Potential Impacts of Software Science on Software Life Cycle Management

M. H. Halstead

Report Number:
77-237

Potential Impacts of Software Science

on

Software Life Cycle Management

M. H. Halstead
Purdue University

CSD-TR 237
July 1977

Prepared for:

Human Factors, Individual and Group
Productivity Measures Session,
Software Life Cycle Management Workshop
U.S. Army Computer Systems Command,
Airlie House, Airlie, Virginia, August 22-23, 1977

Research sponsored in part by

NSF Grant No. MCS 7605611

2 77

Potential Impacts of Software Science on Software Life Cycle Management

M. H. Halstead
Purdue University

Abstract

A listing of present needs in Software Engineering is followed by a brief discussion of new and "quasi-complete" engineering disciplines and their relation to corresponding branches of natural science. Recent results in Software Science are then described, suggesting a natural science base. The paper concludes with specific suggestions of areas in which these results might well provide guidance and insight into problems of software development and maintenance.

Engineering Needs

The successful management of large programming projects over their complete life cycles depends largely upon the discipline of Software Engineering. But in its present state of development this discipline still falls far short of being a "quasi-complete" branch of engineering.

Substantial progress is urgently needed on many fronts. A handful might be mentioned. 1). *Problem Specifications.* Software Engineering should be able to provide optimal methods for obtaining them, resolving their ambiguities and incompatibilities, determining their completeness, and the "best" way to present them to programmers. 2). *Programmer Productivity.* In this area, a "quasi-complete" discipline would provide guidelines for programmer selection, training, and subsequent evaluation; as well as quantitative methods for estimating the man-hours and organization required to achieve any well specified goal. 3). *Program Testing.* Quantitative methods are needed for estimating the effects of source language, modularity, program level or complexity, volume and program clarity upon program testing and maintenance. 4). *Job Scheduling.* More reliable techniques are needed for estimating man-power requirements of a project as a function of specifications, programming language, team size and mix, memory constraints and required reliability. 5). *Optimizing Implementation versus Maintenance costs.* A quasi-complete engineering discipline should provide quantitative guidelines for sound tradeoff studies in this important but complex area.

Natural Science and Engineering

For any engineering discipline to be quasi-complete, it must rest upon and be grounded in a "hard" natural science, a science with sound metrics, reproducible experiments, and dependable "laws". In virtually all cases, branches of engineering have preceded (and perhaps stimulated) the natural science upon which they are now based. During that time, however, their value to mankind was severaly limited. But now, for example, aeronautical engineering based on fluid dynamics, power engineering based on thermodynamics, electrical engineering based on electrodynamics, and mechanical engineering based on statics, dynamics and strength of materials may all be considered quasi-complete, highly competent, and useful.

Software Science

A considerable body of evidence now exists which suggests that the metrics, methods, and hypotheses of software science [7] may be capable of providing such a base for software engineering. It is pertinent to note, however, that theories are not theorems. They require independent experimental confirmation at more than one laboratory. Unlike mere mathematical models, theories must provide new insight into natural phenomena, and they only become important when they are shown to predict previously unrecognized and unexpected relationships in areas beyond their originally intended scope. Further, a theory is never complete, but continues to be used only until its recognized inadequacies can be eliminated by a new theory.

Software science is based on a handful of language independent parameters which can be measured (or counted) directly from any hard copy or computer program. These are the number of unique operators (η_1), the number of unique operands (η_2), the total usage of operators (N_1), and the total usage of operands (N_2). A fifth parameter, the number of conceptually unique input/output operands (η_2^*) required by a procedure call upon the program is also an important language-independent metric.

These basic metrics are not independent, and a number of quite useful relationships have been found among them. For example, denoting the sum of N_1 and N_2 as the length N, it has been found that programs tend to obey the relation

$$N = \eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2$$

Further, denoting the sum of η_1 and η_2 as the vocabulary η, the program volume V is

$$V = N \log_2 \eta$$

The potential (or least possible) volume V^* is

$$V^* = (2 + \eta_2^*) \log_2 (2 + \eta_2^*)$$

which gives an implementation level L of

$$L = V^*/V \simeq \frac{2}{\eta_2} \frac{\eta_2}{N_2}$$

It follows that the product

$$V^* = LV$$

is invariant under translation from one language to another.

