1985

The Analysis of Software Development and Testing Processes: An Empirical Study

T. J. Yu

Herbert E. Dunsmore
Purdue University, dunsmore@cs.purdue.edu

Report Number:
85-508

The Analysis of Software Development and Testing Processes:

An Empirical Study

T. J. Yu

H. E. Dunsmore

Department of Computer Sciences
Purdue University
West Lafayette, Indiana 47907

CSD-TR-508
Feburary, 1985

ABSTRACT

One of the goals of researchers in software engineering is to understand the software development process better. The main issues are how to predict programming effort and control program quality. In this experiment, we designed three ways to objectively measure the programmer's ability. From these measures of programmer's ability and the metrics algorithmically derived from the program, we studied their relationships with (1) programming effort, (2) testing effort, and (3) program defects. The data investigated were collected from 44 versions of Pascal programs written by the student subjects. A significant correlation between testing effort and corrected defects was discovered in this study. We also fould 20% of the final defects were introduced during the formal testing process. Some suggestions for further study are provided.

1. Introduction

After many years of software design experience, we now know that it is very hard to control the software development process. One of the major problems is that we cannot predict programming effort accurately. Although there are a lot of models that have been proposed, none of them really give good enough results[1]. Another problem is that we do not know how to control program quality so that a completed program will have no or a very few defects. Although there is typically a significant amount of effort allocated to software testing, for most large software projects there are usually quite a few defects that are not discovered until after the software is placed into operation[2].

In order to understand and control the software development process, we need to be able to quantify some attributes of the product itself, of the development process, and of the personnel involved. Generally we classify software metrics into four categories:

(1) Program metric - This is a measure which can be derived from the program directly, such as program size (e.g., lines of code), the number of decisions (i.e., boolean expressions), or the number of variables. Generally program metrics can be determined objectively by using a software counting tool.

(2) Process metric - This is a measure pertaining to the software development process which is theoretically independent of the product. Examples of such measures are programming time, testing time, and the degree of use of modern software development practices.

(3) Programmer metric - This is a measure related to a programmer's experience, ability, style, etc. It is generally agreed that the programmer is one of the most important factors involved in the software development process. However, it is

---

1. A good cost estimation model should provide the prediction for 80% of estimates within 25% of actuals.

2. From a study of four large software products (each product had about ninety thousands lines of code) from a large company [Shen85], it was found that these four products had 137, 223, 91, and 132 defects each after being delivered to customers.

difficult to find a good way to quantify the ability of programmers. Furthermore, no one likes to be quantified by a single number. In this study we examined several metrics which were related to the programmers' experience and abilities.

(4) Quality metric - Software quality is an elusive concept. It can refer to such distantly–related items as errors present in the software, the "user friendliness" of the system, or even its portability and maintainability. In this study, we considered the quality metric as the number of defects remaining in the program after a certain point. We recognize the importance of other factors of program quality, but there is no agreement about how to measure them objectively. We think it important to investigate this concept using an algorithmic measure.

In recent years a large number of cost and effort estimation model based on program metrics have been proposed [Hals77, Chapter 29 of Boehm81, and Curt84]. In addition, several researchers have proposed models that purportedly can be used to predict the number of occurrences of defects in delivered software modules [Motl77, Otte79, Smit82, Shen85]. We hypothesized that all types of metrics were intercorrelated, and should therefore be analyzed simultaneously. Our analysis below provides support to this hypothesis.

2. Experimental Design

In order to investigate program quality under different testing strategies, we conducted a controlled experiment. The experiment can be called a pretest, posttest, different treatment design [Camp63]. Before the discussion of the experimental design, we would like to introduce the notation used in representing it:

R : Random assignment of experimental subjects

O : Observation during the experiment

X : eXposure to treatment

Our design was used to evaluate the differences of several treatments. In this experiment, we randomly divided the subjects into five groups. Each group was exposed to a testing strategy which was different from others. The experiment design is represented as follows:

$$R\ O_{11}\ O_{12}\ O_{13}\ X_1\ O_{14}\ O_{15}$$

$$R\ O_{21}\ O_{22}\ O_{23}\ X_2\ O_{24}\ O_{25}$$

$$R\ O_{31}\ O_{32}\ O_{33}\ X_3\ O_{34}\ O_{35}$$

$$R\ O_{41}\ O_{42}\ O_{43}\ X_4\ O_{44}\ O_{45}$$

$$R\ O_{51}\ O_{52}\ O_{53}\ X_5\ O_{54}\ O_{55}$$

where O_{i1} : pretest scores (programmer measures)

O_{i2} : program development metrics (program and process measures)

O_{i3} : program defect measures (before)

X_t : different testing strategies

O_{i4} : program testing effort

O_{i5} : program defect measures (after)

We hypothesized that the mean values of software metrics in all groups were the same before the treatment, but would be different after the treatment. That is,

$$O_{11} \approx O_{21} \approx O_{31} \approx O_{41} \approx O_{51}$$

$$O_{12} \approx O_{22} \approx O_{32} \approx O_{42} \approx O_{52}$$

$$O_{13} \approx O_{23} \approx O_{33} \approx O_{43} \approx O_{53}$$

$$O_{14} \not\approx O_{24} \not\approx O_{34} \not\approx O_{44} \not\approx O_{54}$$

$$O_{15} \not\approx O_{25} \not\approx O_{35} \not\approx O_{45} \not\approx O_{55}$$

To test our hypothesis, we employed the statistical test known as "single factor analysis of variance (ANOVA)" [Nete74]. It is used for testing the significance of differences among two or more independent samples. The significance level we selected in this study was .05. That is, when we concluded the group means were *significantly* different, the probability of drawing an incorrect conclusion was only

.05. Because ANOVA analysis might not be appropriate for some metrics which are not *normally* distributed, we also employed a nonparametric statistical test, known as the Kruskal-Wallis rank test [Sieg56]. This was used to test the differences of the rank among groups. As it turned out, the results of nonparametric analysis gave the significance levels similar to that of ANOVA. Therefore, we only present the results of ANOVA.

Besides ANOVA, we were also interested in the correlation between pairs of all the metrics. One of the statistics we used to investigate the correlation is the well-known Pearson correlation coefficient (represented by r). Because the Pearson correlation coefficient might not be appropriate for some metrics, we also used another nonparametric statistic known as the Spearman rank correlation coefficient (represented by s) [Sieg56]. The significance level was also set at .05 for both statistics, unless otherwise specified.

3. Subjects and Empirical Environment

In the summer of 1983, the Department of Computer Science at Purdue University offered a course for graduate students and upper–level undergraduates concerning software metrics and experimental design. The class was eight weeks long. There were 44 students who completed the class and served as the subjects for the experiment reported here. All subjects had taken at least one CS course in advanced programming, algorithm analysis, or data structures. Their programming ability and their familiarity with Pascal were tested by a pretest to ensure that all were qualified to participate in this experiment.

For the purpose of conducting the experiment we set up a run-time system to maintain a log of their activities. The data entered into the file were subjects' log-in times, log-out times, and the progressing versions of their programs as they were submitting.

4. Pretest

The pretest was an attempt to measure the individual differences among the subjects. Programming ability was believed to be an important factor which would be strongly related to programming effort and program quality for the programs developed during the experiment. It is typical to use the number of years of programming experience or to use subjective measures to quantify this factor [Vess83]. Most of our subjects were in the first year of graduate school. Therefore, "number of years experience" would differ very little for them, thus was not a good measure of their abilities. For this experiment, we designed three objective ways to measure their abilities. We used in the pretest a Pascal program (approximately 600 lines of code) which was a small LISP interpreter. This program was used to collect the following measures:

(1) Cloze Procedure

The word *cloze* refers to the human tendency to complete a familiar, but not quite finished, pattern. [Cook84] In a cloze procedure used in this context, the subjects were presented a program listing with every fifth variable[3] in the program replaced with blanks. The subjects were given two hours of class time to fill in 50 blanks in the Pascal program. We enclose a small sample cloze procedure in Appendix 2. The number of correctly filled-in blanks was called the "cloze procedure score".

