
MASSACHUSETTS
INSTITUTE OF TECHNOLOGY
LIBRARIES

WORKING PAPER
ALFRED P. SLOAN SCHOOL OF MANAGEMENT

Productivity Impacts of Software Complexity and Developer Experience

Geoffrey K. Gill
Chris F. Kemerer

MIT Sloan School WP #3107-90

January 1990

MASSACHUSETTS
INSTITUTE OF TECHNOLOGY
50 MEMORIAL DRIVE
CAMBRIDGE, MASSACHUSETTS 02139

Productivity Impacts of Software Complexity and Developer Experience

Geoffrey K. Gill
Chris F. Kemerer

MIT Sloan School WP #3107-90

January 1990

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 on the first draft from W. Harrison, D. R. Jeffery, W. Orlikowski, and D. Zweig are gratefully acknowledged.

Productivity Impacts of Software Complexity and Developer Experience

Geoffrey K. Gill

Chris F. Kemerer

MIT Sloan School of Management

Cambridge, MA

January 1990

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 on the first draft from W. Harrison, D. R. Jeffery, W. Orlikowski and D. Zweig are gratefully acknowledged.

Productivity Impacts of Software Complexity and Developer Experience

Geoffrey K. Gill

Chris F. Kemerer

MIT Sloan School of Management

Cambridge, MA

Abstract

The high costs of developing and maintaining software have become widely recognized as major obstacles to the continued successful use of information technology. Unfortunately, against this backdrop of high and rising costs stands only a limited amount of research in measuring software development productivity and in understanding the effect of software complexity on variations in productivity. The current research is an in-depth study of a number of software development projects undertaken by an aerospace defense contracting firm in the late 1980s. These data allow for the analysis of the impact of software complexity and of varying levels of staff expertise on both new development and maintenance.

In terms of the main research questions, the data support the notion that more complex systems, as measured by size-adjusted McCabe cyclomatic complexity, require more effort to maintain. For this data set, the Myers and Hansen variations were found to be essentially equivalent to the original McCabe specification. Software engineers with more experience were more productive, but this labor production input factor revealed declining marginal returns to experience. The average net marginal impact of additional experience (minus its higher cost) was positive.

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.6.1 [Management of Computing and Information Systems]**: Project and People Management; **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.

1 INTRODUCTION

The high costs of developing and maintaining software have become widely recognized as major obstacles to the continued successful use of information technology. For example, Barry Boehm has estimated that $140 billion is spent annually worldwide on software [Boehm, 1987]. Therefore, even a small increase in productivity would have significant cost reduction effects. However, despite the practical importance of software productivity, there is only a limited amount of research in measuring software development productivity and in understanding the effect of software complexity on variations in productivity.

The purpose of this paper is to add to the body of research knowledge in the areas of software development productivity and software complexity. Traditionally, software development productivity has been measured as a simple output/input ratio, most typically source lines of code (SLOC) per work-month [Boehm, 1981, Conte, et al., 1986, Jones, 1986]. While this metric has a long research history, and is widely used by practitioners, criticisms of the metric have sparked new research questions. The numerator, SLOC, is one measure of output. However, in the case of software maintenance, it does not take into consideration possible differences in the underlying complexity of the code being maintained. Since researchers such as Fjeldstad and Hamlen have noted that approximately 50% of the effort in maintenance may be in understanding the existing code, neglecting to account for initial code complexity may be a serious omission [Fjeldstad/Hamlen, 1979].

In terms of the denominator, work-months (often called man-months), this too may be only a rough measure of productivity. Numerous lab experiments and field observations have noted the wide variations in productivity across individual systems developers [Chrysler, 1978, Sackman, et al., 1968, Curtis, 1981, Dickey, 1981]. The idea is that who charges those work-months could be as important as how many work-months are charged. Yet, the usage of the aggregated work-months metric ignores this potential source of productivity variations. Another concern with this metric is that it provides no information on how those hours were spent, in terms of which activities of the systems development life cycle were supported. In particular, some previous research [McKeen, 1973] has suggested that greater time spent in the design phase may result in better systems.

The current research is an in-depth study of a number of software development projects undertaken by an aerospace defense contracting firm in the late 1980s. The data collection effort went beyond collecting only aggregate SLOC/work-month data. In terms of the source code, data were collected on the complexity of the code written, as measured by the McCabe Cyclomatic Complexity measure [1976], and

the variations proposed by Myers [1977], and Hansen [1978]. For maintenance projects, these data were also collected on the existing system code before the maintenance effort was begun. Code size and complexity were measured on approximately 100K non-comment Pascal and Fortran SLOC in 465 modules of source code. These data allow for the analysis of the impact of complexity on both new development and maintenance.

In terms of the work-month information, data were collected by person by week in terms of both the number of hours and the activities upon which those hours were spent. This information was obtained by collecting and processing approximately 1200 contemporaneously-written activity reports. Therefore, analysis of the impact of varying levels of staff expertise and varying styles of project management is possible.

The statistical results are reported in full in Section 4. In terms of the main research questions, the data support the notion that more complex systems, as measured by cyclomatic complexity, require more effort to maintain. The magnitude of this effect is that the productivity of maintaining the program drops an estimated one SLOC/day for every extra path per 1000 lines of existing code. For this data set, the Myers and Hansen variations were found to be essentially equivalent to the original McCabe specification. In terms of productivity, the data support the notion that more time spent in the design phases of the project is related to increased productivity, at a rate of over six SLOC/day for each additional percentage point of design activity. Software engineers with more experience were more productive, but there appeared to be declining marginal returns to experience. The average net marginal impact of additional experience (minus its higher cost) was positive.

This paper is organized as follows. Section 2 describes the main research questions and compares and contrasts the current work with previous research. Section 3 describes the data site and the data collection methodology, and presents the data. Section 4 presents the analysis of the source code, and Section 5 presents the results of the productivity and complexity analysis. Finally, Section 6 offers some conclusions and suggestions for future research.

2 RESEARCH QUESTIONS AND PRIOR RESEARCH

A vast literature exists on software engineering metrics and modeling, and a comprehensive review is clearly beyond the scope of this paper.[1] Rather, in this section the current research questions will be

---

[1] The interested reader is referred to [Conte, et al., 1986] and [Jones, 1986] for broad coverage of a number of related topics. Also, an earlier, lengthier version of this section appears in [Gill, 1989].

developed within the framework of specific prior research on the following topics: a) software output metrics, b) new software development productivity, and c) software maintenance productivity.

2.1 Software Metrics

There are many possible ways to measure the productivity of software development. The most widely used metric for size involves counting source lines of code (SLOC) in some manner.[2] 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.

One problem with almost all the counting rules for SLOC is that they ignore comments. While comments are not executable, they do provide a useful function and are therefore properly regarded as part of the output of the system development process. For this reason, comment lines were separately counted in the current research. This issue is discussed more thoroughly in the methodology section.

The most widely accepted metric for complexity is that proposed by McCabe [1976].[3] He 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:

---

[2] As Boehm [1987] has described, this method has many deficiencies. In particular, SLOC 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.

[3] Another option would have been to use one of the Halstead Software Science metrics in place of the McCabe metric [Halstead, 1977]. However, previous research has shown no advantage for these metrics over the McCabe metric [Curtis, et al., 1979; Lind and Vairavan, 1989], and more recent work has been critical of Software Science metrics [Shen, Conte, and Dunsmore 1983; Card and Agresti, 1987]. For these reasons, the McCabe metric was chosen.

