1978

Measuring Improvements in Program Clarity

Ronald D. Gordon

Report Number:
78-265

MEASURING IMPROVEMENTS IN PROGRAM CLARITY

Ronald D. Gordon

Purdue University
Department of Computer Sciences
West Lafayette, Indiana 47907

CSD-TR 265

May 1978

Measuring Improvements in Program Clarity

by Ronald D. Gordon

Purdue University
Department of Computer Sciences
West Lafayette, Indiana 47906

April 1978

ABSTRACT

The sharply rising cost incurred during the production of quality software has brought with it the need for the development of new techniques of software measurement. In particular, the ability to objectively assess the clarity of a program is essential in order to rationally develop useful engineering guidelines for efficient software production and language development.

A functional relation between the clarity of a program and the number and frequency of operators and operands which occur in the program is presented. This measure of program clarity provides an estimate of the amount of mental effort required to understand the program, assuming that the reader is fluent in the programming language employed.

This measure is tested by applying it to several published examples which demonstrate improvements in program clarity. The objective assessment which is provided using this measure is found to agree with the experimental data gathered.

Keywords and Phrases: program clarity, software measurement, software complexity, cognitive psychology, software science

CR categories: 4.0, 4.6

Introduction

An important problem in quality software research is the measurement of programming style. Many different techniques have been presented which attempt to provide a quantitative assessment of the various aspects which contribute to the quality of software products. At the present, many experts are attempting to prepare guidelines which, if followed, will have the effect of improving programming style. Such an improvement in style will decrease the amount of work required to prepare a program and the amount of effort expended in understanding and maintaining the resulting product. The essential issue then, is to minimize such mental effort. In other words, to accurately assess programming style we must be able to accurately assess the amount of mental effort expended in preparing and understanding the code.

In this paper a measure of program clarity is presented which is a function of the number and frequency of the operators and operands occurring in the program. The resulting value represents the amount of mental work which must be performed in order to comprehend the function of the code.

Several factors influence how easy or difficult it is to understand a particular program. The factors may be categorized into three broad areas: programmer ability, program form, and program structure. Several researchers have studied many of these factors in detail.

The level of fluency of the programmer with the programming language employed greatly influences the difficulty experienced during program comprehension. Shneiderman[66] performed several experiments with groups of both inexperienced and experienced programmers. His empirical results show that there are marked differences between those groups with respect to certain coding practices. For example, novice programmers have more difficulty with modular programs, while experienced programmers find the straight line code more difficult to understand. Long sequences of IF-THEN-statements were easier to follow for the inexperienced programmers than was the corresponding nested IF-THEN-ELSE structure. The reverse was true for advanced programmers.

The programmer's familiarity with the problem domain can also strongly influence the ease with which a program is understood. Such familiarity might enable a programmer to recognize blocks of code very quickly as if a template matching operation had been mentally employed. Shneiderman[67] suggests that background information be collected about the types of programs which a programmer has worked with and the resulting information used as a covariate in a statistical analysis of various performance measures.

Many researchers have studied the influence of program form on the amount of effort required for program comprehension. Weissman[77,78] conducted several empirical studies focusing on such popular issues as commenting, the placement of declarations, indentation, and the use of

mnemonic variable names. His major contribution was the development of a suitable experimental methodology designed to enable researchers to gather suitable empirical evidence which could be used to validate the effects of program style on the relative difficulty of comprehension experienced by programmers. Several experimental procedures are presented in his reports which attempt to measure the degree of comprehension achieved by a programmer after studying a given program. Hand-simulation tasks, various types of quizzes, and methods of self-evaluation were employed. useful results.

Some experiments have been performed which attempt to assess the impact of program structure on program comprehensibility. Program structure includes several factors related to the syntactic representation of an algorithm in a programming language. For example, the number of executable statements, the complexity of the control flow graph of the program, the depth of statement nesting, clustering of data references, and the locality of operations are all factors which influence program structure and affect the clarity of the program.

Previous work which studied the affects of program structure include that of Gannon and Horning[20]. Ten programming language differences between TOPS-2 and TOPS-10 were studied. In order to assess the desirability of various syntactic elements, the persistence of program bugs during program development was measured. The syntactic form of conditional IF-statements was studied by Sime, Green, and Guest[71]. They found that nested IF-THEN-ELSE structures were easier to comprehend than the corresponding IF-GO TO forms for naive programmers. The relationship between the complexity of control flow and program understanding was investigated by Love[53]. That study found that programs with simple control flow were easier for experienced programmers to understand. No statistically significant difference in comprehensibility was found for inexperienced programmers, however, as the complexity of the control flow was varied.

Gilleadi[22] proposed a measure of program structure in an attempt to measure how well-structured a given flowchart was with respect to the precepts of structured programming. Under such a measure, the flowchart was mapped to a representative finite state, sequential automaton. The state transitions of the automaton correspond to the control flow paths represented by the flowchart. A measure of software work was applied to the automaton to estimate the amount of work represented by the flowchart. Ideally, well-structured flowcharts would have small measured values since their representation of the task to be performed was short and elegant. Unfortunately, this measure did not agree well with the accepted notions of good structure in many instances.

The purpose of this study is to develop and empirically verify a measure of program clarity which is a simple function of the program's structure. Such a measure does not reflect the influence of programmer fluency or familiarity with the problem area. In this

study, it is assumed that the programmer is fluent in the language employed and not so familiar with the problem area that recognition is accomplished instantaneously, after only a cursory inspection of the code. Further, factors which affect the program form are considered to be of marginal significance. Comparable methods of indentation, paragraphing, and variable nomenclature are assumed to have been employed when comparing two programs. When these assumptions are not valid, the results which are obtained using the measure formulated may not accurately approximate the observed difficulty in understanding the program.

The measure presented provides an estimate of the mental effort required for comprehension. Such an absolute measure is useful since it allows us to compare two programs, perhaps solving different tasks, and determine not only which is easier to understand, but also how much of a difference is involved.

Measurable Properties of Algorithms

Several program features have traditionally been associated with program clarity. Such features include the number of statements[53,76] and the density of GO TOs[14,59]. However, very few substantive hypotheses relating such features to program clarity have been presented. On the other hand, the methodology of experimental science has shown remarkable success in providing useful theories and workable laws governing many properties of algorithms[27,28]. A consise review of many of the results developed in this field has been prepared by Fitzsimmons and Love[17B].

The relationships developed make use of the following measurable properties of a program:

η_1 - The number of distinct operators
η_2 - The number of distinct operands
N_1 - The total number of operators
N_2 - The total number of operands

These parameters may be easily obtained for programs written in any programming language. The theories developed relate these values to such properties as program length, implementation level, volume, etc. Software science has provided useful relations which enable the potential number of software errors to be calculated, and allow the amount of effort or programming time

Several fundamental properties are defined for an implementation of an algorithm based on the four basic parameters specified above. These properties include the program vocabulary, η, and the program length, N:

$$\eta = \eta_1 + \eta_2$$
$$N = N_1 + N_2$$

The program volume, V, is defined in terms of these measures, as the number of total usages of operators and operands in the implementation, times the number of bits which would be required to provide a unique designator for each of the η different items composing the program vocabulary. The program volume then has the units of bits.

$$V = N \log_2 \eta$$

The program length, N, may be closely approximated as a function of the number of unique operators and operands used in the implementation of the algorithm[27]. The resulting expression has been subjected to several experimental tests, and the results indicate that for well-written programs, the approximation is very accurate[28,31].

$$\hat{N} = \eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2$$

Program level is a measure of the succinctness of an implementation of an algorithm. The highest level at which an algorithm may be represented is in the form of a procedure call. The level of such a representation is taken to be one. More lengthy representations involving many operators and the repetitious use of operands have a lower level[26]. The following estimator for implementation level is used:

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

It should be noted, however, that program level may affect the ease or difficulty of understanding in two contrasting ways. For a person who understands all of the terms involved, a concept may be grasped more quickly or easily the higher the level at which it is presented. On the other hand, in order to convey a given concept to a person less familiar with, or less fluent in, a specific area requires a greater volume, and a lower level. As the saying goes, "A Word to the Wise is Sufficient," and for a person fluent in a language, the difficulty of comprehension is expected to vary inversely with the level.