(2) Extended Cloze Procedure

We were afraid that it would be very easy to complete most blanks correctly by considering the context of the blank. In many cases, knowledge of the Pascal language and observation of other surrounding statements could lead to a good guess as to what should be inserted in the blank. We felt that this could lead to

3. Cook *et al* suggested using tokens (operators, variables, literals, reserve words, and delimiters). But, we think operators, reserve words and delimiters are so language- or context-related that they can be easily guessed by someone who does not really understand what the program is doing. Therefore, we used only variables in our work.

an inflated cloze procedure score. Thus, we concluded that it would be more meaningful to determine how long it took the subject to complete ALL blanks correctly. In order to do this, we modified the process of cloze procedure and arrived at what we call the *extended cloze procedure*. The subjects were asked to log onto the system, to get a copy of the program containing cloze blanks, to change the blanks to what they had written in class, and to run the program until it worked. Obviously, during the latter process it might be necessary to alter what they had put in the blanks several times until the program ran correctly. In order to properly debug the program, the subjects had to understand more about it than was required for a good score on the cloze procedure. Therefore, the time spent debugging the program seemed to be a good measure of their abilities. The amount of time taken to get the program running was called the "extended cloze procedure score".

(3) Comprehension Quiz

The subjects were also given one hour of class time to answer ten multiple-choice questions about the program. The number of correct questions was recorded as their "quiz score".

The measures discussed above were originally used to measure the comprehensibility of the program. In the Spring of 1983, we used these measures in a study involving seven programs and 59 subjects. The results showed little variance among programs, but significant variance among programmers. Therefore, we thought these metrics might help us measure programmer's abilities.

Analysis of data

(1) There was little variance among the cloze procedure scores (see Table 1, 80% of the scores were between 45 and 49). This was probably due to so much time given to the subjects in this process.

(2) There was little variance among the comprehension quiz scores either (All scores were between 4 and 8.). We believe the reason is that some questions

were so easy that everyone answered them correctly, while some questions were so hard that everyone missed them.

(3) Cook *et al* showed that there was a positive relationship between their subjects' cloze procedure scores and comprehension quiz scores [Cook84]. However, from our experiment the Spearman rank correlation coefficient between them was only 0.12 (see Table 6). Therefore, we conclude that they are not *significantly* correlated.

(4) The subjects had great variance in their extended cloze procedure scores. Its range was from 26 minutes to 537 minutes with mean of 107 minutes. In another experiment in the Spring of 1983, the subjects reflected that they did not really understand the program by doing the cloze procedure test, but they thought they fully understood the program after doing the extended cloze procedure test. Therefore, we conclude that the extended cloze procedure is a better measure of a programmer's ability than the other two measures.

5. Program Development

The subjects were asked to write two programs in Pascal. One program was called the *calculator* program; the other was called the *database* program. The calculator program read infix arithmetic expressions, produced the corresponding postfix notations, and printed the computed values of the expressions. The database program was a translator for a simple database query language (DBL). It read text file of a DBL program and produced a Pascal program which was a translation of the input. This study concentrated on only the calculator program. In order to have another measure of individual differences, we used the programming effort to construct the database program as a measure of each subject's programming ability. This metric was called E_{db}.

The development strategy the subjects were directed to use was a top–down and data–structure–first process [Wang84]. There were four major milestones during the development process. The subjects were required to work on their programs in

blocks of time from one to four hours. Each block of time was referred to as a *work session.* The subjects were asked not to work more than eight hours a day on the program to avoid fatigue. After each work session, every subject was interviewed to produce a session report which indicated how much time he spent in this session. The development process can be divided into four stages as follows:

(1) Specification

The subjects received a handout which clearly stated the input and output specifications of the problem. The subjects reported the time they used to study the handout and related materials.

(2) Design

After the specification phase, the subjects started their design. They designed their global data structures first and then the procedures associated with that global data structures. They next defined the local variables for each procedure, but no executable code was written in this phase. This design process was very similar to that employed by programmers using abstract data types. After design, programs were submitted to the Pascal counting tool which recorded the desired metrics.

(3) Coding

During this phase the actual code was written. After each work session, the subjects handed in a compilable version of their programs so that we could record the metrics as the programs evolved during the development process.

(4) Minimal Acceptance Testing

We gave the subjects six test cases. They used these test cases to detect and correct errors in their programs.

The effort from (2) to (4) was accurately collected from their session reports and was recorded as their programming effort (E_p). The effort involved in constructing comment lines was excluded from this metric.

Software Metrics Collected during Programming Development

(1) *loc* - lines of code. This was a count of the declarative and executable statements in the program. Comments and blank lines were excluded.

(2) $v(G)$ - McCabe's cyclomatic complexity [McCa76]. This was a count of the conditional statements, loops, procedures (including the main program), and binary Boolean operators such as AND and OR.

(3) *vars* - number of unique variables

The remainder of the program metrics we considered were originally proposed by Halstead [Hals77]. They were collectively called Software Science Metrics.

(4) η_1 - number of unique operators

(5) η_2 - number of unique operands

(6) N_1 - total number of operators

(7) N_2 - total number of operands

(8) *Program Length* : $N = N_1 + N_2$

(9) *Estimated Length* : $\hat{N} = \eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2$

(10) *Volume* : $V = N \times \log_2(\eta_1 + \eta_2)$

(11) *Difficulty* : $D = \frac{\eta_1}{2} \times \frac{N_2}{\eta_2}$

(12) *Software Science Effort* : $E = D \times V$. We divided this number by 18×3600 so that it had the unit of "hours".

Analysis of data

(1) *loc*, N, and *Volume* were highly[4] correlated with each other (see Table 3). We expected this, because all three are measures of program size. We also found that $v(G)$ was highly correlated with these three size measures. Because Pascal

4. When we say two metrics are highly correlated, we mean their R^2 is larger then .50, where R^2 is the coefficient of multiple determination. This implies that more than 50% of the total variance of one metric can be explained by the other.

was designed to be a structured language and programmers were trained to write structured programs[5], we concluded that: "$v(G)$ for a highly structured program has a strong size component."

(2) Unique operands (η_2) and unique variables (*vars*) were highly correlated (r=.79). Although they seemed to be good measures of the complexity of data structures, they were correlated with neither programming effort nor program defects.

(3) *Difficulty* (D) and *Software Science Effort* (E) were highly correlated with program size. But, neither one of them was related to either programming effort or program defects.

(4) $\hat{N}$ was not correlated with N (let alone equal to N). However, we found $\hat{N}$ was highly correlated with η_2 (r = 0.96). It appeared that for these Pascal programs the number of operators η_1 reaches a near–maximum level of about 60. Thus, $\hat{N}$ became simply a linear function of η_2.

(5) E_{db}, the programming effort of the database program, was highly correlated with the programming effort (E_p) of this calculator program (r = .68). From Figure 1, it showed a strong relation between E_p and E_{db}. We conclude that there is a tendency for some programmers to be faster than others regardless of the program involved.

(6) No program metrics were better correlated with E_p than *loc*. Although the correlation between E_p and *loc* was significant (r = .39), from Figure 2 we thought it was not good to use *loc* alone to estimate programming effort. We also used multiple linear regression to find the multiple relation between E_p and other program metrics. Because most program metrics were size related, there was no combination of others that did significantly better than *loc* alone.

(7) Because the distribution of the programmer metrics was not a normal distribution, we thought it would be better to use the nonparametric Spearman rank

5. Students in the department of computer science at Purdue University are not allowed to use the goto statement in their Pascal programs.

test to find their correlations with other metrics. We found that extended cloze score and programming effort were significantly correlated at α level equal to 0.1 (s=.28). No other programmer metrics had better performance than the extended cloze procedure. From Figure 3, we could see a weak relation between these two metrics. The relation was very weak, because we believe that programming effort involves many other important factors which are not related to the programmer's ability.