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

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.

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] notes that, calculating V(G) by individual conditions (as McCabe suggests), V(G) is 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. Therefore, one question investigated in this research is a validation of the CYCMIN, CYCMID, and CYCMAX complexity metrics.

2.2 Productivity of New Software Development

While the issue of software development productivity is of practical significance and has generated a considerable amount of research attention, what has often been lacking is detailed empirical data [Boehm, 1987]. Past research can generally be categorized into two broad areas. The first are termed "black box" or "macro" studies which collect data on gross inputs, gross outputs and limited descriptive data in order to build software production models. Examples of this type of work would include the classic work of Walston and Felix [1977] and Boehm [1981] and more recent work by Jeffery and Lawrence [1985] and Banker, et al. [1987].

A second type of research is so-called "glass box" or "micro" models in which the typical methodology is a controlled experiment involving software developers doing a small task where data are collected at a more detailed activity level, rather than the gross level of the "black box" models. Examples of this type of "process" research include Boehm, et al. [1984], and Harel and McLean [1984].

A possible criticism of the first type of research is that by examining high level data, many potentially interesting research questions cannot be answered. For example, one concern is how best to allocate staffing over the phases of the system development lifecycle. Data collected only at the project summary level cannot be used to address this question.

On the other hand, the "micro" level research is sometimes criticized for being artificial. In order

to meet the demands of experimental design and their practical constraints, the researchers are forced either to use small samples or to abstract from reality (e.g. by using undergraduate students as subjects) to such an extent as to bring the external validity of their results into question [Conte, et al 1986].

The current research was designed to mitigate both of these potential shortcomings. The approach taken is of the "macro" production modelling type, but data were collected at an in-depth, detailed level. Input (effort) data were collected by person by week and mapped to contemporaneously collected project phase/task data. Output data included not only SLOC, but McCabe, Myers, and Hansen complexity measures as well.

Therefore these data allow the research to address the following detailed research questions:

- What are the productivity impacts of differing phase distributions of effort? Specifically, does a greater percent of time spent in the design phase improve overall productivity?
- How much does the productivity improve with personnel experience? Are there declining marginal returns to additional experience, and if so, what is their magnitude?

These questions can be answered on the basis of empirical data from real world sites and therefore this research avoids some of the potential external validity concerns of the classic experimental approach.[4]

2.3 Software Maintenance Productivity

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.

Software maintenance is an area of increasing practical significance. It is estimated that from 50-80% of information systems resources are spent on software maintenance [Elshoff, 1976]. However, according to Lientz and Swanson [1980] relatively little research has been done in this area. Macro-type studies have included Vessey and Weber [1983] and Banker, et al. [1987]. Micro-level studies have included Rombach [1987] and Gibson and Senn [1989].

---

[4]Of course, the field study methodology is unable to provide many of the tight controls provided by the experimental approach and therefore requires its own set of caveats. Some of these are discussed in the conclusions section.

One area that is of considerable interest for research is the effect of existing system complexity on maintenance productivity. It is generally 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.

3 DATA COLLECTION AND ANALYSIS

3.1 Data Overview and Background of the Datasite

A prerequisite to addressing the research questions in this study was to collect detailed data about a number of completed software projects, including data on individual personnel. All the projects used in this study were undertaken in the period from 1984-1989, with most of the projects in the last three years of that period. All the data originated in one organization, a defense industry contractor hereafter referred to as Alpha Company. This approach meant that because all the projects were undertaken by one organization, neither inter-organizational differences nor industry differences should confound the results. A possible drawback is that this concentration on one industry and one company may somewhat limit the applicability of the results. However, two factors mitigate this drawback. First, because the concepts studied here are widely applicable across applications, there is no *a priori* reason to believe that there will be significant differences in the trends across industries. Second, the defense industry is a very important industry for software[5] and the results will clearly be of interest to researchers in that field.

The projects at this site were small custom defense applications. 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

---

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

department. The company produced state-of-the-art custom systems primarily for real-time defense-related applications.

At the start of this study, 21 potential projects were identified for possible study. Two of the projects were, however, eliminated as being non-representative of this sample. The first project was a laboratory experiment in which the major task of the chief software engineer was mathematical analysis. While a small amount of software was written for this project, there was no way to separate the analysis effort from the software development effort. The other project was nearly an order of magnitude larger than the next largest project and was believed to be unrepresentative of the Alpha population in its technical design. This project was the only project to employ an array processor, and a substantial fraction of the software developed on the project was micro-code, also non-representative of the Alpha environment. The data set therefore consists of the remaining 19 software projects. These projects represented 20,134 hours (or roughly 11.2 work-years) of software engineering effort, and were selected because of the availability of detailed information provided by the engineers' weekly activity reports.

Table 3.1.1 is a summary of some of the gross statistics on the projects. NCSLOC is the number of non-comment source lines of code; CSLOC includes comments; DURATION is the number of weeks over which the project was undertaken; TOTALHRS is the total number of labor hours expended by software department personnel; NCPROD is the number of non-comment lines of code divided by the total number of labor hours.

	Median	Mean	Std. Dev.
NCSLOC	3143	5456	5820
CSLOC	4390	8192	8791
DURATION	60	62	32
TOTALHRS	639	1059	982
NCPROD	3.52	5.89	5.27

Table 3.1.1 Basic Statistics on Project Data, n = 19

3.2 Data Sources

The data for this study were derived from five sources:

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.

4. **Project Data Collection Forms** -- A participant in each project filled out a simple

questionnaire describing the project.

5. **Engineer Profile Data Collection Forms** -- A survey on the education and experience of every engineer who worked on one of the projects.

The first data source was the source code generated in each project. Of the code developed for the projects, 74.4% was written in Pascal, and the other 25.6% in a variety of other third generation languages. 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 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 number of hours worked on the project and the person charging the project were obtained from the 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.

At Alpha Company, software engineers wrote a summary called an activity report of the work they had done on each project every week. The standard length of these descriptions was one or two pages to describe all the projects. If an engineer worked on only one project, the description of the work for that project would occupy the entire page. If several projects were charged, the length of the descriptions tended to be roughly proportional to the amount of time worked, with the entire report fitting on a page. An example of an activity report appears in Appendix A.

Project data collection forms developed for this research provided background data on each project. The surveys requested descriptions of the project, qualitative ranking of various characteristics of the development process (number of interruptions, number of changes in specifications, etc), and ratings of the quality of the software developed. Unlike the data above, these data are subjective, and results based on it are interpreted cautiously. It is useful, however, to have some measure of these qualitative variables in order to obtain some feel for how the results may be affected. A copy of the survey form used to collect these is included in Appendix B.

The engineer profile forms obtained data on each engineer's educational background, work experience and Alpha company level. If the employee in question was still with the company (24 out of the 34 engineers in the sample), he or she was asked to fill the form out personally. If the employee no

longer worked for the company, personnel records were consulted. A copy of the engineer profile survey is included in Appendix C.

3.3 Categorization of Development Activities

Although unusual for this type of research, this study had access to the week-by-week description of what work was done by each engineer on each project. These descriptions were filled out by the individual engineers every week to inform their managers of the content of their work and also to provide documentation of the work charges in case of a DOD audit. Because these documents were written contemporaneously by everyone who was working on a project, they are believed to present an accurate picture of what happened during the software development process. Approximately 1200 reports were examined for this research.