The Difficulty of Program Construction

The relationships of software science have been employed to estimate the difficulty of program construction. Hypothesizing that the difficulty of programming increases as the volume of the program increases and decreases as the program level increases, Halstead[30] suggested the ratio $E = V/L$ as a measure of the mental effort required to create a program. Program impurities, or flaws in program structure, lower the level of the implementation and/or increase the program length and yet do not represent an increase in the amount of programming effort expended. In order to minimize the influence of such impurities on the measure of programming effort, the estimator of program length, $\hat{N}$, is substituted for N in the calculation of program volume. The resulting measure for program effort is:

$$E\pi = \frac{\hat{N} \log_2 \eta}{\hat{L}}$$

Several studies have been performed to test the usefullness of this measure for programming effort. In one experiment, Zislis[84] selected a dozen algorithms from the Communications of the ACM and prepared rough specifications for each. This set of program specifications was then used to implement a suitable program. Three implementation languages were employed — PL/I, Fortran, and APL. For each of the 36 programs thus obtained, the software science measures were obtained and the measure $E\pi$ calculated.

An estimate of programming time was obtained by dividing $E\pi$ by the number of elementary mental discriminations performed per second of human thought. This constant of proportionality is refered to as the Stroud number and is considered to be roughly 18 discriminations per second for concentrating programmers. The correlation coeficients between the actual and predicted programming times for the programs in this experiment were:

Fortran	0.87
PL/I	0.94
APL	0.93

The predicted time for the entire experiment was 22.51 hours. This estimate compares well with the observed time of 20.15 hours.

In another experiment performed by Gordon and Halstead[23], eleven problems were selected from two published sources. Each problem statement was utilized to prepare a correctly running program which was verified by executing a complex test case for which a correct answer was known. The entire time required to prepare the solution was measured. The software science measures were then applied to predict the programming time for each of the resulting programs.

The total predicted programming time was 374 minutes, which was within 3% of the actual total time required, 385 minutes. The correlation coefficient between the estimated and actual times was

0.934. In contrast, when the number of program statements was correlated with the observed programming times, a coefficient of 0.887 was obtained. Although it is a common practice to attempt to estimate programming time from such a simple measure, this experiment demonstrated that statement counts can predict programming times less well than software science.

The scope of the hypothesis was enlarged as published data became available for large programs. The work reported by Akiyama[1], in an independent study of programming effort and software system debugging, may be used to obtain an estimate of the total effort involved during the development of a large programming project. A value of 84 labor months of effort is obtained when such an analysis is carried out[18]. This figure compares reasonably well with Akiyama's estimate of about 100 labor months.

Measuring Improvements in Clarity

The results presented in the previous section indicate that it is indeed possible to obtain a good approximation of the amount of mental work performed during program construction as a function of easily measured features of the program itself.

Not all of the effort expended during program construction, however, can be estimated based on the resulting code. This is a consequence of the expenditure of "wasted" effort. As a program is produced, a false start may have been made, and portions of the code struck out once a better approach was recognized. While methods of top-down or structured programming attempt to minimize this effect, it is not uncommon for the programmer to consider several software mechanisms or data structures and their effects on subsequent processing before settling upon one which is regarded best suited for the task at hand[12]. The effort expended on such endeavors is not available for measure, since the code is simply not present in the final version of the program. This type of behavior is not reflected in the measure $E\pi$. The term $N \log_2 \eta$ assumes that the programmer prepares the code with perfect insight initially, whereas in actual practice, additional operators and operands are considered and rejected.

For a moment, let us consider how such code revisions affect the program which results. When a statement is first drafted, the translation from thought to concrete form is imprecise, yielding an expression which is perhaps unclear, or at worst, incorrect. As one might polish a composition, so the programmer refines what he has written, in order to present a more perfect statement of the solution. As a result, the program is clearer. It is easier to understand because it is a more coherent representation of the mental image developed by the programmer.

Subsequently, when the code is studied, a reader would need to expend less than the original amount of effort in order to comprehend the ideas of the first programmer. This would be expected, since the refined program most faithfully portrays those concepts upon which its operation is based.

Thus, what we might actually be able to measure in this manner is not the total amount of effort expended during program construction, but the amount of mental effort required to subsequently comprehend the program. We can measure programming effort insofar as the amount of program revision is small. When the program has undergone many changes, the resulting code fails to reflect the additional expenditure of effort.

While it is true that the measure presented was developed based on a model of program generation, it is not unreasonable to expect it to yield an accurate estimate for the amount of mental work expended during program assimilation. The process which is performed while working to understand a program is in many respects, similar to the

process of program generation. In order to comprehend the program, the reader might retrace the thought process which was followed during program generation. A clear program provides several signposts, enabling the reader to follow the development of the program in reverse order as it were, from solution to the initial functional specification of the module.

We may consider the relationship which exists between programming and comprehension effort from another viewpoint. Once a programmer understands an algorithm, the additional effort required to express it as a program in a language for which fluency has been achieved is relatively small. In such a case, the total programming effort is approximately equal to the effort expended for comprehension.

These considerations provide the motivation for additional experiments in order to assess the validity of a measure such as E as an indicator of the amount of effort required to understand a given program. Already, the evidence provided by the experiments conducted support the use of such a measure. Insofar as programming and comprehension efforts are closely related, the success of those experiments indicates that program clarity may be accurately approximated as a simple function of the number and frequency of operators and operands occuring in the program.

In formulating an expression for the amount of mental effort required for program comprehension, the effect of impurities which are present in the implementation should be taken into account. While it is true that the presence of impurities does not represent an increase in programming effort expended, they are expected to have an adverse effect on the comprehensibility of the code. As a result, no attempt should be made to minimize their effect. Consequently, the true program length, N, rather than the estimator of program length, $\hat{N}$, is used in approximating the effort of understanding. The following elegant estimator for the amount of mental effort required to understand a program then results:

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

Experimental Verification of the Hypothesis

Recently, several articles and texts have been published which provide numerous examples of good programming style contrasted with implementations lacking proper organization or structure, and clarity. These examples provide an excellent source of empirical data which may be utilized to test the proposed measure of program understandability. Since the use of good programming style and proper structuring reduce the amount of effort involved in understanding the program (the programs which result are "clearer") one needs only to calculate the corresponding values of E_c for each good and bad version and verify that the poorly written code has a higher measured value of E_c than its well-written counterpart.

This experimental procedure proceeds on the premise that the examples presented in the literature are correctly ranked according to their clarity. Although the authors have rated the code presented in an obviously subjective manner, it is reasonable to expect that in nearly all cases, their analysis of the select few programs chosen for publication is correct. That is, in a large majority of cases, programs that are well written and easy to understand will be contrasted with less clear, poorly written code. Since the authors have carefully fashioned the examples used in their publications, ostensibly to make their point very clear, such a conclusion is well founded. Conscientious authors will develop examples which provide a high contrast between good and bad style in order to emphasize and demonstrate the various issues being developed.

With this view, a large sample of contrasting programs was gathered. This sample could then be used to test various methods of measuring program clarity. In order to be considered accurate, such a measure would have to provide an ordering, consistent with the opinions expressed in the literature.

A few published examples, however, have not been generally accepted as good examples of improvements in programming style. Instead, some authors have argued that the "better" code which was presented demonstrated only a minimal improvement in clarity, and a few reasoned that the code might actually be considered poorer! Such objections, for example, have been raised by Ledgard[50] over the tree searching and insertion example presented by Knuth[44]. In such a situation, the code in question was not included in the sample of programs used to test the performance of the hypothesis.

Table 1 lists the sources for the programs collected. The sample includes the work of several different authors. The tasks addressed by these programs cover a wide spectrum of topics and were written in several different languages including COBOL, PL/I, and FORTRAN.

In order to be included in the sample, both the poorly written and the contrasting well-written code had to perform the same task, or very nearly so. If, in addition to restructuring the program in order to demonstrate an improvement in clarity, the author also chose to

include additional code, say to perform parameter validation or error recovery, the example was not included unless the author contended that this improved program, including the supplementary code, still required less total effort to comprehend. The use of a different algorithm, or a different type of data structure was allowed. Many of the examples were included which demonstrate improvements in clarity as a result of such changes.

Another obvious requirement which each program had to meet was simply that the type of improvement cited had to be an improvement in clarity, rather than improved execution speed, decreased memory requirements, higher accuracy, etc. Further, examples which demonstrated improvements in clarity due to the use of more descriptive variable names or techniques such as "prettyprinting"[36,49], were not included.