6. Formal Testing

In order to test the variability of program quality under different testing strategies, we designed five separate sets of test data. Each set contained the same number of cases (10), but a varying number of unique tests. The easiest set of test data contained two unique tests, but had multiple copies of each test. The hardest set of test data contained ten unique tests. The other sets of test data contained four, six, and eight unique tests. The information collected in this phase was

(1) E_t : effort for running these test cases

(2) E_f : effort for fixing the errors.

(3) E_{t+f} : total testing effort = $E_t + E_f$

Analysis of data

We used ANOVA to investigate the variance among groups. The results appear in Table 7. As we expected, there was no variance among the groups before formal testing in terms of any metrics. (That was because, up to that point, they had all been given exactly the same task). We did expect to see significant variability of testing effort (E_t *or* E_f) among the different groups. However, such a result did not occur. We will discuss later why we think this happened.

7. Defect Measurement

Researchers have different opinions concerning what is a software defect. By Myers's definition [Myer76], "A software error is present when the software does not do what the user 'reasonably' expects it to do." In order to quantify software defects algorithmically, we counted the number of failed test cases as the defect measure. We constructed extensive sets of test cases which covered almost every possibility in the input domain. We ran each program on these test cases. The number of failed test cases was defined as the number of defects in the program. There were two major drawbacks of this process:

(1) In spite of our attempts to generate independent test cases, some were equivalent, i.e., test case A would fail, if and only if test case B failed.

(2) Some test cases had a partial ordering relation, i.e., test case A would fail if test case B failed, while the failure of test case A did not imply the failure of test case B.

To avoid these two problems we tried to design test cases which were as independent as possible. Besides this, we used clustering analysis to find equivalence classes of test cases and to delete any redundant test case results. For this program, ws had first designed 55 test cases. After clustering analysis, 50 test cases remained for further examination. The following measures of defects were collected:

D_1 : defects in the program before formal testing

D_2 : defects still in the program after formal testing

$D_{dif} = D_1 - D_2$

D_{find} : defects found during formal testing

D_{new} : defects introduced during formal testing $= D_2 - (D_1 - D_{find})$

D_{old} : defects undetected during formal testing $= D_2 - D_{new} = D_1 - D_{find}$

Analysis of data

(1) From the ANOVA analysis, we found that the group means of any defect measures were not *significantly* different (see Table 7). This result implied :

   (a) The variance of other unknown factors was more significant than the variance of test cases, or

   (b) The variance of test cases was so small that we can not observe any difference among them, or

   (c) Different sets of test cases really had no effect on the program quality. (i.e., hard or easy tests made no difference.)

(2) Most defect measures had significant correlations with testing effort (see Table 5.). The strongest correlation was found between D_{find} and E_{t+f} (r=.51). This relation can also be observed from Figure 4. We conclude that the more effort our subjects spent in testing, the more defects they found.

(3) A large portion of defects were introduced during formal testing.

$$\frac{\sum D_{new}}{\sum D_2} = 20\%$$

That is, 20% of the final defects resulted from the formal testing process. It was also interesting to point out that there were three programs which had more defects after formal testing.

(4) Although there were quite a few defects caused by formal testing, we do not advocate abandoning it. In this study, about 63% of the defects were removed by formal testing.

8. Productivity and Defect Density

Researchers in software engineering frequently use the productivity measure which is defined as

$$P = \frac{loc}{E_p}$$

to show the performance of the programmers or the complexity of the problems. It is also very popular to use defect density which is defined as

$$DD = \frac{D_1}{loc}$$

to show the quality of the programs. In this study, we also investigated the relationship of productivity and defect density to other metrics. However, we cannot find any metrics related to them except size metrics. By examining Figure 5 and Figure 6, we find larger programs seem to have higher productivity and smaller defect density. However, we cannot make this conclusion because in general all programs in our study are doing the same tasks. We should conclude that the programmer who spent less effort was more productive regardless of the size of his program, and the program which had fewer defects was better regardless of its size. The data appearing in Figure 5 and Figure 6 make us believe that the traditional definitions of productivity and defect density may not be appropriate. Both of them tend to encourage programmers to write large programs. Such results are also observed in [Basi84] and [Shen85]. Therefore, we suggest that these two measures need further study.

9. Conclusion

In this experiment, we have studied programmer metrics, program metrics, process metrics, and defect metrics. The most interesting inferences we make from this work are

(1) A programmer will probably exhibit high productivity if he has been highly productive in the past. This also suggests that the best way to measure a programmer's programming ability is to measure his performance in writing other programs.

(2) The more testing effort, the more defects we will find. But, we may also introduce more defects. A good future research topic would be to study the testing

process so that we can decide how to allocate optimal effort in testing.

(3) Software science metrics are of very little utility. They are all size-related measures. If program size (i.e. *loc*) has a correlation with some other measures, so do Software Science metrics. If program size does not, neither do Software Science metrics.

(4) It is not a good idea to use program size alone to estimate programming effort. Program size is related to programming effort, but it can only explain 15% of the total variance of programming effort in our study. Therefore, we conclude that any effort estimation model should include program size and other factors which are independent of program size.

(5) Except for testing effort, no other metrics were significantly correlated with any defect measures. We think this is due to the characteristic of this controlled experiment. When the subjects were all doing the same task, their performance mostly depended on individual differences which we have no good way to quantify. It also reveals that the software testing process is much harder to control than the software development process.

(6) The extended cloze procedure is better than the cloze procedure score or comprehension quiz score in terms of the relation with programming effort. However, this measure has little use if we want to include it in the effort estimation model. Further research is needed to design a better measure of individual differences or to revise the effort estimation model.

There were, of course, some problems in conducting this experiment:

(1) "Defects" were not well-defined. Several defects may require only one program change, while one defect may require several program changes. It is hard to judge if this defect measure falls onto an interval scale (or even an ordinal scale). Furthermore, the quality of a program is related not only to the number of defects, but also to the severity of each defect.

(2) Our formal testing procedure was not well–designed. We should have had a greater variability among test cases, so that we could see greater variability in program defects after formal testing.

Through this controlled experiment, we found a few relationships among software metrics. If we want to control the software development process, we need to put more engineering discipline into it. Otherwise, we will not be able to control the software development process successfully.

Acknowledgements

The authors would like to thank Andrew Wang for his work on setting up the run-time system to collect all programs and program metrics. Thanks to Francie Newbery for her contributions in administering this experiment and collecting programmer metrics. Thanks to Cristina Ruggieri for designing a thorough and complete set of test cases for the calculator programs. Thanks to Mark Pasch for collecting the defect metrics for those programs. Vincent Shen made valuable suggestions regarding this experimental design and this paper. We are also indebted to all students in the class of CS 590E, Summer 1983. Without their cooperation, this research would not have been possible.

This work has been sponsored by the IBM Corporation through the Santa Teresa Laboratory, San Jose, California and by the U.S. Army Institute for Research in Management Information and Computer Systems, Atlanta, Georgia.

References

[Basi84] Basili V.R. and Perricone T. Software errors and complexity : an empirical investigation. *Communications ACM* 27,1 (1984), 42-42.

[Boeh81] Boehm, B. W. *Software Engineering Economics*. Prentice-Hall, Englewood Cliffs, NJ (1981).

[Camp63] Campbell, D. T. and J. C. Stanley. *Experimental and Quasi-Experimental Designs for Research.* Rand McNally and Co., Chicago, IL (1963).

[Cook84] Cook, C, W. Bregar and D. Foote. A preliminary investigation of the use of the cloze procedure as a measure of program understanding. *Information Processing and Management* 20, 1-2 (1984), 119-208.