These activity reports provided a large descriptive database; however, in order to test the hypothesis relating to productivity a key problem was to determine a way to quantify these data. The method used was to categorize the work done each week by each engineer into one of nine standard systems development lifecycle phases employed by Alpha. One advantage of this approach is that these categories corresponded quite closely to the "local vocabulary", i.e., the engineers at Alpha Company tended to organize their work in terms of the above categories. This connection proved very helpful when the activity reports were read, and the work categorized. The nine categories that were used are described in Appendix D.

To categorize a project's activity reports, the researcher selected an engineer and then read through all the engineer's activity reports sequentially. The work done on the project each week was categorized into one or more of the project phases. For the large majority of reports only one phase was charged per report; however, for a fraction of the activity reports the effort was split into two phases.[6]

To obtain consistency across projects and people, and because access to the data was limited to on-site inspection, only one researcher was used to do the categorization task for all the reports. Because the researcher read the activity reports in a sequential manner, he was able to follow the logic of the work, as well as getting the "snapshot" that any one report would represent. Finally, in two cases where the reports were unclear, the engineers were shown their own reports and asked to clarify what was actually done.[7]

---

[6] On a few reports (less than 5%) a maximum of three phases were charged.

[7] Of course a possible limitation of this single rater approach is an inability to check for a possible methods bias. Although a conscious research design, it does introduce a possible source of error.

In some cases, all of the activity reports were not available, typically because the report had been forgotten or lost. In addition, the reports had only been kept for the past three years so that they were generally not available prior to that time.[8] For four of the nineteen projects studied, 25% or more of these data were missing, and no attempt was made to reduce the activity reports for those projects. To obtain the best breakdown of phases for the unassigned time on the remaining 15 projects, such time was categorized in proportion to the categorization of the rest of the project. The median amount of time categorized as unassigned was 9.9% (see Table 3.3.1).

OBS.	TOTAL HOURS	PERCENT UNASSIGNED
1	565	2.8
2	1965	21.1
3	3491	11.1
4	2048	9.9
5	925	22.9
6	2507	24.1
7	761	10.5
8	168	4.1
9	445	14.6
10	182	3.8
11	267	1.8
12	97	7.2
13	1571	21.4
14	147	0.0
15	1779	8.9
Median	761	9.9
Mean	1128	11.0
Std. Dev.	1041	8.1

Table 3.3.1 Hours not Recorded on Activity Reports

3.4 Measure of Software Engineer Experience

As noted earlier, a large amount of research supports the notion that individual productivity varies widely and that experienced professionals are generally more productive. It is, however, possible to gain experience in many different ways. For example, one definition of experience would be simply the number of years of work experience. This method has the problem that it does not take into account the amount of education that an engineer has -- a bachelor's degree is equated with a doctorate. There does not seem to be a generally accepted method to combine the educational background with the years of experience.

Fortunately, for the purposes of this study, Alpha Company categorizes its scientific/engineering staff into five levels (from most senior to least):

[8] A few engineers had their reports available in their own private files.

Scientists:
Level 1: Principal Scientist. Requires peer review to be promoted to this level.

Level 2: Staff Scientist. Requires peer review to be promoted to this level.

Engineers:
Level 3: Senior Engineer. Entry level for Ph.D's. No peer review required.

Level 4: Engineer. Entry level for Master's degree engineers. No peer review required.

Level 5: Junior Engineer. Entry level for bachelor's degree engineers.

A sixth level was defined to include support staff (secretary, etc), part time college students in computer science, and other non-full time professionals. Because many of the projects studied had rather few people, six levels of detail were not required. For this reason, the aggregate levels of scientist (levels 1 and 2), engineer (levels 3, 4, and 5) and support (other) were used. This simple division is also used at Alpha, in that to be promoted to scientist level, a peer review is required. Non-engineering staff were kept in their own group (support staff) because they were only para-professional staff.[9] Table 3.4.1 shows the number of people in each of the levels with each of their highest educational degree levels. All the engineers in this department had technical degrees (most in computer science) so that no attempt was made to separate the different fields.

	< Bachelors	Bachelors	Graduate	Total
Scientist	0	2	8	10
Engineer	1	14	5	20
Support	4	0	0	4

Table 3.4.1 Educational Degrees Categorized by Level

It is clear that the company levels are related to the highest level of education reached. The level is also closely related to the number of years of experience as shown in Table 3.4.2.

	Average Number of Years of Company Experience	Average Number of Years of Total Industry Experience
Scientist	15.6	20.0
Engineer	5.1	5.8

Table 3.4.2: Average Years of Experience for Company Levels

Use of these levels had many advantages. The first was that it provided a reasonable way to combine degree and length of work experience into a single measure, as these levels are good indicators of both experience and educational level. The engineer's compensation is also highly dependent on his or

[9] Only 5.4% of the total labor hours for the projects in the dataset were charged by these para-professionals.

her company level, and therefore project cost is directly related to the use of staff of different levels.[10] Finally, use of the "level" concept was the common practice at Alpha Company, and therefore provided a natural way to discuss the research with the participants.

3.5 Statistics on Source Code

One of the advantages of this data site to the research was that the actual source code was available, thus all statistics of interest about the code could be calculated. In addition to NCSLOC (non-comment, non-blank lines of source code), a second statistic called CSLOC was also collected. CSLOC is NCSLOC plus the number of lines of comment (still excluding blank lines).[11] A potential problem with all SLOC statistics is that they do not take into account the complexity of the code. As discussed earlier, the most popular measure of complexity was developed by McCabe and both Myers and Hansen have offered extensions to McCabe's measure. All of these complexity measures were collected at the module level in order to provide empirical evidence for the relationship among the three measures. The code studied was contained in a total of 465 modules of which 435 were written in Pascal and 30 in FORTRAN.

Collecting these data required metrics programs for each language. Complexity results were collected for the vast majority of code which was in Pascal or FORTRAN. No complexity statistics were generated for two projects (one written in C, the other in ADA). Table 3.5.1 shows the percent of code of each project for which the cyclomatic complexity metrics were generated. They were generated for 17 of 19 projects and, of those, 100% of the code was analyzed for 13 of the 17. In the remaining four projects, an average of 98.3% of the code was analyzed.[12]

---

[10] There is little variation in the salaries of engineers at a given level. Because proposals for projects are costed using a standard cost for each level, but billed using actual costs, large deviations tend to cause client dissatisfaction and are avoided by Alpha Company.

[11] If a comment appeared on the same line as an executable line of code both the comment and the line of code incremented CSLOC. If this method were not used, comments would be significantly undercounted. In particular, the measurement of comment density would be distorted.

[12] To adjust for the fact that complexity figures were not available for the utility routines for four of the seventeen projects, an average complexity ratio (cyclomatic complexity/NCSLOC) was defined for each project. By assuming that the complexity densities remained constant for the project, the total complexity (including non-Pascal or FORTRAN code) was calculated by multiplying the average complexity density by the total NCSLOC. This adjustment was done on only four of the projects, and for less than 2% of the code on average on those four projects. Therefore, this is not believed to be a major source of error.

OBS	NCSLOC (TOTAL)	PERCENT MEASURED	CYCLOMATIC COMPLEXITY	PHASE DATA
1	11730	100.0	YES	YES
2	16363	99.1	YES	YES
3	9879	0.0	NO	YES
4	6682	100.0	YES	NO
5	4287	100.0	YES	YES
6	10218	100.0	YES	YES
7	16324	100.0	YES	YES
8	789	100.0	YES	NO
9	1441	99.5	YES	NO
10	3228	94.9	YES	YES
11	451	100.0	YES	YES
12	3143	100.0	YES	YES
13	395	100.0	YES	YES
14	347	100.0	YES	YES
15	341	100.0	YES	YES
16	221	100.0	YES	YES
17	1228	99.8	YES	NO
18	2147	100.0	YES	YES
19	14453	0.0	NO	YES

Table 3.5.1 Fraction of Code for which Complexity Measures were Obtained

3.6 Summary of Data Available for Study

Table 3.6.1 shows some major statistics for the projects.