It has also been found that the number of elementary mental discriminations (E) required to produce a program should be given by

$$E = V/L$$

This leads directly to an estimate of programming time (T). Using the Stroud Number (S), or 18 elementary mental discriminations per second, gives

$$T = E/S = V/SL$$

Initially unforseen relationships derivable from the basic metrics include Ostapko's [10] derivation of Rent's Rule for circuit to pin ratios in hardware, Elci's [3] demonstration that the lengths of operating systems are functionally related to the number of their allocatable resources, and Funami's [5] demonstration that programming error rates are related to E. Perhaps the most interesting unexpected finding to date is the observation that the relationships governing computer programs can be applied to technical prose as well.

With respect to deeper understanding or insight, one might list the areas of program purity or "impurity classes", the role of modularity, the quantitative effects of "GO-TO's", the measurement of clarity, and most recently some apparent insight into the learning process itself.

Independent experimental verifications of various facets of the overall theory have been published by Bohrer [2] of Illinois, Elshof [4] of General Motors, Bell and Sullivan [1] of Mitre, Ostapko [10] of IBM and Love and Bowman [9] of General Electric.

Experimental Methodology

To illustrate the relationships and methodology discussed above, we will first present the results and analysis of a simple experiment, and follow with a few summaries of previously published data.

In January 1977 a class of 28 advanced graduate students at Purdue individually programmed Euclid's greatest common divisor algorithm in Fortran, and counted the software parameters in their own versions. Their results are given in Table 1. Students 10, 16 and 27 neglected the implied "End-of-line" operator in Fortran, so their reported values of η_1 have been increased by one, and their values of N_1 have been increased by 12.

Table 1. Software Parameters of 28 Independent Fortran Versions of the GCD Algorithm

Student	n_1	n_2	N_1	N_2	Student	n_1	n_2	N_1	N_2
1	11	6	34	21	15	11	6	34	21
2	11	5	32	19	16	12	7	34	21
3	12	6	34	21	17	12	6	38	21
4	12	6	34	21	18	11	6	33	21
5	10	6	31	21	19	12	6	34	21
6	12	6	34	21	20	11	6	34	21
7	11	6	34	21	21	10	6	34	21
8	11	6	31	21	22	11	6	34	21
9	12	6	34	21	23	11	6	32	21
10	12	6	36	21	24	12	6	35	21
11	11	6	35	21	25	12	5	33	19
12	11	6	34	21	26	12	6	32	21
13	11	5	33	19	27	11	6	34	21
14	12	6	34	21	28	10	6	31	21
					MEANS	11.32	5.93	33.64	20.79
					S.D.	.67	.38	1.50	.63

Using the individual values in Table 1, a number of the software relationships can be evaluated, and the degree of conformity calculated.

Length

Obtaining the observed value of length (N) from

$$N = N_1 + N_2$$

and the estimated length ($\hat{N}$) from

$$\hat{N} = n_1 \log_2 n_1 + n_2 \log_2 n_2$$

gives mean values of

$$N = 54.43 \pm 1.75$$

$$\hat{N} = 54.90 \pm 3.76$$

$$(N-\hat{N})/N = -0.009 \pm 0.058$$

Implementation Level

Obtaining the observed potential volume (V^*) from the condition that a procedure call on a GCD algorithm must have two inputs and one output, or $\eta_2^* = 3$ in

$$V^* = (2 + \eta_2^*) \log_2 (2 + \eta_2^*)$$

and the observed volume (V) from

$$V = (N_1 + N_2) \log_2 (\eta_1 + \eta_2)$$

allows the observed level (L) to be calculated from

$$L = V^*/V$$

The estimated level (L) is obtained from

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

Then for Table 1, the mean values are

$$L = 0.0529 \pm 0.0023$$

$$\hat{L} = 0.0505 \pm 0.0035$$

$$(L-\hat{L})/L = 0.028 \pm 0.067$$

Potential Volume

Obtaining the estimated potential volume ($\hat{V}^*$) from

$$\hat{V}^* = \hat{L}V = \frac{2}{\eta_1} \frac{\eta_2}{N_2} (N_1 + N_2) \log_2 (\eta_1 + \eta_2)$$