[Curt84] Curtis, B., I. Forman, R. Brooks, E. Soloway and K. Ehrlich. Psychological perspectives for Software Science. *Information Processing and Management* 20, 1-2 (1984), 81-96.

[Hals77] Halstead, M. H. *Elements of Software Science.* Elsevier North-Holland, New York, NY (1977).

[McCa76] McCabe, T. J. A complexity measure. *IEEE Transactions on Software Engineering* SE-2, 4 (1976), 308-320.

[Motl77] Motley, R. W. and W. D. Brooks. Statistical prediction of programming errors. *RADC-TR-77-175*, Rome Air Development Center, Griffiss AFB, NY (May 1977).

[Myer76] Myers G.E. *Software Reliability : Principles and Practices.* Wiley, New York, NY (1976).

[Nete74] Neter, J. and W. Wasserman. *Applied Linear Statistical Models.* Irwin, Inc., Homewood, Illinois (1974).

[Otte79] Ottenstein, L. M. Quantitative estimates of debugging requirements. *IEEE Transactions on Software Engineering* SE-5, 5 (1979), 504-514.

[Sieg56] Siegel, S. *Nonparametric Statistics for the Behavioral Science.* McGraw-Hill Publishing Co., NY (1956).

[Shen85] Shen, V.Y., Yu, T.J., Thebaut, S.M., and Paulsen L.R. Identifying error-prone software: an empirical study *IEEE Transactions on Software Engineering* To be published on April 1985.

[Smit82] Smith, C. P. Practical applications of Software Science – the detection of error prone code. *TR 03.184*, IBM Santa Teresa Laboratory, San Jose,

CA (February 1982).

[Vess83] Vessey, I. and Weber, R., Some factors affecting program repair maintenance: an empirical study. *Communication ACM* 26,2 (1983), 128-136.

[Wang84] Wang, A. S. The estimation of software size and effort: an approach based on the evolution of software metrics. Ph.D. Thesis, Department of Computer Science, Purdue University (August 1984).

Table 1. Domain of Software Metrics

Data Distribution							
	Low	Low+1	Mean	Median	Std-Dev.	High-1	High
cloze	11	19	44.386	47	7.456	49	49
extend(min)	26	31	107.023	88.5	80.973	217	537
quiz	4	5	6.227	6	1.008	8	8
E_{db}	8.72	10.38	17.28	15.67	5.90	27.67	41.35
loc	267	314	463	460.5	96.1	656	784
N	1146	1319	1988	1955.5	550.8	2574	4841
$v(G)$	68	89	128	126	32.4	232	236
vars	40	41	64	63	15.5	105	118
η_1	60	60	70.705	70.5	5.975	80	88
η_2	80	84	110.682	112	16.822	157	157
$\hat{N}$	927	955	1188	1170	158	1589	1636
V	8411	9750	14907	14691	4111	20235	35489
E_{ss}	24.11	27.74	59.59	52.59	49.79	86.70	365.07
D	163.7	170.9	241.4	232.7	76.3	321.1	666.6
E_p (hr)	15.92	16.25	26.93	26.73	8.58	49.80	54.32
E_i (min)	5	5	18.61	15	157	59	90
E_f (min)	0	0	20	15	21.21	80	93
E_{i+f}	5	5	38.6	30	31.0	114	152
D_1	1	1	6.80	6.5	3.7	14	17
D_2	0	0	3.11	3	2.3	9	11
D_{dif}	-3	-2	3.68	3.5	3.190	10	11
D_{find}	0	0	4.30	4.5	2.741	10	11
D_{new}	0	0	.61	0	1.298	5	5
D_{old}	0	0	2.5	2	1.874	7	9

Table 2. Correlation Matrix of Programmer Metrics

	cloze	extend	quiz	E_{db}
cloze	1.00	-.24	.05	-.48
extend	-.24	1.00	.14	.30
quiz	.05	.14	1.00	-.16
E_{db}	-.48	.30	-.16	1.00
loc	-.10	.21	-.26	.28
N	.09	.10	-.20	.12
$v(G)$	-.11	.09	-.05	.17
vars	.18	.02	.11	-.07
η_1	-.26	-.06	-.15	.08
η_2	-.06	.12	.07	-.02
$\hat{N}$	-.13	.09	.02	.01
V	.08	.10	-.20	.12
E_{ss}	.07	0.00	-.21	.06
D	.11	.01	-.22	.10
E_p	-.25	.17	-.27	.68
E_t	0.00	-.06	.15	0.00
E_f	.17	-.09	-.03	-.08
E_{t+f}	.12	-.09	.05	-.06
D_1	.18	.06	-.19	.17
D_2	.13	.12	-.06	.08
D_{dif}	.12	-.01	-.18	.14
D_{find}	.19	-.09	-.18	.10
D_{new}	.11	-.17	.05	-.13
D_{old}	.08	.26	-.11	.19

Table 3. Correlation Matrix of Program Metrics

	loc	N	$v(G)$	$vars$	η_1	η_2	$\hat{N}$	V	E_{ss}	D
cloze	-.10	.09	-.11	.18	-.26	-.06	-.13	.08	.07	.11
extend	.21	.10	.09	.02	-.06	.12	.09	.10	0.00	.01
quiz	-.26	-.20	-.05	.11	-.15	.07	.02	-.20	-.21	-.22
E_{db}	.28	.12	.17	-.07	.08	-.02	.01	.12	.06	.10
loc	1.00	.82	.74	.11	-.08	.22	.17	.83	.68	.63
N	.82	1.00	.74	.15	-.19	.18	.11	1.00	.93	.89
$v(G)$	.74	.74	1.00	.03	-.02	.20	.17	.74	.64	.60
$vars$	.11	.15	.03	1.00	.23	.79	.77	.22	-.05	-.13
η_1	-.08	-.19	-.02	.23	1.00	.27	.52	-.14	-.15	-.15
η_2	.22	.18	.20	.79	.27	1.00	.96	.26	-.06	-.23
$\hat{N}$	.17	.11	.17	.77	.52	.96	1.00	.18	-.10	-.25
V	.83	1.00	.74	.22	-.14	.26	.18	1.00	.91	.86
E_{ss}	.68	.93	.64	-.05	-.15	-.06	-.10	.91	1.00	.96
D	.63	.89	.60	-.13	-.15	-.23	-.25	.86	.96	1.00
E_p	.39	.33	.29	.05	.11	.10	.12	.33	.27	.28
E_t	-.29	-.16	-.12	-.10	-.26	-.12	-.18	-.17	-.13	-.11
E_f	-.10	.03	-.04	0.00	-.07	.04	.01	.03	.05	.04
E_{t+f}	-.21	-.06	-.08	-.05	-.17	-.03	-.08	-.07	-.03	-.03
D_1	-.02	.10	-.03	-.25	-.21	-.24	-.27	.07	.18	.20
D_2	-.05	-.08	.01	-.26	-.02	-.13	-.12	-.09	-.01	-.05
D_{dif}	.01	.17	-.04	-.10	-.24	-.18	-.23	.15	.22	.26
D_{find}	-.05	.13	-.02	-.15	-.28	-.16	-.22	.11	.18	.20
D_{new}	-.12	-.15	.06	-.07	-.02	.11	.09	-.15	-.15	-.23
D_{old}	.02	.01	-.03	-.27	-.01	-.24	-.22	-.01	.09	.10