PROJECT	NCSLOC	CSLOC	CYCMAX	TOTALHRS	NCPROD
1	11730	16020	1585.9	565	20.76
2	16363	27278	2291.3	1965	8.32
3	9879	18519	N/A	3491	2.82
4	6682	8989	881.9	1932	3.45
5	4287	6845	595.9	2048	2.09
6	10218	13249	1235.9	925	11.04
7	16324	22887	2014.9	2507	6.51
8	789	877	76.9	493	1.60
9	1441	1992	165.7	152	9.48
10	3228	5711	465.8	761	4.24
11	451	589	75.8	168	2.68
12	3143	4390	424.9	445	7.06
13	395	630	71.8	182	2.17
14	347	635	49.9	267	1.29
15	341	465	41.9	97	3.51
16	221	345	34.8	1571	0.14
17	1228	1964	245.2	639	1.92
18	2147	3786	202.9	147	14.60
19	14453	20483	N/A	1779	8.12

Table 3.6.1 Summary of Results of All Projects

NCSLOC is the non-comment lines of code; CSLOC is the number of lines of code including comments; CYCMAX is McCabe's Complexity; TOTALHRS is the total number of hours worked on the

project; NCPROD is NCSLOC/TOTALHRS or the productivity for non-comment lines of code. Not all the data used in the study were available for all the projects. The cyclomatic complexity data are missing from two of the projects and the phase data from four of the projects. However, for most analyses only a single statistic was required, and, in order to conserve degrees of freedom, all projects with valid data in that subset were used. For example, if the analysis required complexity data, but not phase data, the statistics were generated on the seventeen projects with valid complexity data (including the projects without phase data). It is not believed that this approach introduced any systematic bias.

4 SOURCE CODE ANALYSIS

4.1 Comparison of Metrics

For this portion of the analysis it was possible to analyze the data at the module level, since no work-hours data were required. Table 4.1.1 is a summary of some of the basic statistics for all the modules.

Variable	MEAN	STD DEV
NCSLOC	179.5	254.9
CSLOC	258.5	353.5
CYCMIN	21.2	40.1
CYCMID	23.5	42.0
CYCMAX	25.2	45.9

Table 4.1.1 Summary of Module Statistics

Table 4.1.2 gives the Pearson correlation coefficients for the three complexity metrics used in this study. The most striking result is the extent to which similar metrics are highly correlated with each other.[13] 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 significantly different results using CYCMIN or CYCMID instead of CYCMAX. Therefore, the remainder of this study will only report the most widely used metric, McCabe's CYCMAX.

[13] The NCSLOC and CSLOC are highly correlated with a Pearson coefficient of 0.96 (significance is 0.0001). Because of this high correlation, it is reasonable to examine the results using only NCSLOC which is the more common measure.

	CYCMIN	CYCMID	CYCMAX
CYCMIN	1.0000 0.0000	0.9849 0.0001	0.9861 0.0001
CYCMID		1.0000 0.0000	0.9980 0.0001

Note: The number below the correlation is the statistical significance (alpha). (n = 465).

Table 4.1.2 Correlations Metrics for Code Complexity

The length metric, NCSLOC, and the complexity measure, CYCMAX also turned out to be highly correlated (Pearson coefficient, 0.945). This result indicates that the complexity metric is so closely related to the length that any complexity results will be masked by results based on length and is similar to results obtained by other researchers [Li and Cheung, 1987]. Therefore, CMAXDENS is defined as the ratio of cycles per thousand lines of executable code (NCSLOC). These *complexity density* measures are similar to Gilb's logical decisions per statement [Gilb, 1977]. The complexity density metrics have not often been published. CMAXDENS, the average McCabe complexity per thousand lines of code was 121.3 for this sample (standard deviation of 62.7). This result is of the same magnitude as that found by Selby, in his study of 25 NASA projects [Selby, 1988]. His mean was 81.4, with a standard deviation of 57.2.

4.2 New Development Versus Maintenance Results

Of the nineteen projects in the sample, ten were new development and nine were maintenance. To examine possible first order differences between maintenance and new development projects, several statistics were computed. These statistics are displayed in Table 4.2.1.

	New Development Mean	Maintenance Mean	Wilcoxon Rank Sums Z Approximation
NCSLOC	8094	2525	-2.57 **
DURATION	77.2	45.9	-2.21 *
TOTALHRS	1484	588	-2.25 *
NCPROD	7.04	4.61	-1.18

* Significant at 0.1 level.
** Significant at 0.01 level.

Table 4.2.1 Comparison of Development and Maintenance Projects

For new development projects, NCSLOC is a count of all non-comment non-blank physical lines. For maintenance projects, NCSLOC is a count of all non-comment, non-blank lines added. Based on the non-parametric statistical test, the average values of three variables, NCSLOC, DURATION and

TOTALHRS, can be assumed to be different with a high degree of confidence. Because all of these variables are directly related to the size of the project, this difference simply indicates that, on average, the maintenance projects were smaller than new development projects. However, no significant difference in productivity was found. While the average productivity for maintenance projects is less than for development projects, the fact that this difference is not statistically significant means that we were unable to reject the null hypothesis that maintenance productivity is the same as that for development projects. In addition, because the line count ignored changed lines and actually subtracted for deleted lines, the productivity of maintenance projects may be understated. Therefore, the new development projects and the maintenance projects are combined in later productivity analyses, excluding maintenance specific analysis that requires existing code.

5 RESULTS

Four major productivity effects were investigated of which two applied to all of the projects. They were: (a) Are more experienced programmers more productive, and, if so, by how much? and (b) How does the proportion of time spent in the design phase affect the productivity?

The other two effects studied were appropriate for maintenance projects only. As cited earlier, much has been written under the assumption that characteristics of the code have an effect upon maintenance productivity. One of the reasons that McCabe developed his cyclomatic complexity was to attempt to quantify the complexity of code in order to determine the appropriate length of subroutines so that the code would not be too complex to maintain. This study has examined this effect using the complexity density metric defined earlier. The second maintenance productivity issue analyzed was the impact of documentation in the form of comments.

5.1 Experience

It has long been thought and several studies have shown that the experience of the programmers is an important factor in productivity. Verifying this phenomenon is not always easy, in part because engineers with different levels of experience do different tasks. As described above, there were essentially three different categories of technical personnel. The highest category (scientist) included the top two company levels. The employees in this category were primarily project managers. The second category contained the remaining three professional levels and the engineers in this category were primarily responsible for developing the software. The lowest level provided support. Because the scientists and the

support staff were by and large not directly involved in the writing of the software, their effect on productivity is only indirect. Therefore, the research effort focused on the engineer (levels 3, 4, and 5) staff. To test the hypothesis that the higher level (more education and more experience) engineers are more productive, the following model was created:

$$Y = B_0 + (B_1 * P_3) + (B_2 * P_4) + (B_3 * P_5)$$

where,

Y = NCPROD

P_3 = the percentage of the total development hours spent by level 03 engineers.