and the actual potential volume (V^*) from $\eta_2^* = 3$ yields

$$V^* = 11.61 \pm 0.00$$

$$\hat{V}^* = 11.28 \pm 0.78$$

$$(V^*-\hat{V}^*)/V^* = 0.028 \pm 0.067$$

Vocabulary

Obtaining the observed vocabulary (η) from

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

and the estimated vocabulary ($\hat{\eta}$) by solving iteratively for η as a function of N in

$$N_1 + N_2 = \eta \log_2 (\eta/2)$$

gives the mean values

$$\eta = 17.25 \pm 0.80$$

$$\hat{\eta}(N) = 17.42 \pm 0.38$$

$$(\eta - \hat{\eta}(N))/\eta = -0.016 \pm 0.038$$

Unique Operators

Using the vocabulary $\hat{\eta}(N)$ estimated from length as calculated above, the estimated unique operator count $\hat{\eta}_1(N)$ can be obtained from

$$\eta_1 = (\eta - B)/(A+1)$$

where

$$A = \frac{\eta_2^* \log_2(\eta_2^*/2)}{2 + \eta_2^*} \quad ; \quad B = \eta_2^* - 2A$$

The data in Table 1 give the following average values

$$\eta_1 = 11.31 \pm 0.67$$

$$\hat{\eta}_1(N) = 11.20 \pm 0.28$$

$$(\eta_1 - \hat{\eta}_1(N))/\eta_1 = 0.008 \pm 0.055$$

Unique Operands

In the same way, the observed η_2 can be estimated from η_2^* and length via

$$\hat{\eta}_2(N) = (A\hat{\eta}(N) + B)/(A + 1)$$

And again, the Table 1 data show

$$\eta_2 = 5.93 \pm 0.38$$

$$\hat{\eta}_2(N) = 6.23 \pm 0.10$$

$$(\eta_2-\hat{\eta}_2(N))/\eta_2 = -0.054 \pm 0.064$$

Summarizing the relative errors, we have

Length (N)	-0.009 ± 0.058
Level (L)	0.028 ± 0.067
Potential Volume (V*)	0.028 ± 0.067
Vocabulary (η)	-0.016 ± 0.038
Operators (η_1)	0.008 ± 0.055
Operands (η_2)	-0.054 ± 0.064

This can be taken as evidence that for a very small program, replicated by 28 highly fluent programmers, the software hypotheses tested gave reasonably good agreement with the observations.

In order to illustrate the applicability of these relationships to programs large enough to be of practical interest, Elshof's [4] data validating the length equation for commercial PL/1 programs are given in Table 2.

Table 2. Elshof's PL/1 Data

	Number of Programs in Size Class	Length Observed (N)	Length Predicted ($\hat{N}$)
	3	18,592	19,091
	17	10,685	11,049
	23	5,751	6,005
	39	3,165	3,318
	17	1,590	1,663
	11	831	911
	4	369	522
	5	198	195
	1	122	129
Totals	120	41,303	42,883