Table 4. Correlation Matrix for Process Metrics

	E_p	E_t	E_f	E_{t+f}
cloze	-.25	0.00	.17	.12
extend	.17	-.06	-.09	-.09
quiz	-.27	.15	-.03	.05
E_{db}	.68	0.00	-.08	-.06
loc	.39	-.29	-.10	-.21
N	.33	-.16	.03	-.06
$v(G)$	.29	-.12	-.04	-.08
vars	.05	-.10	0.00	-.05
η_1	.11	-.26	-.07	-.17
η_2	.10	-.12	.04	-.03
$\hat{N}$	.12	-.18	.01	-.08
V	.33	-.17	.03	-.07
E_{ss}	.27	-.13	.05	-.03
D	.28	-.11	.04	-.03
E_p	1.00	-.14	.24	.10
E_t	-.14	1.00	.45	.79
E_f	.24	.45	1.00	.90
E_{t+f}	.10	.79	.90	1.00
D_1	.06	.37	.45	.49
D_2	.06	.03	.23	.17
D_{dif}	.03	.41	.35	.44
D_{find}	-.02	.45	.42	.51
D_{new}	-.10	-.06	.02	-.02
D_{old}	.15	.07	.28	.22

Table 5. Correlation Matrix for Defect Measures

	D_1	D_2	D_{dif}	D_{find}	D_{new}	D_{old}
cloze	.18	.13	.12	.19	.11	.08
extend	.06	.12	-.01	-.09	-.17	.26
quiz	-.19	-.06	-.18	-.18	.05	-.11
E_{db}	.17	.08	.14	.10	-.13	.19
loc	-.02	-.05	.01	-.05	-.12	.02
N	.10	-.08	.17	.13	-.15	.01
$v(G)$	-.03	.01	-.04	-.02	.06	-.03
$vars$	-.25	-.26	-.10	-.15	-.07	-.27
η_1	-.21	-.02	-.24	-.28	-.02	-.01
η_2	-.24	-.13	-.18	-.16	.11	-.24
$\hat{N}$	-.27	-.12	-.23	-.22	.09	-.22
V	.07	-.09	.15	.11	-.15	-.01
E_{ss}	.18	-.01	.22	.18	-.15	.09
D	.20	-.05	.26	.20	-.23	.10
E_p	.06	.06	.03	-.02	-.10	.15
E_t	.37	.03	.41	.45	-.06	.07
E_f	.45	.23	.35	.42	.02	.28
E_{t+f}	.49	.17	.44	.51	-.02	.22
D_1	1.00	.52	.78	.87	-.08	.70
D_2	.52	1.00	-.12	.13	.59	.83
D_{dif}	.78	-.12	1.00	.92	-.52	.21
D_{find}	.87	.13	.92	1.00	-.14	.26
D_{new}	-.08	.59	-.52	-.14	1.00	.03
D_{old}	.70	.83	.21	.26	.03	1.00

Table 6. Spearman Rank Correlation Matrix

	cloze	extend	quiz	D_1	D_2	D_{dif}	D_{find}	D_{new}	D_{old}
cloze	1.00	-.48	.12	-.07	-.02	-.06	-.10	.17	.02
extend	-.48	1.00	-.06	.09	-.09	.12	.03	.08	.09
quiz	.12	-.06	1.00	-.18	0.00	-.15	-.14	.29	-.14
D_1	-.07	.09	-.18	1.00	.57	.79	.87	.17	.74
D_2	-.02	-.09	0.00	.57	1.00	.01	.24	.62	.79
D_{dif}	-.06	.12	-.15	.79	.01	1.00	.92	-.12	.33
D_{find}	-.10	.03	-.14	.87	.24	.92	1.00	.15	.34
D_{new}	.17	.08	.29	.17	.62	-.12	.15	1.00	.21
D_{old}	.02	.09	-.14	.74	.79	.33	.34	.21	1.00
loc	-.07	.33	-.12	-.07	-.02	-.07	-.15	.13	.09
N	.12	.19	-.14	-.03	-.09	.05	-.03	.15	0.00
$v(G)$	-.10	.22	-.02	-.11	-.01	-.10	-.14	.24	.01
$vars$	.25	-.01	.16	-.24	-.28	-.07	-.14	.17	-.26
η_2	-.03	.15	.11	-.27	-.14	-.20	-.18	.32	-.27
E_p	-.25	.27	-.19	.11	.18	.04	0.00	.23	.23
E_t	-.03	-.08	.16	.27	.11	.25	.33	.26	.07
E_f	-.02	-.06	.07	.57	.37	.48	.56	.30	.36
E_{t+f}	-.05	-.07	.06	.58	.33	.50	.59	.27	.32

Table 7. Mean value of metics in each group

Group Mean							
Test Cases	2	4	6	8	10	$F(4,39)$[1]	α
Number(44)	7	8	9	9	11		
Cloze	45.7	46.1	40.1	44.8	45.5	0.96	
Extend(min)	72.14	141.9	118.6	120.7	83.27	1.05	
Quiz	6.14	6	6.22	6.11	6.54	0.39	
loc	405	503	501	467	436	1.65	
N	1642	2257	2103	2043	1873	1.47	
$v(G)$	108	137	134	135	125	0.97	
vars	55	59	74	66	63	1.89	
η_1	72	70	71	70	71	0.19	
η_2	96	106	121	116	110	3.09	$< .05$
E_p (hr)	25.49	29.23	30.72	23.89	25.56	0.98	
E_t (min)	12.29	13.38	18.56	28.67	18.27	1.66	
E_f (min)	10	10.63	26.67	23.11	25.12	1.24	
D_1	5.14	7.25	5.67	7.78	7.64	0.879	
D_2	2.71	3.50	2.33	3.11	3.73	0.532	
D_{dif}	2.43	3.75	3.33	4.67	3.91	0.502	
D_{find}	2.57	4.38	3.33	5.67	5.00	1.851	
D_{new}	.14	.63	.00	1.00	1.09	1.347	
D_{old}	2.57	2.88	2.33	2.11	2.64	0.196	

1. $F(4,39)$ is F statistic of degree of freedom 4 and 39.