P_4 = the percentage of the total development hours spent by level 04 engineers.

P_5 = the percentage of the total development hours spent by level 05 engineers.

Development hours are only those hours worked on specifications, design, coding, or debugging. The results of this regression were:

$$\begin{array}{lcccccc} \text{NCPROD} = & -15.75 & + 27.88*P_3 & + & 25.97*P_4 & + & 18.93*P_5 \\ & (-2.03) & (2.89) & & (2.91) & & (1.89) \end{array}$$

$R^2 = 0.45$ Adjusted $R^2 = 0.30$

F-Value = 3.04 n = 15

The interpretation of these results is that adding an additional one percent more time by level 3 engineers working on direct software tasks results in an increase of approximately 2.2 additional NCSLOC/day, *ceteris paribus*. The result for levels 4 and 5 are 2.1 and 1.5 respectively. These results are exactly as expected in that more experienced engineers are more productive. (The t-statistics for the level 3 and 4 engineers are significant at the alpha = .01 level, and at the .10 level for the level 5 engineers.)

By examining the data at this level of detail it is also possible to draw conclusions about the marginal productivity of additional experience, and its value to the firm. In particular, the data show a 37.19% increase in productivity between levels 4 and 5, and a 47.28% increase between levels 3 and 5. What is interesting to note is that the differences in mean salaries for each level is only 15%. Starting at an index salary of 1 for level 1 scientists, this means that the average salary index for a level 3 engineer is .7225 (.85 squared), a level 4 is .6141, and level 5, .5220. Therefore, the net benefit in using a level 3 engineer over a level 5 is approximately 9% (.4728-(1-(.7225/.5220))), and the net benefit of employing

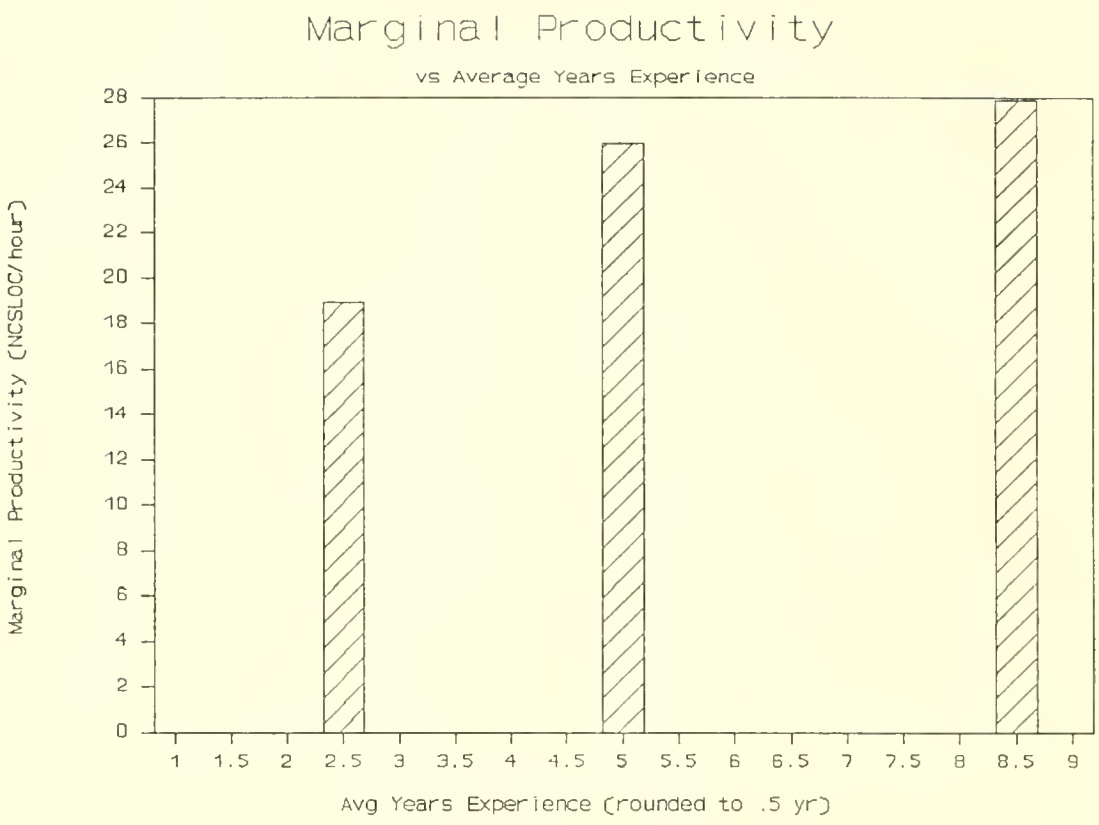

Figure 5.1: Marginal Productivity versus Experience

a level 4 engineer over a level 5 is approximately 20%. This illustrates two points of potential importance to software managers. The first is that more productive staff do not seem to be compensated at a level commensurate with their increased productivity. The intuitive recommendation that follows from this relationship is that managers should seek out higher ability staff, since their marginal benefit exceeds their increased marginal cost. A second result from these data is that there seem to be declining marginal returns to experience (see Figure 5.1). This is consistent with some previous research [Jeffery and Lawrence, 1985]. However, this result may be confounded by a difference in task complexity, i.e., more senior staff may be given more difficult aspects of the system. Further study, perhaps in the form of a lab experiment controlling for task complexity, seems indicated.

5.2 Design Phase Impact

Because of the availability of phase data, it was possible to test the hypothesis developed by McKeen [1983] and Gaffney [1982] that by spending more time in design, the overall productivity of a

project can be improved. Specifically, the more time that is spent analyzing and designing the software, the better and more detailed the design will be. This good design would be expected to lead to a faster and easier overall development process. To test this hypothesis ANALFRAC was defined as the number of hours spent in the specification and design phases divided by the total number of project hours (see Appendix D). The results are plotted in Figure 5.2.