Table 1: Program References

Number	Text	Page	Figure	Language
1A	[22]	36	5a	ALGOL
1B	[22]	36	5b	ALGOL
2A	[22]	36	5c	ALGOL
2B	[22]	36	5b	ALGOL
3A	[40]	305	1a	FORTRAN
3B	[40]	306	1c	FORTRAN
4A	[40]	305	1a	FORTRAN
4B	[40]	306	1b	FORTRAN
5A	[40]	306	1c	FORTRAN
5B	[40]	306	1b	FORTRAN
6A	[40]	306	2a	FORTRAN
6B	[40]	306	2b	FORTRAN
7A	[40]	306	2a	FORTRAN
7B	[40]	307	2c	FORTRAN
8A	[40]	306	2b	FORTRAN
8B	[40]	307	2c	FORTRAN
9A	[40]	307	3a	FORTRAN
9B	[40]	307	3b	FORTRAN
10A	[40]	308	4a	FORTRAN
10B	[40]	309	4b	FORTRAN
11A	[40]	311	6a	FORTRAN
11B	[40]	311	6b	FORTRAN
12A	[40]	311	7a	FORTRAN
12B	[40]	312	7b	FORTRAN
13A	[40]	312	8a	PL/I
13B	[40]	312	8b	PL/I
14A	[40]	313	9a	PL/I
14B	[40]	313	9b	PL/I
15A	[40]	314	10a	PL/I
15B	[40]	314	10b	PL/I
16A	[40]	315	11a	FORTRAN
16B	[40]	316	11b	FORTRAN
17A	[40]	316	12a	FORTRAN
17B	[40]	317	12b	FORTRAN
18A	[40]	317	13a	PL/I
18B	[40]	318	13b	PL/I
19A	[9]	21	2.5a	COBOL
19B	[9]	21	2.5b	COBOL
20A	[9]	24	2.2.1a	COBOL
20B	[9]	24	2.2.1b	COBOL
21A	[9]	24	2.2.2a	COBOL
21B	[9]	24	2.2.2b	COBOL
22A	[9]	24	2.2.3a	COBOL
22B	[9]	24	2.2.3b	COBOL
23A	[9]	25	2.2.4a	COBOL
23B	[9]	25	2.2.4b	COBOL

Table 1, continued

Number	Text	Page	Figure	Language
24A	[9]	25	2.2.4a	COBOL
24B	[9]	25	2.2.4c	COBOL
25A	[9]	25	2.2.4b	COBOL
25B	[9]	25	2.2.4c	COBOL
26A	[9]	41	2.13a	COBOL
26B	[9]	42	2.13b	COBOL
27A	[80]	253	3.17	PASCAL
27B	[80]	253	3.16	PASCAL
28A	[80]	254	3.20	PASCAL
28B	[80]	254	3.22	PASCAL
29A	[80]	256	5.34	PASCAL
29B	[80]	257	5.37	PASCAL
30A	[44]	266	1a	ALGOL
30B	[44]	266	1	ALGOL
31A	[44]	270	3	ALGOL
31B	[44]	270	3.1a	ALGOL
32A	[44]	271	4a	ALGOL
32B	[44]	271	4	ALGOL
33A	[39]	3	1a	FORTRAN
33B	[39]	3	1b	FORTRAN
34A	[39]	11	2a	FORTRAN
34B	[39]	11	2b	FORTRAN
35A	[39]	11	2a	FORTRAN
35B	[39]	12	2c	FORTRAN
36A	[39]	11	2b	FORTRAN
36B	[39]	12	2c	FORTRAN
37A	[39]	12	3a	FORTRAN
37B	[39]	12	3b	FORTRAN
38A	[39]	15	5a	FORTRAN
38B	[39]	16	5b	FORTRAN
39A	[39]	20	7a	FORTRAN
39B	[39]	20	7b	FORTRAN
40A	[39]	21	8a	FORTRAN
40B	[39]	21	8b	FORTRAN
41A	[39]	21	8a	FORTRAN
41B	[39]	22	8c	FORTRAN
42A	[39]	21	8b	FORTRAN
42B	[39]	22	8c	FORTRAN
43A	[39]	22	9a	FORTRAN
43B	[39]	22	9b	FORTRAN
44A	[39]	23	10a	FORTRAN
44B	[39]	24	10b	FORTRAN
45A	[39]	31	11a	FORTRAN
45B	[39]	33	11b	FORTRAN
46A	[39]	39	12a	FORTRAN
46B	[39]	40	12b	FORTRAN

After the programs had been collected, a careful count of the software parameters η_1, η_2, N_1, and N_2 was obtained. For each program it was then possible to calculate the expected amount of mental effort required for comprehension, E_c. The number of executable statements, N_s, was also obtained and this value is listed in Table 2 along with the other parameters. In this manner, a comparison could be made between E_c and the traditionally accepted hypothesis that program understandability is proportional to the number of program statements.

The examples are paired in Table 2, listing the poorer version of the code first, as indicated by the analysis and comment of the original author. A comparison can then be made in order to ascertain how well the measures E_c and N_s agree with the observed ranking presented in the literature. In a few instances, code was presented representing "good," "better," and "best" implementations. Such a situation enables us to make 3 comparisons. For this reason, some programs may appear in more than one comparison within Table 2.

In analyzing the empirical data obtained, credit is given to a proposed measure of program clarity when it indicates that the amount of effort expended for program comprehension had decreased by a significant amount. Unfortunately, just what percentage of improvement should be required is not known precisely. Workers in this area have simply not been able to quantify improvements in programming style. As a result, when an improvement is noted, the amount of improvement is not reported. In the analysis presented here, at least a 10% improvement had to be reflected by a measure in order that the measure be considered in agreement with the literature. Since these examples were developed in order to unquestionably demonstrate an improvement in style, a significant reduction in the amount of mental effort required for comprehension would normally be expected. Indeed, most improvements in style were reflected by a much more dramatic reduction in E_c.

On the other hand, several examples undergo only a localized enhancement which affects only a few of the statements of the entire program. Since the measures studied in this research attempt to assess the total effort required for comprehension, such a change will not result in a very large decrease, overall. A threshold value of about 10% appears to be low enough to account for this latter effect, while high enough to filter out those cases which show only a marginal amount of improvement. The assumption here is that a measure which shows less than a 10% reduction in effort is not in agreement with what the author contends is a clear improvement.

Table 2: Empirical Data Gathered

Number	η_1	η_2	N_1	N_2	V	D	E_c	N_s
1A	13	10	35	30	294	19.50	5734	12
1B	11	7	26	22	200↓	17.29↓	3460↓	8↓
2A	13	7	30	25	238	23.21	5518	8
2B	11	7	26	22	200↓	17.29↓	3460↓	8≈
3A	7	5	17	13	108	9.10	979	3
3B	5	7	16	13	104≈	4.64↓	483↓	4↑
4A	7	5	17	13	108	9.10	979	3
4B	8	7	22	18	156↑	10.29↑	1607↑	6↑
5A	5	7	16	13	104	4.64	483	4
5B	8	7	22	18	156↑	10.29↑	1607↑	6↑
6A	8	4	26	14	143	14.00	2008	13
6B	4	4	10	10	60↓	5.00↓	300↓	5↓
7A	8	4	26	14	143	14.00	2008	13
7B	4	4	5	4	27↓	2.00↓	54↓	1↓
8A	4	4	10	10	60	5.00	300	5
8B	4	4	5	4	27↓	2.00↓	54↓	1↓
9A	8	9	28	21	200	9.33	1869	8
9B	8	8	19	15	136↓	7.50↓	1020↓	6↓
10A	12	15	56	40	456	16.00	7304	20
10B	10	8	24	15	163↓	9.38↓	1525↓	7↓
11A	13	8	38	27	286	21.94	6263	8
11B	10	6	33	25	232↓	20.83≈	4833↓	6↓
12A	7	3	10	6	53	7.00	372	4
12B	5	3	6	6	36↓	5.00↓	180↓	2↓