RANGE OF X AXIS: 8 55
RANGE OF Y AXIS: 8 55

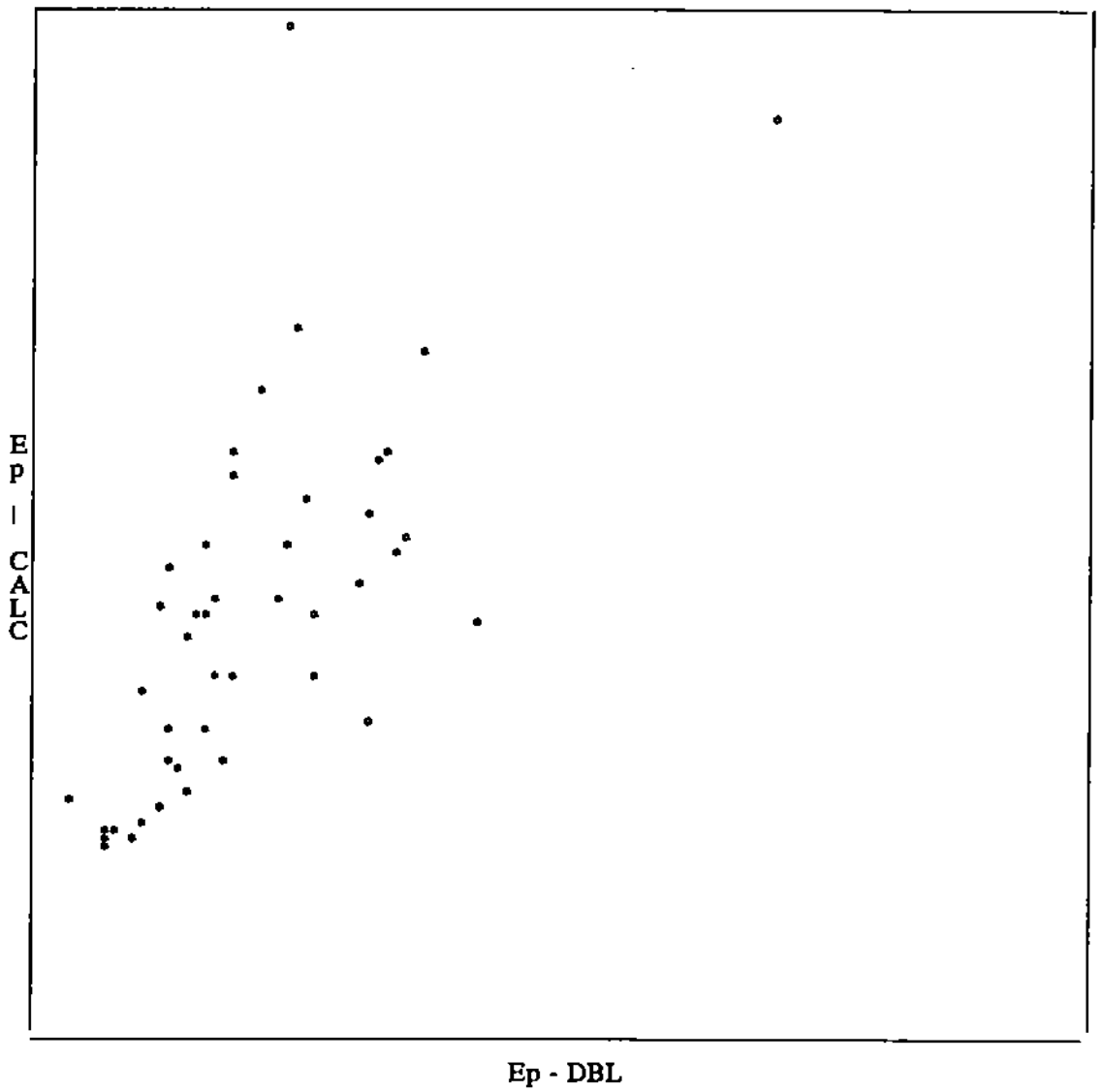

Figure 1. Relation of Programming Effort

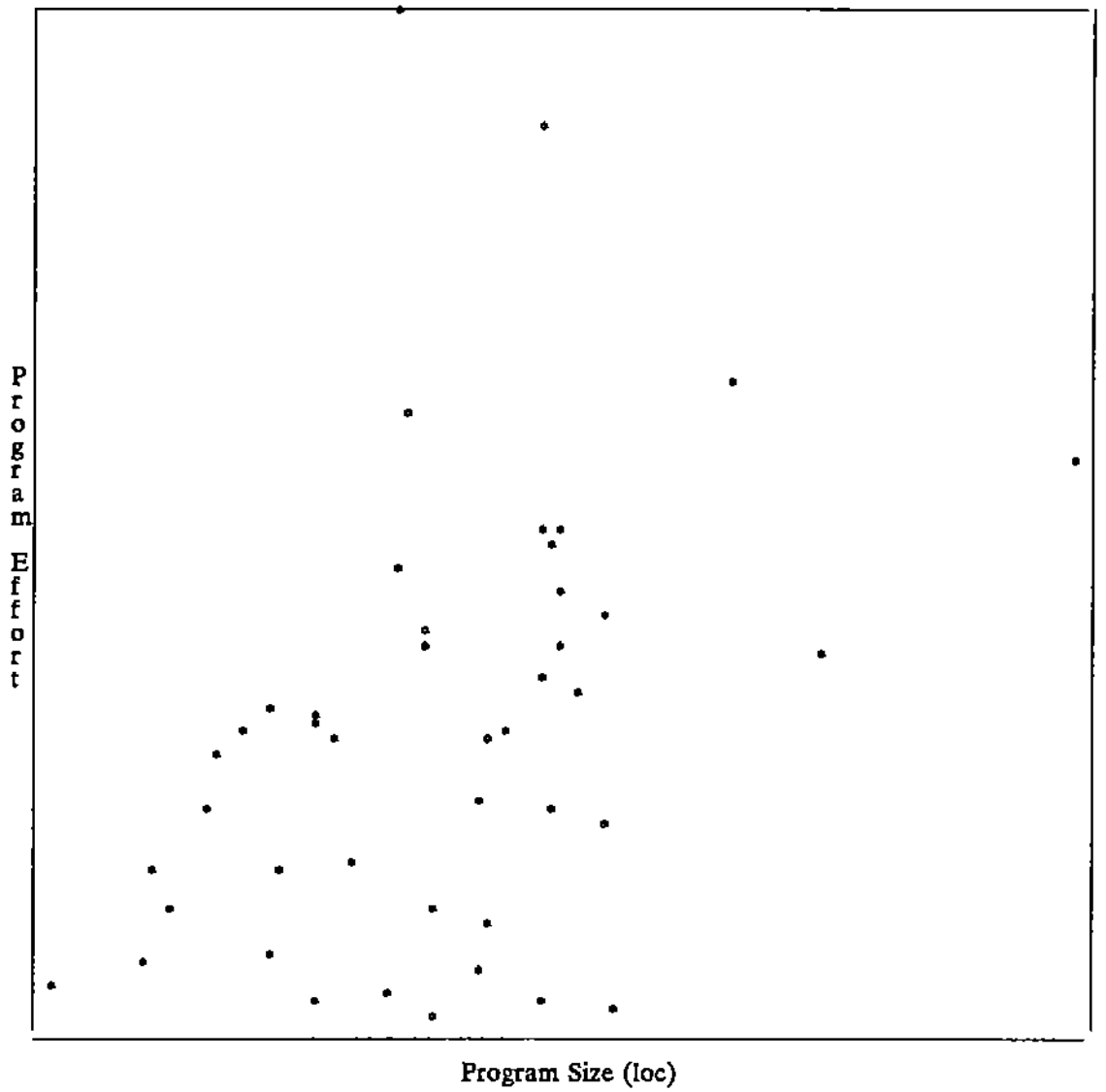

Figure 2. Programming Effort vs. Program Size

RANGE OF X AXIS: 0 220
RANGE OF Y AXIS: 10 60

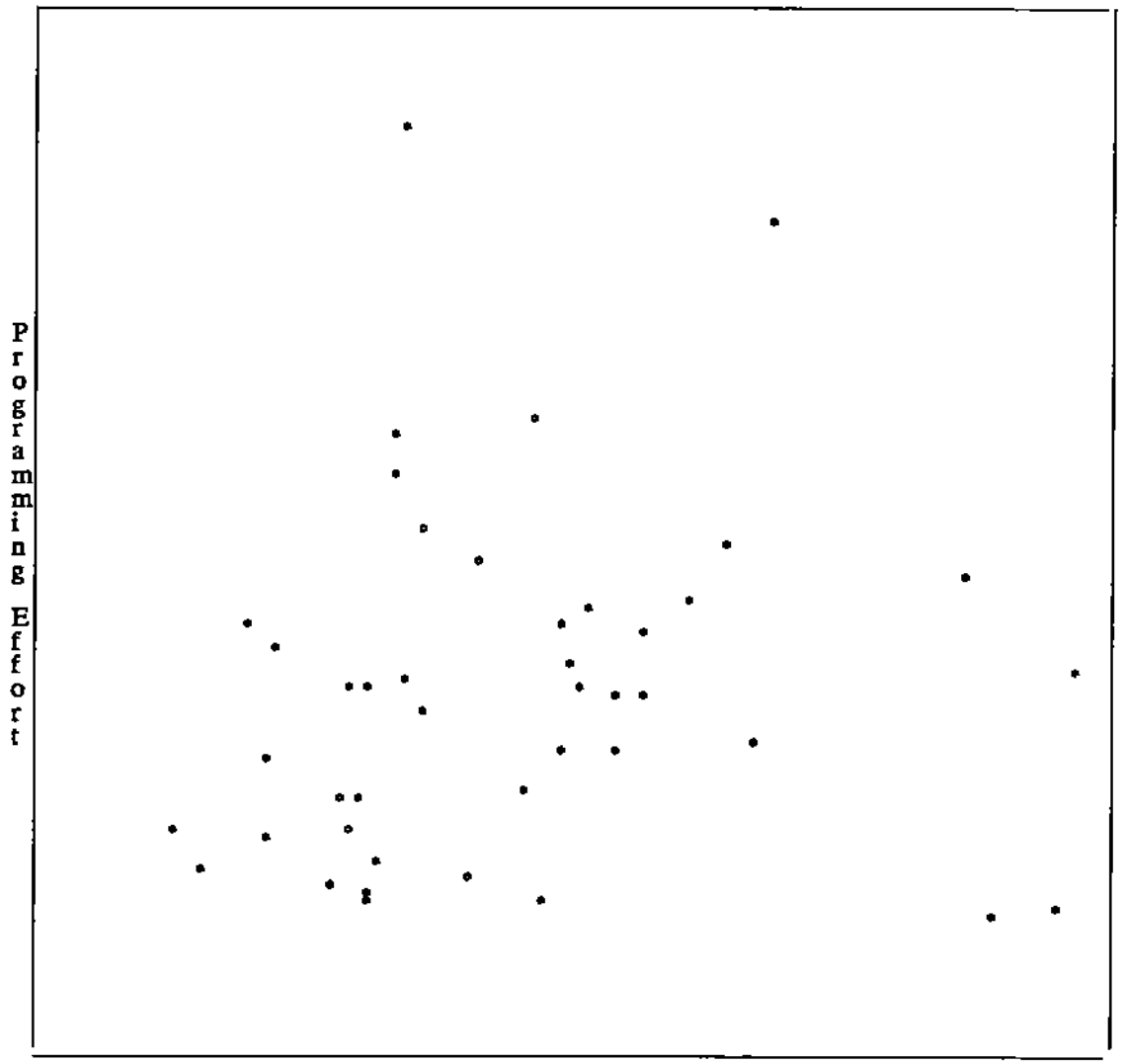

Figure 3. Programming Effort vs. Extended Cloze Procedure

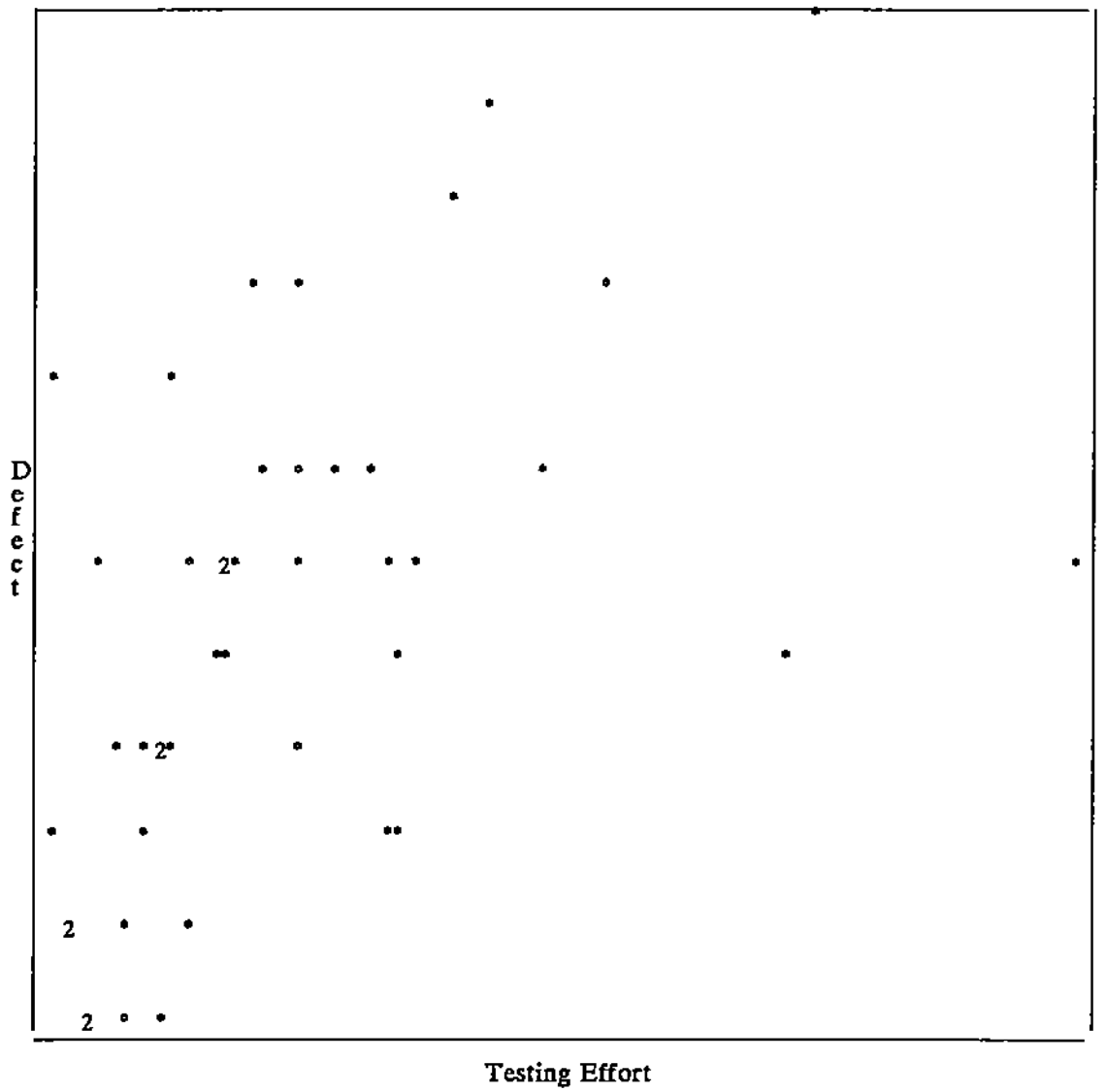

Figure 4. Corrected Defect vs. Testing Effort

RANGE OF X AXIS: 267 784
RANGE OF Y AXIS: 8.16 33.91

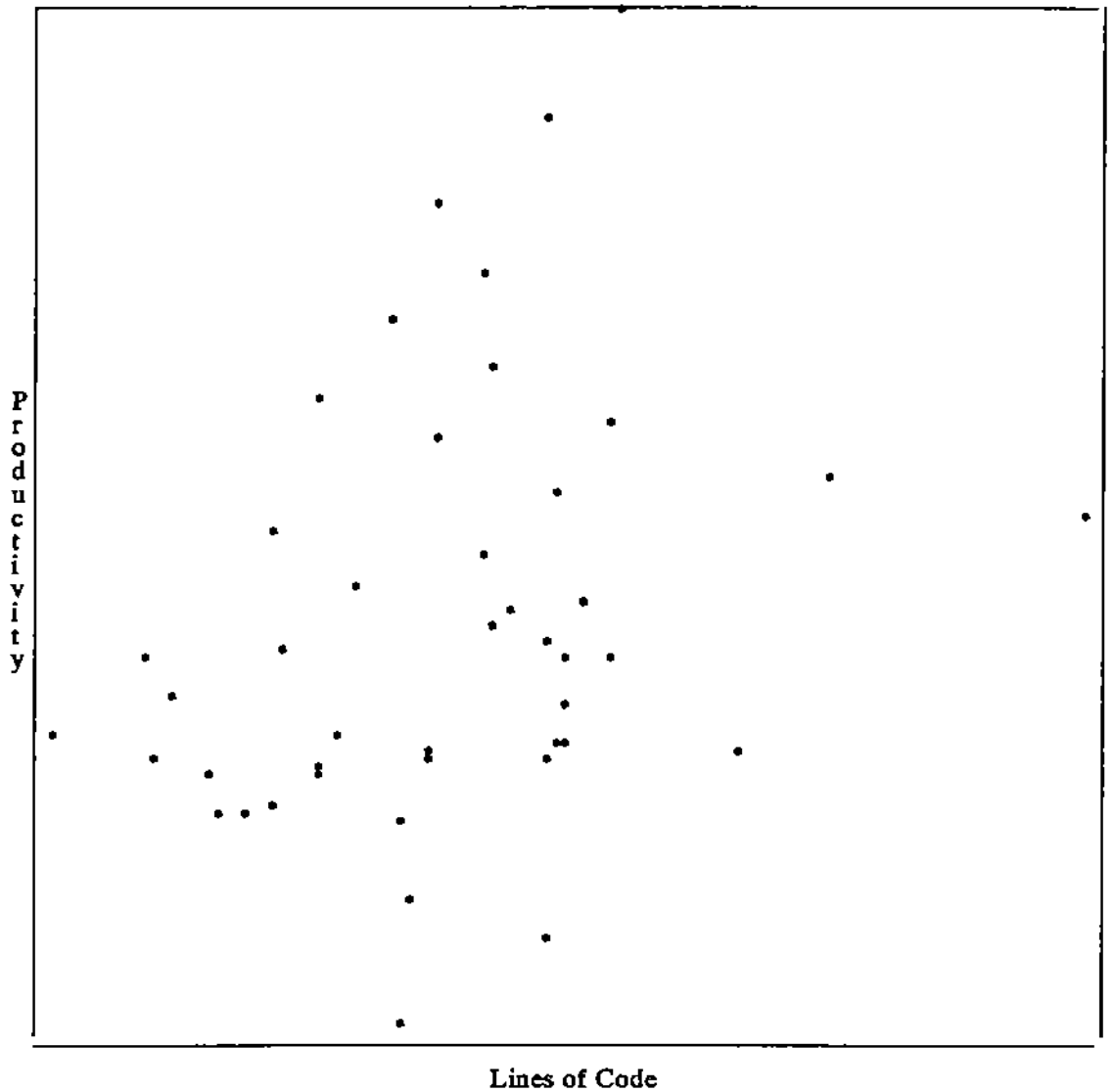

Figure 5. Productivity (*loc*/hr) vs. Lines of Codes

RANGE OF X AXIS: 267 784
RANGE OF Y AXIS: .18 5.36

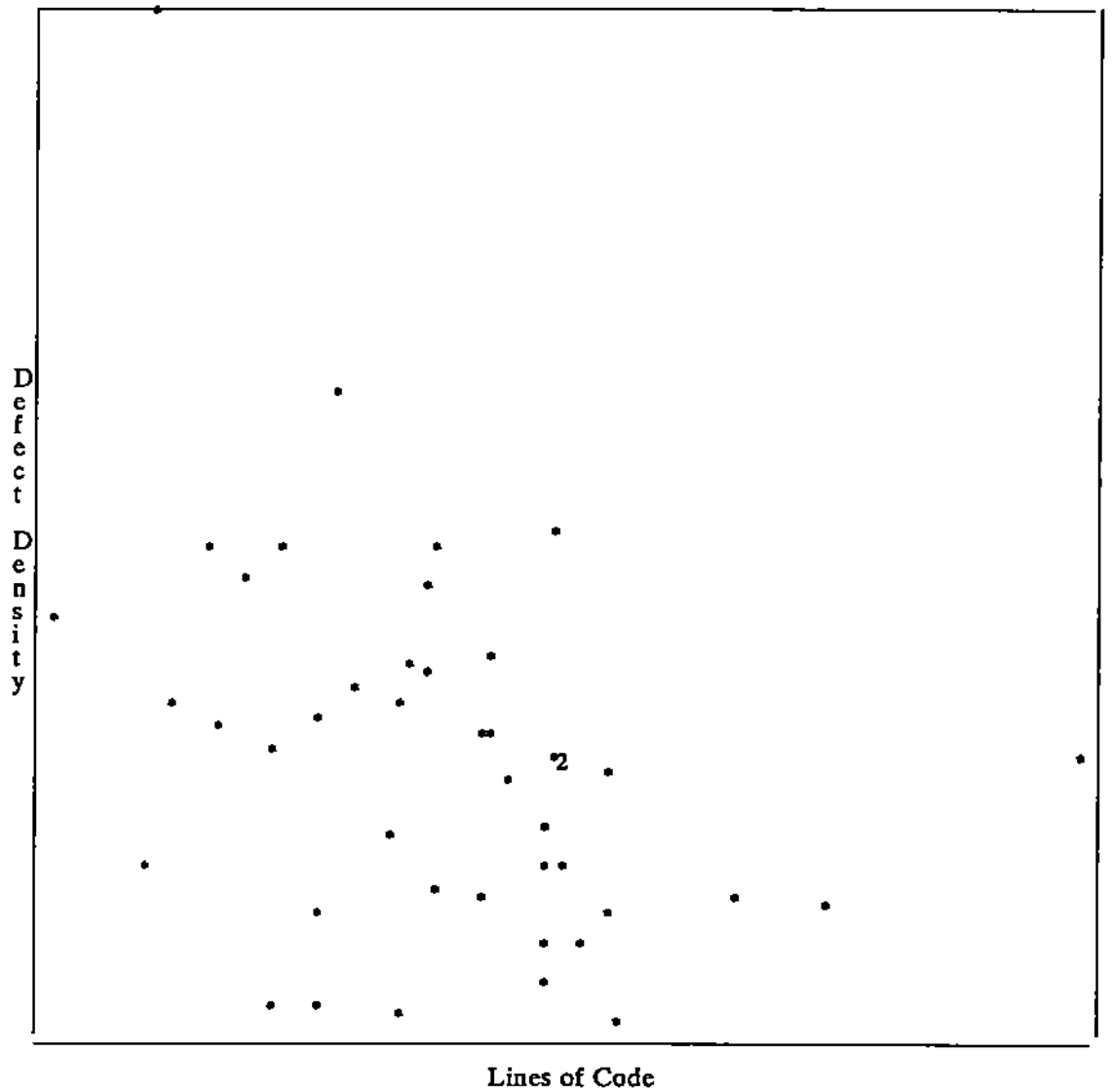

Figure 6. Defect Density (D_1/ 100 *loc*) vs. Lines of Code

Appendix 1: α value of corresponding statistic

degree of freedom = n-2 = 40					
r or s	.202	.257	.304	.357	.393
Student-t	1.303	1.684	2.021	2.423	2.704
α	.2	.1	.05	.02	.01

α value of correlation coefficient (two tailed test)