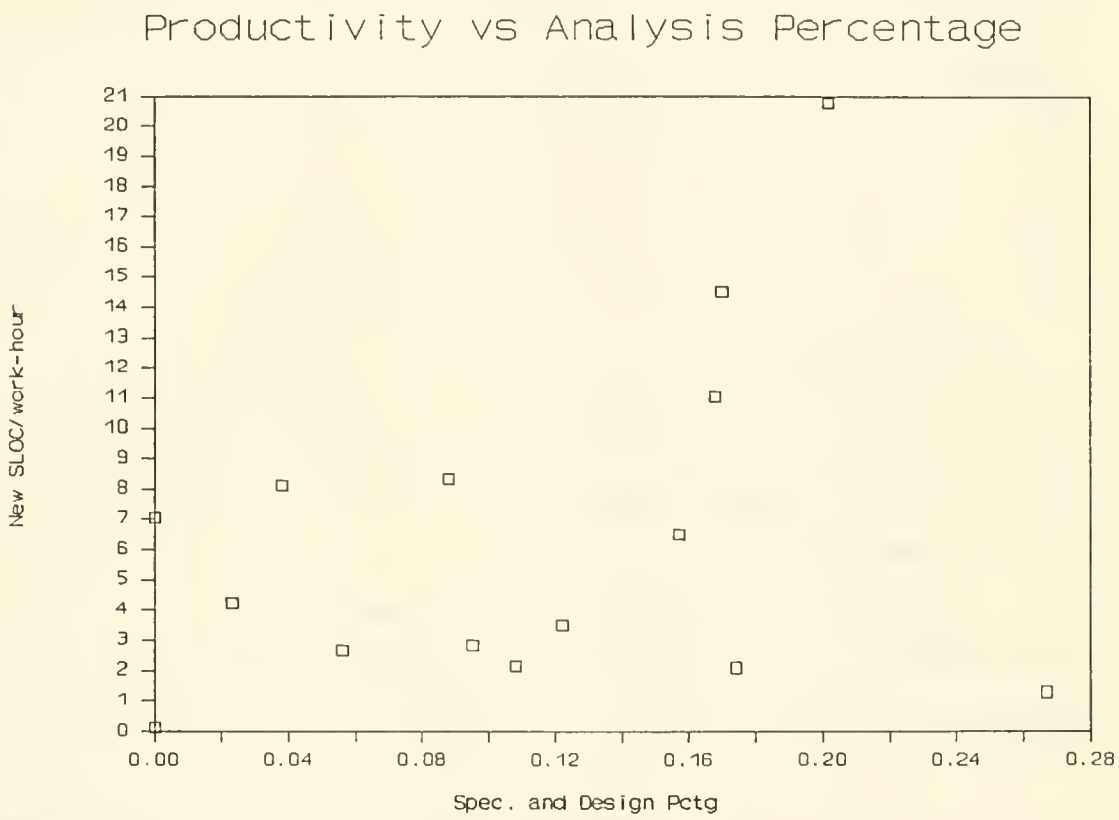

Figure 5.2: Productivity versus Fraction of Time Spent in Analysis

The result for a linear regression model of these data is:

NCPROD = 3.97 + 21.48 * ANALFRAC

(1.56) (1.14)

R^2 = 0.09 Adjusted R^2 = 0.02

F-Value = 1.30 n = 15

Therefore, these data do not provide support for the notion that more time in the specification

and design phases has a measurable impact on source lines of code productivity. Of course the limited number of data points reduce the power of this test.

5.3 Complexity

Complexity density (CYCMAX/NCSLOC) is the appropriate metric for evaluating the effects of code complexity because McCabe's complexity is so closely related to program length. If McCabe's complexity were left 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 complexity. The first is that managers typically do not have a great deal of control over the size of a program (which is intimately connected to the size of the problem). However, by measuring and adopting complexity standards, and/or by using CASE restructuring tools, they can manage 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 an 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.[14]

Logically, the initial complexity density should only affect those phases that are directly related to writing the code. User support, system operation, and management functions should not be directly affected by the complexity of the code. For this reason, a new partial productivity measure was defined:

ALLDPROD: The total number of lines divided by the time spent in all non-overhead activities: specifying, designing, documenting, coding or testing and debugging.

To perform this analysis both the phase data and the cyclomatic complexity data for the maintenance projects were needed. Because all the data were required, only seven data samples were available. A regression of ALLDPROD was performed with the complexity density of the initial code (INITCOMP):

---

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

ALLDPROD = 28.7 - 0.134 * INITCOMP

(3.53) (-2.69)

R^2 = 0.59 Adjusted R^2 = 0.51

F-Value = 7.22 n = 7

This model suggests that productivity declines with increasing complexity density (see Figure 5.3).[15]

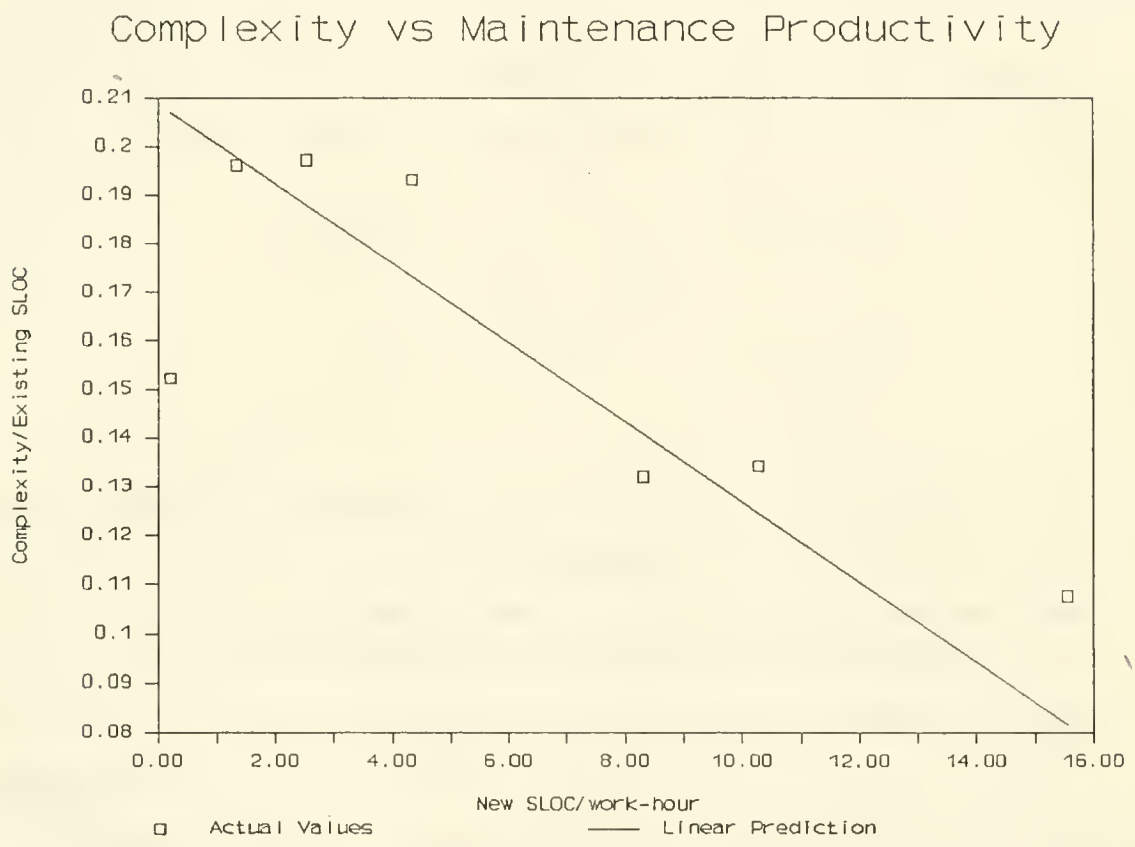

Figure 5.3: Maintenance Productivity versus Complexity Density

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

[15] A possible concern with this model might be multicollinearity between the independent regressor and the intercept term. To test for this, the Belsley-Kuh-Welsch test was run, and no significant multicollinearity was detected [Belsley, et al. 1980].

study is indicated.[16] 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.

5.4 Source Code Comments

The number of comments in the code has also been suggested to affect the maintainability of the code [Tenny, 1988]. A regression of ALLDPROD as a function of the percentage of the lines of code that were comments (PERCOMM) was also performed. (These data are plotted in Figure 5.4.)

$$\begin{array}{lcccl} \text{ALLDPROD} & = & 4.96 & + & 0.05 * \text{PERCOMM} \\ & & (1.04) & & (0.36) \end{array}$$

$R^2 = 0.02$ Adjusted $R^2 = -0.14$

F-Value = 0.13 n = 8

The results indicate that the density of comments does not seem to be related to the maintainability of the code. Because more difficult code may be commented more heavily, it is not clear whether heavily commented code will be easier to maintain. Therefore, another question is whether the programmers adjust their programming styles to the complexity of the code which is being written. For example, one would expect programmers to comment complex code more heavily and to break complex code into more subroutines. If this were the case, the density of comments should increase with the complexity density. In fact, the Pearson correlation coefficient between those two metrics is -0.30 (alpha = 0.0001, n = 465). The interpretation of this negative correlation is that more difficult code get commented less heavily, suggesting that comment density may be simply a function of the time available, rather than being a function of the need for comments.