Table 2, continued

Number	η_1	η_2	N_1	N_2	V	D	E_c	N_s
13A	10	4	17	8	95	10.00	952	8
13B	7	3	12	8	66↓	9.33≈	620↓	7↓
14A	8	8	21	14	140	7.00	980	6
14B	9	8	19	14	135≈	7.88↑	1062≈	6≈
15A	6	4	20	14	113	10.50	1186	7
15B	5	4	12	10	70↓	6.25↓	436↓	5↓
16A	14	11	36	25	283	15.91	4507	7
16B	15	9	29	19	220↓	15.83≈	3485↓	7≈
17A	9	6	19	12	121	9.00	1090	4
17B	7	5	13	12	90↓	8.40≈	753↓	4≈
18A	26	34	189	131	1890	50.09	94677	73
18B	22	19	83	61	771↓	35.32↓	27246↓	11↓
19A	7	5	9	5	50	3.50	176	3
19B	8	4	13	6	68↑	6.00↑	409↑	3≈
20A	8	2	11	2	43	4.00	173	5
20B	6	2	7	2	27↓	3.00↓	81↓	4↓
21A	9	4	13	5	67	5.63	375	6
21B	9	4	11	4	56↓	4.50↓	250↓	6≈
22A	5	2	6	2	22	2.50	56	3
22B	3	1	3	1	8↓	1.50↓	12↓	2↓
23A	9	9	20	15	146	7.50	1095	9
23B	11	9	17	12	125↓	7.33≈	919↓	5↓
24A	9	9	20	15	146	7.50	1095	9
24B	8	10	10	13	96↓	5.20↓	499↓	5↓

Table 2, continued

Number	η_1	η_2	N_1	N_2	V	D	E_c	N_s
25A	11	9	17	12	125	7.33	919	5
25B	8	10	10	13	96↓	5.20↓	499↓	5≈
26A	15	14	47	41	428	21.96	9390	14
26B	8	6	12	9	80↓	6.00↓	480↓	4↓
27A	15	8	27	21	217	19.69	4275	9
27B	13	6	20	13	140↓	14.08↓	1974↓	6↓
28A	13	6	25	21	195	22.75	4445	9
28B	13	6	21	17	161↓	18.42↓	2973↓	7↓
29A	13	8	25	21	202	17.06	3447	9
29B	11	8	25	21	195≈	14.44↓	2821↓	9≈
30A	13	7	28	25	229	23.21	5318	7
30B	13	7	28	23	220≈	21.36≈	4708↓	8↑
31A	15	7	35	26	272	27.86	7578	10
31B	15	7	35	29	285≈	31.07↑	8868↑	9≈
32A	11	7	38	15	221	11.79	2605	12
32B	12	4	31	9	160↓	13.50↑	2160↓	10↓
33A	7	2	8	2	32	3.50	111	3
33B	5	2	5	2	20↓	2.50↓	49↓	2↓
34A	8	4	19	10	104	10.00	1040	7
34B	5	4	12	10	70↓	6.25↓	436↓	5↓
35A	8	4	19	10	104	10.00	1040	7
35B	4	4	5	4	27↑	2.00↓	54↓	1↓
36A	5	6	12	12	83	5.00	415	5
36B	4	4	5	4	27↑	2.00↓	54↓	1↓

Table 2, continued

Number	η_1	η_2	N_1	N_2	V	D	E_c	N_s
37A	5	6	12	12	83	5.00	415	3
37B	6	5	10	8	62↓	4.80≈	299↓	1↓
38A	13	21	58	45	524	13.93	7299	11
38B	14	19	41	33	373↓	12.16↓	4538↓	9↓
39A	9	6	17	12	113	9.00	1020	6
39B	7	6	13	12	93↓	7.00↓	648↓	4↓
40A	10	5	22	9	121	9.00	1090	13
40B	9	5	17	11	107↓	9.90↑	1055≈	7↓
41A	10	5	22	9	121	9.00	1090	12
41B	8	5	14	7	78↓	5.60↓	435↓	7↓
42A	9	5	17	11	107	9.90	1055	7
42B	8	5	14	7	78↓	5.60↓	435↓	7≈
43A	8	6	12	9	80	6.00	480	4
43B	7	6	9	9	67↓	5.25↓	350↓	1↓
44A	10	10	19	14	143	7.00	998	11
44B	9	10	17	14	132≈	6.30≈	830↓	10≈
45A	16	23	67	46	597	16.00	9556	26
45B	12	24	32	31	326↓	7.75↓	2524↓	6↓
46A	13	16	75	62	666	25.19	16763	15
46B	11	16	48	29	366↓	9.97↓	3650↓	11↓

The Experimental Results

Several factors affect the overall or net clarity of a program as perceived by an individual. The diverse nature of such elements is conveyed by the large number of various techniques, transformations, and guidelines proposed as aids to good programming style. As observed in the examples, several seemingly unrelated aspects of programming style must be addressed individually before a thorough understanding of program clarity may be achieved.

At least this is certainly the attitude presented in the literature today, and supported through the ongoing stream of publications which treat several seemingly distinct aspects contributing to program clarity by displaying numerous and sundry examples. Lacking an understanding of the underlying fundamental issues and concepts involved, we could hope to do no better than to collect and categorize the many techniques which arise, forming at best a crude patchwork of ad hoc remedies for the unfathomable programs which many programmers prepare.

Ideally, the development of a single theory of program clarity would tie together many of these diverse proposals and provide a unified framework upon which further study may be based. Such a theory would be recognized by its ability to accurately assess the clarity of programs incorporating the many proposed techniques for improving clarity. Further, the simple parameters utilized in such a successful theory would be identified as crucial or atomic elements affecting the mental process involved during comprehension.

The experiment performed is a direct attempt to test one formulation of just such a theory. By selecting a wide class of programs, covering the broadest spectrum of proposed methodologies, the ability of a proposed measure to account for the various aspects of program clarity on a fundamental level may be assessed. Only a theory capable of suitably approximating the essential processes involved during program comprehension can be expected to be able to account for the many unrelated examples presented in the literature.

The proposed hypothesis was subjected to this test. Additionally, an alternate hypothesis relating the number of executable statements to program clarity was examined. The results of the experiment provide significant evidence in support of the use of E_c as a measure of clarity. A total of 46 comparisons was made, ranking 76 versions of various programs. For 40 of these comparisons, the measure E_c properly indicated a significant decrease in the amount of effort required for comprehension. This measure performed better than the alternative measure N_s. The latter agreed with the published reports in only 31 cases.

The results provide a significant indication that program clarity may be assessed by means of a simple formulation of elementary program features. Each example meeting the criteria above, from every article located during the literature search undertaken, was included in the

sample. A few years ago, such a sample would have been very small, and as time passes, the sample will grow. At present, the sample is as large as it can be, or very nearly so. While the absolute number of examples present in the sample is not extremely large, the sample does present a substantial test of the hypothesis because of the great diversity of techniques covered. The results of this experiment demonstrate the ability of the hypothesis to directly address those factors which fundamentally affect program clarity.

Supportive evidence is also provided by an unusual example proposed by Ledgard and Marcotty[50], demonstrating two programs which solve the same problem but which employ an entirely different set of control structures. The point stressed in that article was that both implementations required an equal amount of effort in order to comprehend them. Yet the programs, one 29 and the other 24 statements long, could not be considered of equal complexity on the basis of size. Depth of statement nesting and the types of structures employed differed, making any conclusions drawn from such a basis fallacious. Nonetheless, E_c varied by less than 3% for the two programs, indicating, in agreement with the authors, that the programs required almost an equal amount of mental effort to understand.

A LOOK AT A FEW OF THE EXAMPLES

We now turn our attention to a few of the individual examples, and take a closer look at the improvements made to the code along with the corresponding changes in E_c. The first example is a selected fragment of code which is presented by Kernighan and Plauger[40]. The examples compared, 15A and 15B, are shown in Figures 1 and 2. Here the difficulty encountered when IF-statements are nested in a tree-like fashion is demonstrated. If the code is rewritten so as to present a more linear structure, a much clearer program results. The analysis of the code shows an improvement of roughly 65% over the original version using the software science measures.

```
IF X >= Y
   THEN IF Y >= Z
           THEN SMALL = Z;
           ELSE SMALL = Y;
   ELSE IF X >= Z
           THEN SMALL = Z;
           ELSE SMALL = X;
```

Figure 1: The Code of Example 15A

```
SMALL=X;
IF Y < SMALL
   THEN SMALL = Y;
IF Z < SMALL
   THEN SMALL = Z;
```

Figure 2: The Code of Example 15B

Not only has E_c provided a reasonable indication of clarity here, but it has done so without resorting to an analysis of such macroscopic features such as the depth of statement nesting, or a detailed flow analysis of the program. The success enjoyed by this approach leads us to recognize that while such features are indeed factors affecting program clarity, they are not the most atomic features. Evidently, the process of comprehension may be modeled as a process which deals with the details of a program on a much more microscopic level.

