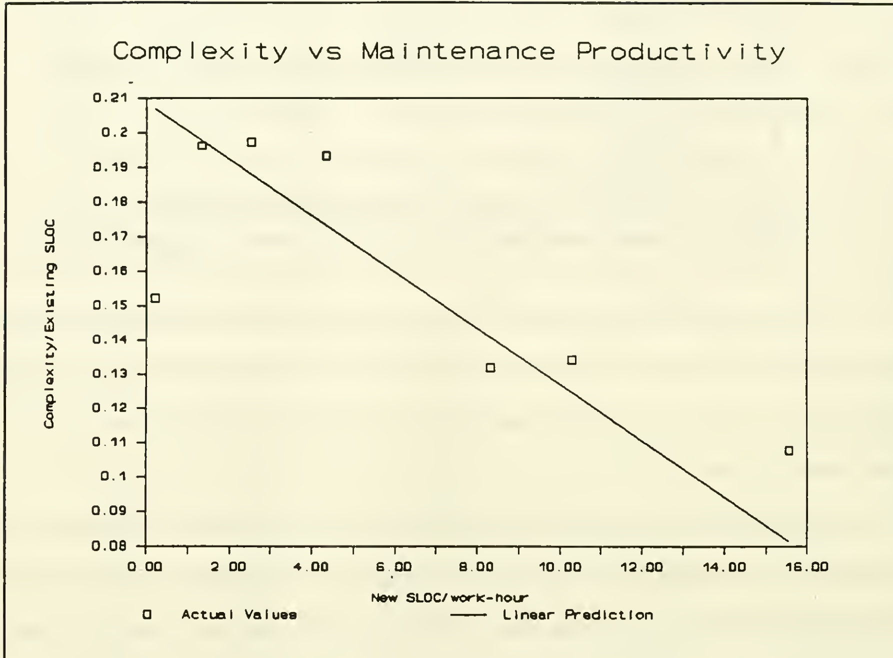


Figure 4.3: Maintenance Productivity versus Complexity Density

maintenance productivity of each of the seven projects, and is shown in Figure 4.3[5]

---

[5] Given the apparent linearity of this relationship, an exploratory regression was estimated, with full knowledge that the small number of data points precluded standard analysis. The results were:

$$\text{PRODUCTIVITY} = \underset{(3.53)}{28.7} - \underset{(-2.69)}{0.134} * \text{CMAXDENS}$$

$R^2 = 0.59$ Adjusted $R^2 = 0.51$

F-Value = 7.22 $n = 7$

From examining the figure, it is evident that the first point is an outlier. On examining that point more closely, it was found that there were two reasons that it might not be representative. It was the smallest project examined with only 221 lines of code (NCSLOC) added. Furthermore, 21% of the hours in the project could not be directly assigned because the activity reports had not be filed. The remaining hours were therefore assigned proportionately based upon the other activity reports. This was the largest percentage for any of the projects studied (no other project had more than 15%). (Continued next page.)

This figure suggests that productivity declines with increasing complexity density. While it would be speculative to ascribe too much importance to the results, which are based on only seven data points, the results do seem sufficiently strong and the implications sufficiently important that further study is indicated. If these results continue to hold on a larger independent sample, then such results would provide strong support for the use of the complexity density measure as a quantitative method for analyzing software to determine its maintainability. It might also be used as an objective measure of this aspect of software quality.

5 SUMMARY AND CONCLUDING REMARKS

This paper began by directly addressed the two research questions raised by Shepperd regarding the McCabe cyclomatic complexity metrics. The first question was whether the analytical criticisms of the original metric had any practical significance. Based upon the data used in this study, neither the Myers nor Hansen variations seem to result in values that are equivalent to the original specification for all practical purposes to which they might be put. Thus, these data support Shepperd's hypothesis that the variations do not represent a significant difference over the original formulation. The second research question revolves around the empirically-derived value added by the metric, given the high correlations found by previous research between the metric and standard size measures, most particularly LOC. The data from this study provide additional evidence of this high correlation. This is particularly noteworthy since these data are from actual systems, and, in particular, include data on maintenance projects, which is a main intended domain of the metric.