---

[16] Programming style may also affect productivity. For example, one would expect that by increasing the complexity density (CYCMAX/NCSLOC) of newly written code a programmer would require more time to write these more complex lines of code. The correlation of the complexity density with the productivity is -0.43 (alpha = 0.09), suggesting that productivity does decrease with increasing complexity density.

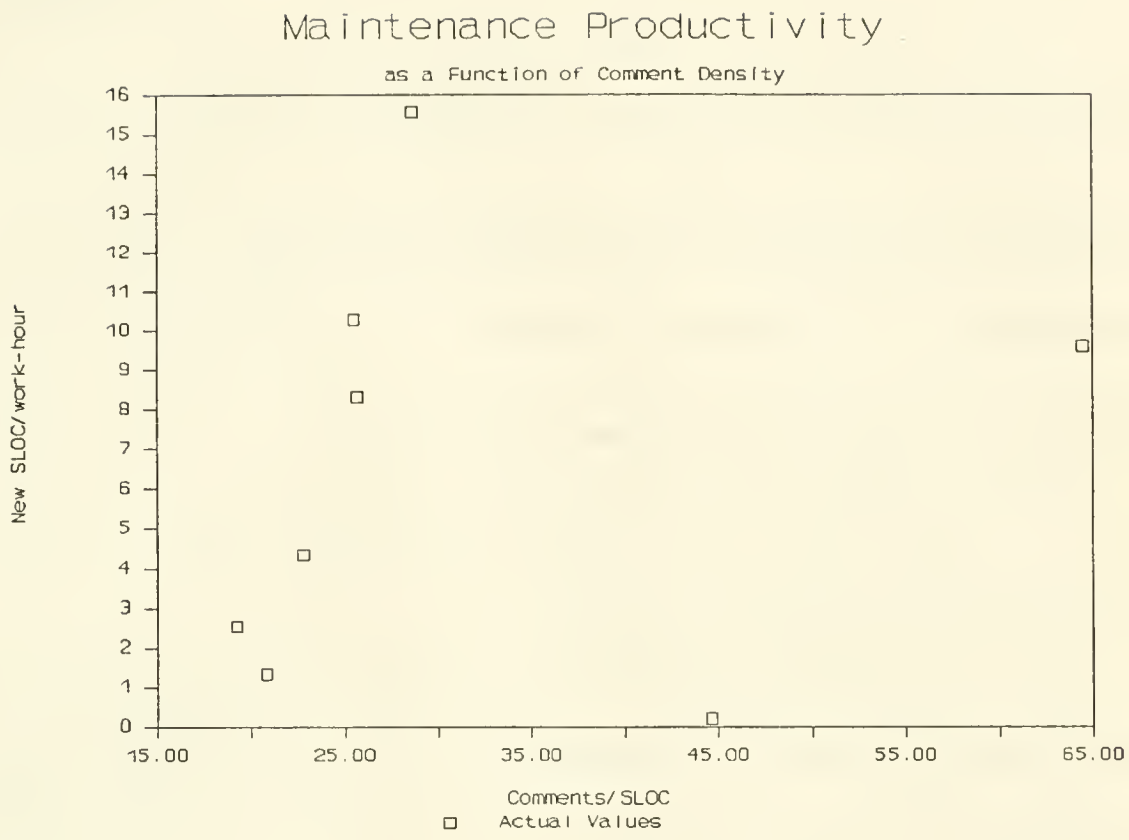

Figure 5.4: Maintenance Productivity versus Comment Density

6 CONCLUSIONS

Through the availability of detailed data about the software projects developed and maintained at this data-site this paper has been able to test a number of relationships between software complexity and other factors affecting development and maintenance productivity. In particular, increased software complexity, as measured by the McCabe cyclomatic complexity density metric, is associated with reduced maintenance productivity. Greater emphasis on the design phase is associated with increased productivity. And, systems developers with greater experience were found to be more productive, although at a declining marginal rate.

How can software development and maintenance managers make best use of these results? To begin with, the measurement of these effects was made possible only through the collection of metrics related to software development inputs and outputs. In particular, use of these data argue for the

collection of detailed data beyond mere SLOC and work-hour measures. Based on these data, software productivity can be improved through employment of higher experience staff and reduction in the cyclomatic complexity of existing code. Staff with higher experience were shown to be more productive, at an average rate *greater* than their increased cost. Finally, cost effective reduction in the cyclomatic complexity of existing code should be attempted. One approach may be by measuring complexity of code as it is written, and requiring that it adhere to an existing standard established through on-site data collection. A second approach may be through existing restructuring products, although, of course, the cost effectiveness of this latter solution is contingent upon the costs of acquiring and operating the restructurer, as well as the size of the existing software inventory. An additional caveat related to this solution is that these restructurers may impose one time additional maintenance costs in terms of initially reducing the maintainers' familiarity with the existing code structure. This likely effect must also be taken into account.

In terms of future research, the software development and maintenance community would benefit from validation of these results on data from other firms particularly from other industries. Also, the effects of complexity on maintenance need to be replicated on a larger sample before accepting these results as guides for standard practice. Beyond these validation-type studies, further work could be devoted to the experience issue, in terms of attempting to determine the causes of the apparent declining marginal returns to experience shown by these data. Research on complexity impacts could examine the actual longitudinal impacts of the use of commercial restructurers. In general, more empirical studies involving detailed data collection will help to provide the insights necessary to help software development progress further towards the goal of software engineering.

7 BIBLIOGRAPHY

[Banker, et al. 1987]
Banker, Rajiv D.; Datar, Srikant M.; and Kemerer, Chris F.; "Factors Affecting Software Development Productivity: An Exploratory Study"; in Proceedings of the Eighth International Conference on Information Systems; December 6-9, 1987, pp. 160-175.

[Basili, et al. 1983]
Basili, Victor R.; Selby, Richard W. Jr.; and Phillips, Tsai-Yun; "Metric Analysis and Data Validation Across FORTRAN Projects"; in IEEE Transactions on Software Engineering; SE-9, (6):652-663, 1983.

[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).

[Belsley, et al. 1980]
Belsley, D.; Kuh, E. and Welsch, R.; Regression Diagnostics; John Wiley and Sons, NY, NY, 1980.

[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.

[Card and Agresti, 1987]
Card, D. N. and Agresti, W. W.; "Resolving the Software Science Anomaly"; Journal of Systems and Software; 7 (1):29-35, March, 1987.

[Chrysler, 1978]
Chrysler, Earl; "Some Basic Determinants of Computer Programming Productivity"; Communications of the ACM; 21 (6):472-483, June, 1978.

[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.

[Curtis, 1981]
Curtis, Bill; "Substantiating Programmer Variability"; Proceedings of the IEEE 69 (7):846, July, 1981.

[Curtis et al., 1981]
Curtis, Bill et al.; "Measuring the Psychological Complexity of Software Maintenance tasks with the Halstead and McCabe Metrics"; IEEE Transactions on Software Engineering; SE-5:99-104, March 1979.

[Dickey, 1981]
Dickey, Thomas E.; "Programmer Variability"; Proceedings of the IEEE 69 (7):844-845, July, 1981.