If n > 30, Pearson correlation coefficient and Spearman rank correlation coefficient have the same significance level.

degree of freedom : $n_1 = 4$, $n_2 = 40$		
F	2.61	3.83
α	.05	.01

α value of *F* Statistic

Appendix 2: A sample of the cloze procedure

```
(*******************************************)
(*       cloze format is {________}        *)
(*******************************************)

procedure readlist (var current,parent:lptr; var flag:boolean);

var
     c:char;    (* used to read in all characters of a list *)
     i:integer; (* index used to read in the atoms *)

begin (* readlist *)
     repeat
          readecho({________});
     until c in [_a_.._z_,left,right,_0_.._9_];
     if c in [_a_.._z_,_0_.._9_] then begin
               (* read in the atom *)
               {________}:=false;
               new(current);
               ans(current,{________});
               i:=1;
               current^.car[i]:={________};
               readecho(c);
               while c in [_a_.._z_,_0_.._9_] do begin
                    i:={________} + 1;
                    current^.car[i]:=c;
                    {________}(c)
                    end; (* while *)
               if c = right then
                    current^.{________}:=true
               else
                    readlist(current^.ptr2,current,{________})
               end (* character *)
     else if c = left then begin
               (* open a new sublist data structure *)
               new(current);
               {________}(current,tenblank);
               flag:=true;
               readlist({________}^.ptr1,current,flag);
               readlist({________}^.ptr2,current,flag)
               end (* left *)
     else begin
          if flag then begin
               {________}^.car:=emptylist;
               flag:=false
               end
          else
               parent^.{________}:=true
          end (* right *)
end; (* readlist *)
```