The code presented in Figure 3 represents the simplest solution to the task of Example 15. The solution utilizes a procedure which is assumed available within the language employed. The corresponding increase in clarity is reflected by a decrease in E_c, from 436 elementary mental discriminations (EMD) for the code of Example 15B, to 15 EMD for the procedure call.

```
CALL MINIMUM (SMALL, X, Y, Z);
```

Figure 3: A Minimal Solution for Example 15

The presentation of the solution at such a high level of abstraction requires significantly less effort to understand. It is important to note that such an analysis is appropriate for a programmer capable of fluently recognizing the operator MINIMUM, and its behavior. If, however, the amount of effort expended by a particular person was observed to be above that predicted by the measure, we may be justified in concluding that the assumption of fluency is false, since the theory does not hold. Instead, the theory provides a measure of clarity under an easily specified standard condition of fluency.

In another example, the desirability of the GO TO-statement as a means of controlling the execution sequence of a program is studied. For those languages which lack suitable alternatives, several guidelines have been established so that the GO TO will be used in only specific, easily understood situations. For example, one might allow its use only to implement a simple iteration, or a break arising from an exceptional condition. The code of Example 13A is shown in Figure 4 and although the GO TO-statements are only used in compliance with the rules suggested above, it is apparent that the code is

difficult to comprehend. A value of 952 discriminations is calculated for this code.

```
DCL NEWIN DEC FLOAT(4);
    LARGE DEC FLOAT(4) INIT(.0E1);

NEXT_C: GET LIST (NEWIN);
        IF NEWIN >= 0
           THEN IF NEWIN > LARGE
                   THEN LARGE=NEWIN;
                   ELSE GO TO NEXT_C;
           ELSE GO TO FINISH;
        GO TO NEXT_C;
FINISH: PUT LIST (LARGE);
```

Figure 4: The Code of Example 13A

The code of Example 13B is much clearer. The control flow has been reorganized so that a more linear representation is possible. Most, but not all of the GO TO's have been removed. Here again, the increase in program clarity is reflected by a decrease in E_c. A value of 620 discriminations is obtained by means of the simple analysis technique proposed. This represents a 35% reduction in the amount of effort which must be expended in order to understand the improved version of the program. Such a reduction is reasonable for the improvements cited.

```
DECLARE (NEWIN, LARGE) DECIMAL FLOAT(4);

        LARGE=0;
NEXT_C: GET LIST (NEWIN);
        IF NEWIN > LARGE
           THEN LARGE = NEWIN;
        IF NEWIN >= 0
           THEN GO TO NEXT_C;
        PUT LIST (LARGE);
```

Figure 5: The Code of Example 13B

The Exceptional Cases

There are six examples for which the value E_c does not reflect the anticipated decrease in programming effort. For an experiment of this nature, involving such a diverse and varied collection of programming examples, the overall behavior of the proposed measure to accurately reflect the improvements claimed is significant evidence in support of the theory, even in light of these few anomalous cases. Surprisingly, a careful investigation of these cases provides additional evidence in favor of such a theory. All six examples are reconciled, the initially curious results being due to the following causes:

- The code by itself did not properly demonstrate the proposed improvement suggested by the author.
- The assumption of fluency during the examination of one example was inappropriate.
- Finally, one author's suspicions that the example presented might not demonstrate an improvement in clarity seemed justified.

THE POORLY CONSTRUCTED EXAMPLES

In COBOL, Chmura and Ledgard advocate the use of the CALL verb in order to invoke general purpose subprogram modules[9]. In this way, the resulting program becomes clearer, and if lucky, someone else might have already written the needed module so that a great deal of work may be avoided. The "poor" code of Example 19A is presented in Figure 6. This code utilizes the PERFORM verb in order to link to a paragraph contained within the program. The "better" code, shown in Figure 7, makes use of the CALL verb to link to a module, presumably already available within the installation's library.

```
PRODUCE-ACTION-RPT-HEADER.
   ACCEPT TODAYS-DATE FROM DATE.
   PERFORM CALCULATE-JULIAN-DAY.
   MOVE JULIAN-DAY
      TO EDITED-JULIAN-DAY IN ACTION-RPT-HEADER.
```

Figure 6: The Code of Example 19A

```
PRODUCE-ACTION-RPT-HEADER.
   CALL "CALCULATE-JULIAN-DAY"
      USING JULIAN-DAY.
   CANCEL "CALCULATE-JULIAN-DAY".
   MOVE JULIAN-DAY
      TO EDITED-JULIAN-DAY IN ACTION-RPT-HEADER.
```

Figure 7: The Code of Example 19B

The code presented in Figure 7, however, is not clearer than that of Figure 6 when considered by itself. Indeed, Example 19B requires an additional statement utilizing the CANCEL verb, and a special symbol must be present to set off the external module name from the rest of the text. These requirements unquestionably add to the complexity of Example 19B. As the analysis shows, Example 19A is easier to understand than is Example 19B. Here, the code presented by the authors simply does not tell the whole story.

Clearly, the authors intended for the reader to imagine the code representing the paragraph CALCULATE-JULIAN-DAY, and in assessing the amount of effort required to understand Example 19A, the amount of effort expended to comprehend the non-standard module was to be included. Example 19B, including the additional mechanisms employed, would then be easier to understand since the standard library module is presumably fluently recognized and the details of its construction do not contribute to the difficulty of understanding the program. Here too, the proposed hypothesis has accurately assessed the situation, and has provided a simple measure which may in addition, be used to gauge how large a module must be before the utilization of the CALL verb becomes profitable.

In another example, the excessive use of GO TO-statements is avoided in an attempt to improve a poorly written piece of code. Unfortunately, the better code, presented as Example 40B, contains a very cumbersome and difficult to comprehend IF-statement. As a result, only a small overall improvement is actually achieved. The code for Example 40A, and the modified version, 40B, is shown in Figures 8 and 9, respectively.

```
      LOGICAL FEM(8), MALE(8)
      READ (5,6) IGIRL, FEM
    9 READ (5,6) IBOY, MALE
      DO 8 I=1, 8
      IF (FEM(I)) GO TO 7
      IF (.NOT.MALE(I)) GO TO 8
      GO TO 9
    7 IF (.NOT.MALE(I)) GO TO 9
    8 CONTINUE
      WRITE (2,10) IBOY
      GO TO 9
      STOP
      END
```

Figure 8: The Code of Example 40A

```
      LOGICAL FEM(8), MALE(8)
      READ (5,10) IGIRL, FEM
20    READ (5,10) IBOY, MALE
      DO 30 I=1, 8
      IF ((FEM(I).AND..NOT.MALE(I)) .OR.
     $    (MALE(I).AND..NOT.FEM(I))) GO TO 20
30    CONTINUE
      WRITE (2,40) IBOY
      GO TO 20
      END
```

Figure 9: The Code of Example 40B

Because standard FORTRAN does not allow us to directly test whether or not two LOGICAL variables are equal, we must resort to an indirect approach. In Example 40A, a chain of GO TO's wind through the program in order to evaluate the situation. In fact, such a test may be most suitably performed by an exclusive-or operation, but alas, FORTRAN does not allow for this either. In Example 40B we are forced to express the exclusive-or operator in a clumsy fashion, leaving the reader to ponder what operation is actually performed by the IF-statement. Neither solution is very attractive and the measure E_c shows only a slight improvement in clarity.