[Elshoff, 1976]
Elshoff, J. L.; "An Analysis of Some Commercial PL/1 Programs"; IEEE Transactions on Software Engineering; SE2 (2):113-120, 1976.

[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 Zvegintvov, 1983].

[Gaffney, 1982]
Gaffney, John E. Jr.; "A Microanalysis Methodology for Assessment of the Software Development Costs"; in R. Goldberg and H. Lorin, eds.; The Economics of Information Processing; New York, John Wiley and Sons; 1982, v. 2, pp. 177-185.

[Gibson and Senn, 1989]
Gibson, V. and Senn, J.; "Systems Structure and Software Maintenance Performance"; Communications of the ACM; March, 1989, v. 32, n. 3, pp. 347-358.

[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"; inpublished 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.

[Harel and McLean, 1984]
Harel, E. and McLean, E.; "The Effects of Using a Nonprocedural Computer Language on Programmer Productivity"; MIS Quarterly; June 1985; pp. 109-120.

[Jeffery and Lawrence, 1979]
Jeffery, D. R. and Lawrence, M. J.; "An Inter-Organizational Comparison of Programming Productivity"; Proceedings of the 4th International Conference on Software Engineering; pp. 369-377; 1979.

[Jeffery and Lawrence, 1985]
Jeffery, D. R. and Lawrence, M. J.; "Managing Programming Productivity"; Journal of Systems and Software; 5:49-58, 1985.

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

[McKeen, 1973]
McKeen, James D.; "Successful Development Strategies for Business Application Systems"; MIS Quarterly; 7 (3):47-65; September, 1983.

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

[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).

[Rombach, 1987]
Rombach, Dieter; "A Controlled Experiment of the Impact of Software Structure on Maintainability"; IEEE Transactions on Software Engineering; V. SE-13, N3; March 1987.

[Sackman, et al. 1968]
Sackman, H.; Erikson, W. J.; and Grant, E. E.; "Exploratory Experimental Studies Comparing Online and Offline Programming Performance"; Communications of the ACM; 11 (1):3-11, January, 1968.

[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.

[Shen, et al., 1983]
Shen, V.; Conte, S.; and Dunsmore, H.; "Software Science Revisited: A Critical Analysis of the Theory and Its Empirical Support"; IEEE Transactions on Software Engineering; v. SE-9, n. 2, March 1983, pp.155-165.

[Tenny, 1988]
Tenny, Ted; "Program Readability: Procedures vs. Comments"; IEEE Transactions on Software Engineering; v. 14, n. 9; September, 1988; pp. 1271-1279.

[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.

[Walston and Felix, 1977]
Walston, C. E. and Felix, C. P.; "A Method of Programming Measurement and Estimation"; IBM Systems Journal, 16 (1):54-73, 1977.

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

Appendix B: Project Data Collection Form

Project Profile Survey

Project Name: _____

Project Number: _____

Applicable Phase(s): _____

Brief Description of the Project:

_____

_____

_____

_____

_____

Target System:

Mainframe: _____
VAX: _____
HP mini: _____
PC: _____
Other: _____ (please specify).

Can you think of any reasons that would make this software development exceptional (please specify estimate of net effect as well):

_____

_____

_____

_____

Application Characteristics:

Was the code primarily generated from scratch for this particular project (Development Type) or did this project modify or extend an existing program (Maintenance Type):

Development _____ Maintenance _____

Choose the best description of the application:

_____	Real Time:	The primary goal of this system was to control hardware in real time.
_____	Diagnostic:	The system was primarily designed to diagnose an independent hardware system. The software was not an integral part of the working system.
_____	Analysis:	The software performed physical analysis on data.
_____	Utility:	The software performed utility functions (for example in configuration management and general plotting programs).

Software Quality:

Choose the number that best describes the quality of the software resulting from this project compared to the average project (1=Poor,3=Average,5=Excellent) on the following three dimensions.

Correctness:	When the program ran without error, this dimension measures the whether the results matched the desired results.
Reliability:	The likelihood of the program running without error.
Maintainability:	The ease with which modifications and extensions can be made to the software.

	Poor				Excellent
Correctness:	1	2	3	4	5
Reliability:	1	2	3	4	5
Maintainability:	1	2	3	4	5

Management Characteristics:

How the project was managed in comparison with similar projects in terms of the following dimensions:

Time Pressure: How much pressure there was to finish the project in a particular calendar period.

Cost Pressure: How much pressure there was to keep the total software development costs down.

Interruptions: How much work on the project was subject to interruptions from external sources (other projects, vacations, sickness, etc.).

Change in Specs: How often the specifications of the software were altered as the project progressed.

Relative Priority: Priority that the project was given by management relative to other projects.

	Low		Average		High
Time Pressure:	1	2	3	4	5
Cost Pressure:	1	2	3	4	5
Interruptions:	1	2	3	4	5
Change in Specs:	1	2	3	4	5
Relative Priority:	1	2	3	4	5

Environment:

Languages:

Pascal	____
FORTRAN	____
PL/I	____
ADA	____
Assembly	____

Was hardware being developed simultaneously? _____

	Very Stable		Average Stability		Very Unstable
Hardware Stability	1	2	3	4	5

Appendix C: Engineer Profile Form

Engineer Profile Survey

Name: _____

Identification Number: _____

Months of Industry Experience: _____

Months of Alpha Company Experience: _____

Highest Degree Received: _____

Field: _____

Year: _____

Current Alpha Company Level (01,02,03,04,05,Co-op,etc): ___

If promoted within the last four years, give the approximate date of the last promotion: _____

Appendix D: Lifecycle Phases

Specifications -- included any work involved in the understanding or writing of software specifications and requirements. If the engineer reported that he or she was working on writing software specifications, his or her hours were included in this category. Furthermore, this phase also included hours in which an engineer reported having in depth discussions with the customer or other engineers about the software requirements.

Design -- included any time spent determining how to solve a pre-specified problem as well as time spent in writing up the design of a software module was placed in this category. This phase included not only the writing up of a formal design, but also informal designs and discussions with other engineers about how the program should be written.

Coding -- included time spent actually writing software code. This category also included time spent in preliminary debugging prior to the point where a module was determined to be ready to test. (Debugging tasks such as eliminating compile time errors were included in this category). As soon as testing started on a module, the **debug** phase was charged.

Debug -- included testing and debug time done after a module was determined to be ready for testing and/or integration.

User Support -- included time spent by software engineers helping users to run the program. Because many of the software development projects were part of an embedded system, and because many of the maintenance projects involved updating a system, software engineers were often required to help the users run the system. This phase was charged when this support did not involve any of the "traditional" software development tasks.

Documentation -- included all the time that was spent actually writing software documentation such as users' manuals and software maintenance manuals with the exception of software specifications and designs.

Management -- included management meetings, cost estimates, and other such management tasks.

Computer System Maintenance -- included all the charges to support tasks needed to keep the computer systems running properly. In many of the projects, dedicated computer systems were purchased for the projects. When this was the case, time spent backing up the disks or repairing the systems was charged to the project. (Note: on projects that used Alpha Company's general purpose computers, this work was charged to an overhead account and did not appear on the projects' charges.)

Training -- included work done solely in a training mode, i.e., if there was no attempt to produce useable output, this phase was charged. On a few of the projects, engineers were required to spend substantial time learning some aspect of the system (a new computer language, system maintenance documentation). For example, this phase was charged in one project when an engineer spent 20 hours learning Turbo Pascal (TM) by reading the manual and writing small programs unrelated to the project.