When the outlying project is removed, the results improve dramatically:

$$\text{PRODUCTIVITY} = \underset{(6.67)}{31.3} - \underset{(-4.98)}{0.142} * \text{CMAXDENS}$$

$R^2 = 0.86$ Adjusted $R^2 = 0.82$

F-Value = 24.84 n = 6

Going beyond this correlation, this paper has proposed the use of a transformed version of the metric, referred to as "complexity density", whereby the ratio of the cyclomatic complexity of the module to its length in LOC is calculated. This ratio is meant to represent the weighted complexity of a module, and hence its likely level of maintenance task difficulty. This proposed complexity density ratio is tested on a small pilot sample of projects and shown to be a good single value predictor of maintenance costs. Thus, it is proposed that this transformed version of the cyclomatic complexity metric can serve, in Shepperd's words, as a "useful engineering approximation" [Shepperd, 1988]. Of course, further empirical research will be required to test these pilot results on a larger sample drawn from a different environment in order to validate the usefulness of the complexity density ratio. The complexity density ratio proposed by this research could prove to be a simple, but practically useful measure of complexity. It incorporates the McCabe cyclomatic complexity metric, a well-known and well-understood measure of complexity. In addition, in the intervening years since its introduction, the collection of the data necessary to compute the metric has been automated by a number of tools.[6] Therefore, the early difficulties in collecting these data have been resolved, and therefore data collection should not prove to be a practical barrier to the ratio's use. The continued search for useful metrics of software product complexity is a necessary first step in the ongoing process of moving software development firmly into the realm of engineering. With well-founded complexity metrics, developers will have objective and useful yardsticks with which to evaluate their initial products. These metrics will also be used by each maintenance team as it seeks to enhance the initial product without increasing the level of unnecessary complexity. Both of these aspects should make a contribution towards the improved management of the software development and maintenance process.

---

[6] e.g., see Language Technology Incorporated's INSPECTOR product, and SET Laboratories' product.

6 BIBLIOGRAPHY

[Basili and Perricone, 1984]
Basili, V. R. and B. Perricone, "Software Errors and Complexity: An Empirical Investigation," *Communications of the ACM*, 27 (1): 42-52, (January 1984).

[Belady and Lehman, 1976]
Belady, L. A. and Lehman, M. M.; "A Model of Large Program Development"; *IBM Systems Journal*, v. 15, n. 1, pp. 225-252, (1976).

[Boehm, 1981]
Boehm, Barry W.; *Software Engineering Economics*; Prentice-Hall, Englewood Cliffs, NJ, 1981.

[Boehm, et al. 1984]
Boehm, B. W.; Gray, T. E.; and Seewaldt, T.; *IEEE Transactions on Software Engineering*; May 1984.

[Boehm, 1987]
Boehm, Barry W.; "Improving Software Productivity"; *Computer*; September 1987; pp. 43-57.

[Conte, et al. 1986]
Conte, S. D.; Dunsmore, H. E.; and Shen, V. Y.; *Software Engineering Metrics and Models*; Benjamin/Cummings Publishing Company, Inc., 1986.

[Evangelist, 1982]
Evangelist, W. M., "Software complexity metric sensitivity to program structuring rules," *Journal of Systems & Software*, 3 231-243, (1982).

[Feuer and Fowlkes, 1980]
Feuer, A. R. and E. B. Fowlkes, "Some results from an empirical study of computer software", Proceedings of the Fourth IEEE International Conference on Software Engineering, Munich, Germany, 1980, pp. 351-355.

[Fjeldstad and Hamlen, 1979]
Fjeldstad, R. K. and Hamlen, W. T.; "Application program maintenance study: Report to our Respondents"; *Proc of GUIDE 48*; The Guide Corporation, Philadelphia, 1979. Also in [Parikh and Zvegintzov, 1983].

[Gilb, 1977]
Gilb, Thomas; *Software Metrics*; Winthrop Press, Cambridge, MA 1977.

[Gill, 1989]
Gill, Geoffrey K.; "A Study of the Factors that Affect Software Development Productivity"; unpublished MIT Sloan Masters Thesis; 1989; Cambridge, MA.

[Hansen, 1978]
Hansen, W. J.; "Measurement of Program Complexity By the Pair (Cyclomatic Number, Operator Count)"; *ACM SIGPLAN Notices*; 13 (3):29-33, March 1978.

[Jones, 1986]
Jones, Capers; *Programming Productivity*; McGraw-Hill, New York, 1986.