The authors make this point and conclude that what is actually needed is either an operator, .XOR., or alternately a change of variable type to INTEGER. Such changes cut to the heart of the matter and allow the test to be made directly. Example 42B utilizes the latter approach. Assuming that the data can be presented using 1 and 0 to represent the values .TRUE. and .FALSE., the operator .EQ. may be utilized to achieve the desired test. The code which results is shown in Figure 10. It is much simpler than the original code of Example 40A, and the measure of program clarity, E_c, indicates a 40% improvement.

```
      INTEGER FEM(8), MALE(8)
      READ (5,10) IGIRL, FEM
20    READ (5,10) IBOY, MALE
      DO 30 I=1, 8
      IF (FEM(I).NE.MALE(I)) GO TO 20
30    CONTINUE
      WRITE (2,40) IBOY
      GO TO 20
      END
```

Figure 10: The Code of Example 42B

Consider next, the code required to implement a billing algorithm from the specifications provided in the table below:

Condition	Action
QTY ≤ 10	set BILL_A to 0.00
10 < QTY ≤ 200	? (do nothing)
200 < QTY < 500	add 0.50 to BILL_A
500 ≤ QTY	add 1.00 to BILL_A

The actions which are specified may be realized in PL/I by employing a nested IF-structure. Such a technique has led to the production of the code presented in Figure 11:

```
IF QTY > 10
   THEN IF QTY > 200
           THEN IF QTY >= 500
                   THEN BILL_A=BILL_A+1.00;
                   ELSE BILL_A=BILL_A+0.50;
           ELSE;
   ELSE BILL_A=0.00;
```

Figure 11: The Code of Example 14A

As one of their simple programming proverbs, Kernighan and Plauger[39] suggest that a null ELSE-clause is a symptom of poorly structured code. Indeed, in this example the nested IF-structure may be reorganized, removing this anomaly, and the authors claim that a clearer program results. This was done, and the resulting code is shown in Figure 12. While an improvement in clarity is claimed, the measure E_c indicates to the contrary, that the code is slightly more difficult to comprehend, although less than a 10% difference is involved.

```
IF QTY >= 500
   THEN BILL_A=BILL_A+1.00;
   ELSE IF QTY > 200
           THEN BILL_A=BILL_A+0.50;
           ELSE IF QTY <= 10
                   THEN BILL_A=0.00;
```

Figure 12: The Code of Example 14B

Consider, however, the following question which might arise after the original programmer and specifications were no longer available: Under what condition is the variable BILL_A left unaltered? This situation may later require that an action be performed. For which version would the inclusion of the additional code be most easily accomplished? Without a doubt, the code of Example 14A is superior to that of Example 14B with respect to these considerations. Evidently, less effort is required in order to formulate a mental image of the program's function and operation given the code of Example 14A.

This reasoning leads one to the conclusion that the authors' claim is not simply justified. In this particular example, the decision table presented is not more simply represented in the CASE-like format employed in Example 14B. Indeed, some "optimization" has been performed in order to prepare a program segment which neatly avoids the unspecified case. This perturbation necessitates the expenditure of additional effort during program construction. During any attempt to understand the resulting code, additional effort must be invested in order to mentally reconstruct the original functional specification.

In this example the authors' choice of problem is considered to be at fault. The point which was to be made was that a CASE statement, missing in PL/I, could be routinely constructed using a nested IF-structure in which the action to be performed appeared as the THEN-clause. Because of the ease with which CASE statements may be formulated and understood in structured languages like Pascal, similar improvements ought to be achievable in PL/I using such a simple convention. The measure E_c does indicate an improvement in clarity for such situations in which the alternative is a nested IF statement implementing a tree-like decision structure.

Even so, the simple scheme which is proposed can never present as clear or as easily understood an image as a CASE statement might for an obvious reason. The IF statement, unlike the CASE operator in Pascal, is a very general verb. When employed, even in this limited role, some effort must be expended in order to recognize the type of operation being synthesized. This additional effort is not required for a programmer who is fluent with the CASE verb and recognizes the operation accomplished directly.

THE REQUIREMENT OF FLUENCY

It is quite understandable that an example which incorporates rarely utilized or poorly understood language features, will require more effort to understand than that predicted by the theory. Actually, the observed effort will vary from individual to individual depending on several factors, including his familiarity with the problem area as well as his understanding of the language constructs employed. The theory provides a normalized measurement, which minimizes the effects of such variations by considering the difficulty of comprehension for a fluent programmer. This somewhat idealized situation is very nearly matched in practice, and as a result, the application of the theory yields an accurate estimate of the difficulty experienced by a wide segment of the programmers most likely to review the code. In general, the results obtained are characteristic of the amount of effort expended under the standard condition of fluency and are quite useful when the effects of a technique or program transformation are to be studied.

An example which demonstrates the necessity for the condition of fluency is that of Examples 4A and 4B. The code is written in

FORTRAN, a small language for which fluency has been achieved by many people. Yet some are very puzzled by the code presented in Figure 13.

```
      DO 1 I=1,N
      DO 1 J=1,N
    1 X(I,J)=(I/J)*(J/I)
```

Figure 13: The Code of Example 4A

The difficulty which arises as one attempts to understand the code of Example 4A stems from the unusual use of the integer division operator. Here, the operator is used not to obtain a quotient, but to achieve truncation. In a sense, it is as if a common element, having a familiar meaning, was used in a context requiring a secondary meaning, one which was not fluently recognized by most programmers. It is true that most good FORTRAN programmers recognize the fact that integer division truncates its result, but when they use the operator, their primary intent is to obtain the quotient of two integers. Through repeated experience with this operator in situations in which it is so used, this meaning is fluently established. Unfortunately, this is not the function of the operator as used in Example 4A.

In contrast to Example 4A, for which a very low value of E_c is obtained, Example 4B presents a much more explicit statement of the operation to be performed. That code yields a much higher value of E_c, while many people will experience less difficulty understanding its operation. It is only when the property of truncation by reference to integer division is fluently recognized, that so small an expenditure of mental effort would be expected in practice. In this case the theory provides an accurate indication of the effort required for a fluent programmer to comprehend the code provided.

```
      DO 20 I=1,N
         DO 10 J=1,N
            IF (I.EQ.J) X(I,J)=1.0
            IF (I.NE.J) X(I,J)=0.0
10       CONTINUE
20    CONTINUE
```

Figure 14: The Code of Example 4B

A QUESTIONABLE EXAMPLE

One example included in the sample was of dubious value to this study. Knuth presents a short segment of code which in his opinion, might be improved by the removal of the GO TO statement which in Example 31A implements a break condition[44]. Already, we have noted that the use of a GO TO in such a situation may be justified, and some

authors have postulated that such usage need not complicate a program unduly. On the other hand, such guidelines may not be applicable in all situations and as was seen in Example 13, a significant improvement may be possible once the code is reorganized to circumvent the need for such a GO TO.

```
                I:=H(X);
                WHILE A[I]≠0 DO
                   BEGIN IF A[I]=X
                            THEN GO TO FOUND FI;
                         I:=I-1;
                         IF I=0
                            THEN I:=M FI;
                   END;
NOT FOUND: A[I]:=X; B[I]:=0;
FOUND:     B[I]:=B[I]+1;
```

Figure 15: The Code of Example 31A

After performing the suggested modification, the code shown in Figure 16 results. Yet, after studying the modified version of this algorithm, Knuth finds the results inconclusive, the program "perhaps, somewhat easier to understand."

```
I:=H(X);
WHILE A[I]≠0 AND A[I]≠X DO
   IF I=0
      THEN I:=M
      ELSE I:=I-1 FI;
IF A[I]≠X
   THEN A[I]:=X; B[I]:=0 FI;
B[I]:=B[I]+1;
```

Figure 16: The Code of Example 31B

Perhaps the code is not easier to understand! The improvement in clarity, if any, is not apparent. The measure E_c indicates that the code is slightly more difficult to understand, and it is easy to see in Figure 16 why this is so. The logical condition controlling the execution of the WHILE-statement is much more complex, and the condition which actually leads to the termination of the loop must be tested for separately. All of these factors must be considered in studying the code and as a result of the added complexity of Example 31B, the amount of effort required to comprehend the statement of the program will be larger than that expended during a study of Example 31A. Such a conclusion is reasonable and the results obtained using the measure E_c support this position.

Summary and Conclusions

Several factors have been observed which influence the amount of mental effort expended in order to understand a computer program. The fluency of the reader with the programming language employed and the familiarity with the problem area may greatly influence the ease or difficulty of the task[66,68]. In this study, it is assumed that the programmer is fluent in the language employed and not so familiar with the problem area that recognition is accomplished instantaneously, after only a cursory inspection of the code.