[Kearney, et al, 1986]
Kearney, J. K., R. L. Sedlmeyer, W. B. Thompson, M. A. Gray and M. A. Adler, "Software Complexity Measurement", Communications of the ACM, 29 (11): 1044-1050, (November 1986).

[Li and Cheung, 1987]
Li, H. F. and Cheung, W. K.; "An Empirical Study of Software Metrics"; *IEEE Transactions on Software Engineering* SE-13 (6):697-708, June, 1987.

[Lientz and Swanson, 1980]
Lientz, B. P. and Swanson, E. B.; *Software Maintenance Management*; Addison-Wesley Publishing Company, 1980.

[Lind and Vairavan, 1989]
Lind, Randy K. and Vairavan, K.; "An Experimental Investigation of Software Metrics and Their Relationship to Software Development Effort"; *IEEE Transactions on Software Engineering*; 15 (5):649-653, 1989.

[McCabe, 1976]
McCabe, Thomas J.; "A Complexity Measure"; *IEEE Transactions on Software Engineering*; SE-2 (4):308-320, 1976.

[Myers, 1977]
Myers, G. J.; "An extension to the Cyclomatic Measure of Program Complexity". *SIGPLAN Notices*; 12 (10):61-64, 1977.

[Paige, 1980]
Paige, M., "A metric for software test planning", Proceedings of COMPSAC 80, Buffalo, New York, 1980, pp. 499-504.

[Parikh, 1986]
Parikh, Girish; "Restructuring your COBOL Programs"; *Computerworld Focus*; 20 (7a):39-42; February 19, 1986.

[Parikh and Zvegintzov, 1983]
Parikh, G. and Zvegintzov, N.; eds. *Tutorial on Software Maintenance*, IEEE Computer Science Press, Silver Spring, MD (1983).

[Selby, 1988]
Selby, Richard W.; "Empirically Analyzing Software Reuse in a Production Environment"; *Software Reuse -- Emerging Technologies*; Traz, W. eds. IEEE Computer Society Press, NY NY 1988.

[Shepperd, 1988]
Shepperd, M., "A critique of cyclomatic complexity as a software metric," Software Engineering Journal, 3 (2): 30-36, (March 1988).

[US GSA, 1987]
Parallel Test and Evaluation of a Cobol Restructuring Tool; Federal Software Management Support Center; United States General Services Administration, Office of Software Development and Information Technology, Falls Church, Virginia Sept 1987.

[Vessey and Weber, 1983]
Vessey, I. and Weber, R.; "Some Factors Affecting Program Repair Maintenance: An Empirical Study"; *Communications of the ACM*; 26 (2):128-134, February, 1983.

[Woodward, et al, 1979]
Woodward, M. R., M. A. Hennell and D. A. Hedley, "A measure of control flow complexity in program text," *IEEE Transactions on Software Engineering*, 5 (1): 45-50, (1979).

[Zuse and Bollmann, 1989]
Zuse, H. and P. Bollmann, "Software metrics: using Measurement Theory to Describe the Properties and Scales of Static Software Complexity Metrics," *ACM SIGPLAN Notices*, 24 (8): 23-33, (1989).

Appendix A: Example of an Activity Report

Junior Software Engineer's Name Weekly Report 4-1-1987

==

Project #1	35 hrs.	Spent the whole time working with [Staff Scientist] and [Principal Scientist] trying to find the cause of the difference between [Baseline Runs] and [New Instrument Runs]. After a lot of head scratching it turns out that both were correct. We just had to play with the format of the data and units to get the two data sets to match to within [xxx] RMS. This is still above the [yyy] RMS limit, but the [baseline runs] and [new instrument runs] started off being [zzz] RMS apart. So, until the [new instrument] can be improved, my software can do no better than it is doing now.
Project #2	1 hrs.	I had very little time to work on this, but I did manage to start to finish the modify adhoc target routine.
Project #3	9 hrs	The automatic height input for the calibration routine is up and running. I also made some more modifications so that they can return to the old method if the IEEE board fails.
Project #4	2 hrs	[Staff Scientist] and I spent the two hours trying to get the microVAX up after we installed the [Special Boards]. We ended up having to pull the boards back out to reboot the machine. The boards were eventually installed, but there still seems to be something wrong.

Total = 45.0 hours

3313 075

Date Due

NOV 24 1991

Lib-26-67

MIT LIBRARIES DUPL

3 9080 00701567 7