Researchers have also attempted to develop relationships between measurable program features such as the degree of documentation, use of indentation and descriptive variable names, and the ease of comprehension[20,66,70,77,78]. While studies found that subjects clearly preferred programs written using such features, no statistically significant differences in comprehension were detected[53,69].

This investigation considers such factors to be of marginal significance. Comparable methods of indentation, paragraphing, and variable nomenclature are assumed to have been employed throughout the program. When these assumptions are not valid, the results which are obtained using the hypothesis formulated may not accurately approximate the observed difficulty in understanding the program.

This study has investigated the relationship which exists between the number and frequency of operators and operands occurring in a program and the observed difficulty experienced in understanding the program. An expression was presented which yields an estimate of the number of elementary mental discriminations performed during comprehension.

In order to test the proposed measure, several published examples demonstrating improvements in program clarity were obtained. In the majority of cases, when the measure was applied the improvements cited in the literature were reflected by a corresponding reduction in the predicted effort for comprehension. When the few cases for which such a reduction was not observed were examined more closely, mitigating factors that tended to support the theory were uncovered. Alternate measures of program clarity, such as the number of executable statements or the program volume, did not reflect the improvements in clarity cited as well.

Early research in software science demonstrated that it had been possible to estimate the amount of effort expended during program construction using a formula closely related to this measure of program clarity[23,84]. Once a programmer understands an algorithm, the additional effort required to express it as a program in a language for which fluency has been achieved is relatively small. In such cases, programming time and comprehension time would be approximately equal. The empirical data available for both small programs and large systems are highly correlated with the measures of

programming and comprehension effort. The data demonstrate that the hypothesis presented is capable of providing reasonable estimates of mental work over a wide range of program sizes $_c$ from programs requiring a fraction of a minute to understand and prepare, to systems which are the product of 100 labor months of effort. Alternate methods of assessing program clarity, for example, functions of the number of lines of code produced, fail to fit the data over such a wide range of program sizes and complexity[76].

A deeper understanding of the relationship which exists between the effort required for program comprehension and that required for program construction has been reached. The initial attempts to provide a measure which would provide a suitable approximation of the amount of effort expended during program construction have not been invalidated. Rather, additional confirmation has been provided supporting the hypothesis that the measure $E\pi$ provides a good approximation of programming effort when the initial version of a program is considered. As further effort is expended to enhance the clarity of the program, succeeding versions will have required a total expenditure of labor which exceeds the estimate provided by the measure $E\pi$. Then, the measure of comprehension effort, E_c, properly reflects the amount of effort required to understand the revised program, and this effort is less in comparison to the effort required to understand the initial version. In effect, such a measure closely approximates a function which yields the minimum mental effort required for either the construction of the program when the version at hand is the initial form produced, or the comprehension of the program in its present form.

Our understanding of the complex problem of measuring program clarity now has a useful, working foundation based on the hypothesis developed and supported by the empirical evidence thus far obtained. More experiments to refine and deepen this knowledge are called for. These experiments may now be performed with a sharper resolution as a result of this research, to verify and replicate the experiments performed here, and to focus on the contribution of less influential features which affect program clarity. Both small and large programs should be included in such studies, and a variety of techniques for assessing comprehensibility employed.

BIBLIOGRAPHY

[1] F. Akiyama, "An Example of Software System Debugging," Proceedings of the IFIP Congress, 1971.

[3] Rudolf Bayer, "A Theoretical Study of Halstead's Software Phenomenon," Technical Report 69, Department of Computer Sciences, Purdue University, West Lafayette, Indiana, May 1972.

[4] Robert Bohrer, "Halstead's Criterion and Statistical Algorithms," Proceedings of the Eighth Annual Computer Science/Statistics Interface Symposium, Los Angeles, California, February 1975.

[5] S. J. Boles and John D. Gould, "A Behavioral Analysis of Programming: On the Frequency of Syntactical Errors," Research Publication RC 3907, IBM Thomas J. Watson Research Center, Yorktown Heights, New York, 1972.

[8] J. Buxton and B. Randell, editors, "Software Engineering Techniques," Report on a Conference Sponsored by the NATO Science Committee, Rome, Italy, 1969.

[9] Louis J. Chmura and Henry F. Ledgard, Cobol With Style: Programming Proverbs, Rochelle Park, New Jersey: Hayden Books, 1976.

[10] Linda M. Cornell and Maurice H. Halstead, "Predicting the Number of Bugs Expected in a Program Module," Technical Report 205, Department of Computer Sciences, Purdue University, West Lafayette, Indiana, October 1976.

[12] Ole-J. Dahl, Edsger W. Dijkstra, and C. A. R. Hoare, Structured Programming, New York, New York: Academic Press, 1972.

[13] Peter J. Denning, "Guest Editor's Overview," ACM Computing Surveys, Volume 6, Number 4, December 1974, pages 209-211.

[14] Edsger W. Dijkstra, "GO TO-Statement Considered Harmful," Communications of the ACM, Volume 11, Number 3, March 1968, pages 147, 148.

[15] James L. Elshoff, "Measuring Commercial PL/I Programs Using Halstead's Criteria," ACM SIGPLAN Notices, Volume 11, Number 5, May 1976, pages 38-46.

[16] James L. Elshoff, "A Numerical Profile of Commercial PL/I Programs," Research Publication 1927, General Motors Research Laboratories, Warren, Michigan, April 1975.

[17] J. C. Emery, "Modular Data Processing Systems Written in COBOL," Communications of the ACM, Volume 5, Number 5, May 1962, pages 263-268.

[17B] Ann Fitzsimmons and Thomas Love, "A Review and Evaluation of Software Science," ACM Computing Surveys, Volume 10, Number 1, March 1978, pages 3-18.

[18] Yasao Funami and Maurice H. Halstead, "A Software Physics Analysis of Akiyama's Debugging Data," Technical Report 144, Department of Computer Sciences, Purdue University, West Lafayette, Indiana, May 1975.

[19] John D. Gannon, "An Experimental Evaluation of Data Types on Programming Reliability," ACM SIGPLAN Notices: Language Design for Reliable Software, Volume 12, Number 3, March 1977, page 141.

[20] John D. Gannon and James J. Horning, "Language Design for Programming Reliability," IEEE Transactions on Software Engineering, Volume SE-1, Number 2, June 1975, pages 179-191.

[21] Thomas S. Gilb, Software Metrics, Cambridge, Massachusetts: Winthrop Publishers, 1977.

[22] Amos N. Gileadi and Henry F. Ledgard, "On a Proposed Measure of Program Structure," ACM SIGPLAN Notices, Volume 9, Number 5, May 1974, pages 31-36.

[23] Ronald D. Gordon and Maurice H. Halstead, "An Experiment Comparing FORTRAN Programming Times With the Software Physics Hypothesis," AFIPS Conference Proceedings, New York, New York, Volume 45, 1976, pages 935-937.

[24] John D. Gould, "Some Psychological Evidence on How People Debug Computer Programs," International Journal of Man-Machine Studies, Volume 7, Number 2, March 1975, pages 151-182.

[25] Maurice H. Halstead, Elements of Software Science, New York, New York: Elsevier North-Holland, Inc., 1977.

[26] Maurice H. Halstead, "Language Level, a Missing Concept in Information Theory," ACM SIGME: Performance Evaluation Review, Volume 2, Number 1, March 1973, pages 7-9.

[27] Maurice H. Halstead, "Natural Laws Controlling Algorithm Structure?", ACM SIGPLAN Notices, Volume 7, Number 2, February 1972, pages 19-26.

[28] Maurice H. Halstead, "Software Physics: Basic Principles," Research Report RJ 1582, IBM Thomas J. Watson Research Center, Yorktown Heights, New York, May 1975.

[29] Maurice H. Halstead, "A Theoretical Relationship Between Mental Work and Machine Language Programming," Technical Report 67, Department of Computer Sciences, Purdue University, West Lafayette, Indiana, February 1972.

[30] Maurice H. Halstead, "Toward a Theoretical Basis for Estimating Programming Efforts," Proceedings of the ACM National Conference, Minneapolis, Minnesota, Volume 30, October 1975, pages 222-224.

[31] Maurice H. Halstead and Rudolf Bayer, "Algorithm Dynamics," Proceedings of the ACM National Conference, Atlanta, Georgia, Volume 28, August 1973, pages 126-135.

[32] Maurice H. Halstead and Paul M. Zislis, "Experimental Verification of Two Theorems of Software Physics," Technical Report 97, Department of Computer Sciences, Purdue University, West Lafayette, Indiana, June 1973.

[33] Maurice H. Halstead, Ronald D. Gordon, and James E. Elshoff, "On Software Physics and GM's PL/I Programs," Research Publication 2175, General Motors Research Laboratories, Warren, Michigan, June 1976.

[34] P. Henderson and R. Snowdon, "An Experiment In Structured Programming," BIT, Volume 12, Number 1, 1972, pages 38-53.

[35] I. D. Hill, R. S. Scowen, and B. A. Wichmann, "Writing Algorithms in ALGOL 60," Software-Practice and Experience, Volume 5, Number 3, July-September 1975, pages 223-244.

[36] Jon F. Hueras and Henry F. Ledgard, "An Automatic Formatting Program for PASCAL," Technical Report 14, Department of Computer and Information Sciences, University of Massachusetts, Amherst, Massachusetts, August 1976.

[37] Thomas E. Hull, "Would You Believe Structured FORTRAN?" ACM SIGNUM Newsletter, Volume 8, Number 4, October 1973, pages 13-16.

[38] Michael A. Jackson, Principles of Program Design, New York, New York: Academic Press, 1975.

[39] Brian W. Kernighan and Phillip J. Plauger, The Elements of Programming Style, New York, New York: McGraw-Hill, 1974.

[40] Brian W. Kernighan and Phillip J. Plauger, "Programming Style: Examples and Counterexamples," ACM Computing Surveys, Volume 6, Number 4, December 1974, pages 303-319.

[42] Donald E. Knuth, "An Empirical Study of FORTRAN Programs," Software-Practice and Experience, Volume 1, 1971, pages 105-133.

[43] Donald E. Knuth, "A Review of 'Structured Programming'," Technical Report 371, Department of Computer Sciences, Stanford University, Stanford, California, June 1973.

[44] Donald E. Knuth, "Structured Programming With GO TO-Statements," ACM Computing Surveys, Volume 6, Number 4, December 1974, pages 261-301.

[45] Donald E. Knuth and R. W. Floyd, "Notes on Avoiding GO TO-Statements," Information Processing Letters, Volume 1, Number 1, February 1971, pages 23-31.

[46] Henry F. Ledgard, "The Case for Structured Programming," BIT, Volume 14, Number 1, 1974, pages 45-57.

[47] Henry F. Ledgard, Programming Proverbs, Rochelle Park, New Jersey: Hayden Books, 1975.

[48] Henry F. Ledgard, Programming Proverbs for FORTRAN Programmers, Rochelle Park, New Jersey: Hayden Books, 1975.

[49] Henry F. Ledgard and William C. Cave, "Cobol Under Control," Communications of the ACM, Volume 19, Number 11, November 1976, pages 601-608.

[50] Henry F. Ledgard and Michael Marcotty, "A Genealogy of Control Structures," Communications of the ACM, Volume 18, Number 11, November 1975, pages 629-639.

[52] L. T. Love and A. B. Bowman, "An Independent Test of the Theory of Software Physics," ACM SIGPLAN Notices, Volume 11, Number 11, November 1976, pages 42-49.

[53] Thomas Love, "An Experimental Investigation of the Effect of Program Structure On Program Understanding," ACM SIGPLAN Notices: Language Design for Reliable Software, Volume 12, Number 3, March 1977, pages 105-113.

[55] Daniel D. McCracken, "Revolution in Programming: An Overview," Datamation, Volume 19, Number 12, December 1973, pages 50-52.

[56] Daniel D. McCracken and Gerald M. Weinberg, "How to Write a Readable FORTRAN Program," Datamation, Volume 18, Number 10, October 1972, pages 73-77.

[57] Clement L. McGowan and John R. Kelly, Top Down Structured Programming Techniques, New York, New York: Petrocelli Books, 1975.

[58] Edward F. Miller, Jr. and George E. Lindamoond, "Structured Programming: Top Down Approach," Datamation, Volume 19, Number 12, December 1973, pages 55-57.

[59] Peter Naur, "GO TO-Statements and Good ALGOL Style," BIT, Volume 3, Number 3, 1963, pages 204-208.

[60] Peter Naur, "Programming Language, Natural Language, and Mathematics," Communications of the ACM, Volume 18, Number 12, December 1975, pages 676-683.

[61] Peter Naur and B. Randell, editors, "Software Engineering," Report on a Conference Sponsored by the NATO Science Committee, Garmische, Germany, 1968.

[62] Karl J. Ottenstein, "A Program to Count Operators and Operands for ANSI-Fortran Modules," Technical Report 196, Department of Computer Sciences, Purdue University, West Lafayette, Indiana, June 1976.

[63] K. V. Roberts, "The Readability of Computer Programs," Computer Bulletin, Volume 10, Number 4, March 1967, pages 17-24.

[65] J. T. Schwartz, "What Constitutes Progress in Programming?" Communications of the ACM, Volume 18, Number 11, November 1975, pages 663, 664.

[66] Ben Shneiderman, "Exploratory Experiments in Programmer Behavior," International Journal of Computer and Information Sciences, Volume 5, Number 2, June 1976.

[67] Ben Shneiderman, "Human Factors Experiments for Developing Quality Software," Department of Information Systems Management, University of Maryland, College Park, Maryland, (unpublished paper).

[68] Ben Shneiderman, "Measuring Program Quality and Comprehension," Technical Report 16, Department of Information Systems Management, University of Maryland, College Park, Maryland, February 1977.

[69] Ben Shneiderman and D. McKay, "Experimental Investigations of Computer Debugging and Modification," Proceedings of the 6th International Congress of the International Erogonomics Association, July 1976.

[70] Ben Shneiderman, R. Mayer, D. McKay, and P. Heller, "Experimental Investigation of the Utility of Detailed Flowcharts in Programming," Communications of the ACM, (to appear, 1977).

[71] M. E. Sime, T. R. Green, and D. J. Guest, "Psychological Evaluation of Two Conditional Constructions Used in Computer Languages," International Journal of Man-Machine Studies, Volume 5, Number 1, January 1973, pages 105-113.

[72] Michael J. Spier, "A Critical Look at the State of Our Science," ACM SIGOPS Notices, Volume 8, Number 2, April 1974, pages 9-15.

[73] Michael J. Spier, "Software Malpractice - A Distasteful Experience," Software-Practice and Experience, Volume 6, Number 3, July-September 1976, pages 293-299.

[74] Thomas B. Steel, Jr., "Guest Editorial," IEEE Transactions on Software Engineering, Volume SE-1, Number 4, December 1975, page 349.

[76] Claude E. Walston and Charles P. Felix, "A Method of Program Measurement and Estimation," IBM Systems Journal, Volume 16, Number 1, 1977, pages 54-73.

[77] Laurence M. Weissman, "A Methodology for Studying the Psychological Complexity of Computer Programs," Ph.D. Thesis, Department of Computer Science, University of Toronto, Toronto, Ontario, Canada, 1974.

[78] Laurence M. Weissman, "Psychological Complexity of Computer Programs: An Experimental Methodology," ACM SIGPLAN Notices, Volume 9, Number 6, June 1974, pages 25-36.

[79] Niklaus Wirth, "An Assessment of the Programming Language PASCAL," IEEE Transactions on Software Engineering, Volume SE-1, Number 2, June 1975, pages 192-198.

[80] Niklaus Wirth, "On the Composition of Well-Structured Programs," ACM Computing Surveys, Volume 6, Number 4, December 1974, pages 247-259.

[81] James M. Yohe, "An Overview of Programming Practices," ACM Computing Surveys, Volume 6, Number 4, December 1974, pages 221-245.

[82] Edward A. Youngs, "Human Errors in Programming," International Journal of Man-Machine Studies, Volume 6, Number 3, May 1974, pages 361-376.

[83] Edward Yourdan, Techniques of Program Structure and Design, Englewood Cliffs, New Jersey: Prentice-Hall, 1975.

[84] Paul M. Zisls, "An Experiment in Algorithm Implementation," Technical Report 96, Department of Computer Sciences, Purdue University, West Lafayette, Indiana, June 1973.