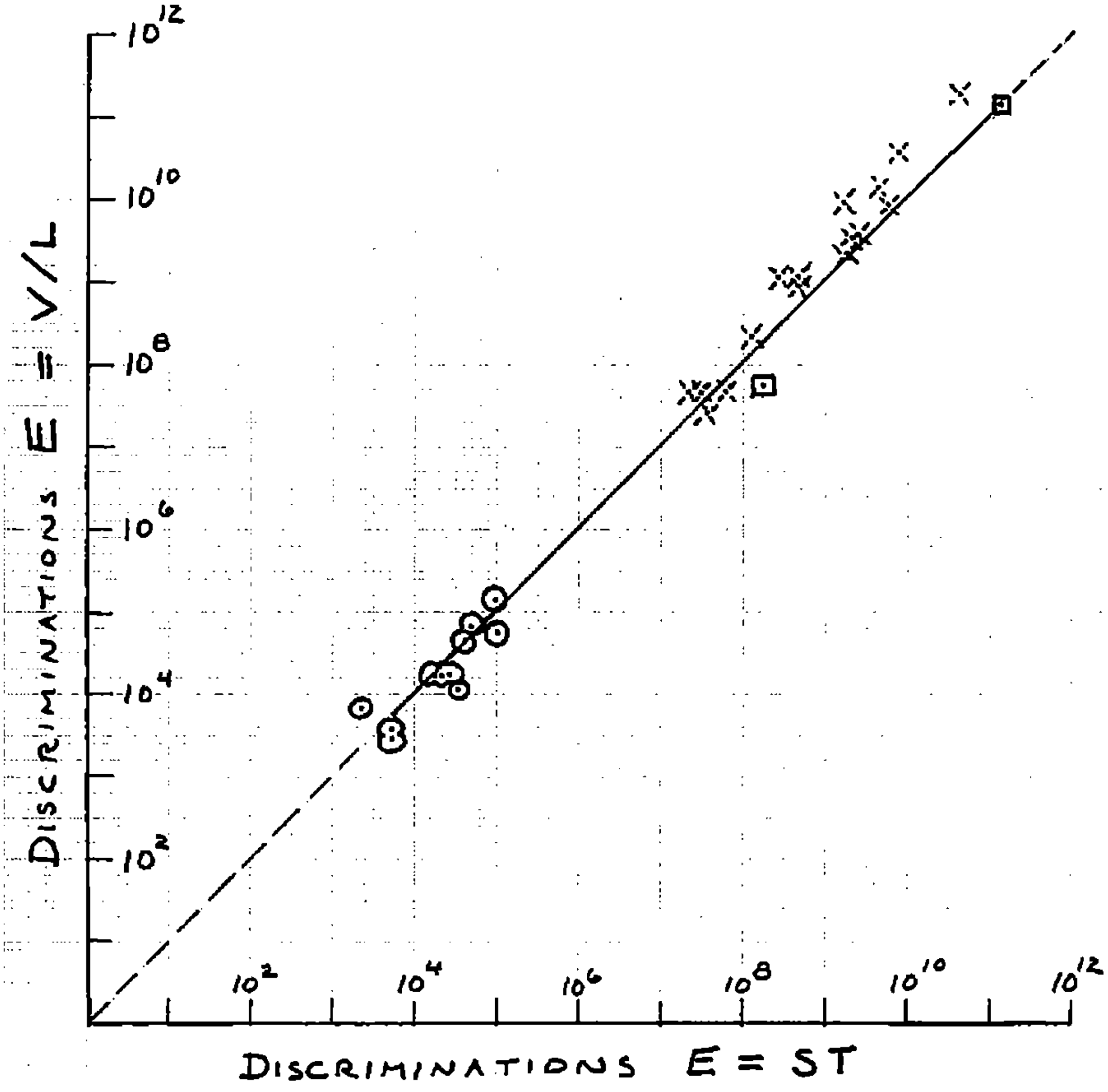
FIGURE 1. ELEMENTARY MENTAL DISCRIMINATIONS

⊙ GORDON-HALSTEAD DATA

✕ JOHNSON DATA

⊡ WALSTON-FELIX DATA

SOLID LINE EXTENDS FROM 5 MINUTES TO 1000 MAN-YEARS. S = 18 disc./sec.



20 Squares to the Inch

Figure 1 has been taken from another paper [8], which indicates that projects ranging from well under one day to well over one hundred man years follow the Effort Equation in a general way.

Indicated Studies

A number of areas in which software science might prove useful to Software Life Cycle Management come immediately to mind. While each of the five areas cited in the introduction as needing substantial improvement should benefit directly, we can perhaps be more specific.

With respect to *task specifications*, for example, one might suggest five steps.

1) Start with small tasks, requiring only one programmer from 10 minutes to 8 hours to implement, and gather a substantial number of samples.
2) Analyze the technical English problem statements, obtaining η, N, V, L and V*, E and T.
3) Perform a similar analysis of the resulting programs.
4) Study the effect of different methods and techniques of problem statement on the resultant small programs.
5) Expand the study to large programs.

Similarly, *programmer productivity* relationships are sufficiently important to warrant large scale investigations.

1) Repeat experiments to determine individual programmer variances between actual and calculated programming times.
2) Note that the calculated values of $\hat{T}$ for very large programs have all been based upon average values of language level. Because deviations from average would be expected to contribute to the observed variance in T, it should be illuminating to actually measure this effect.

3) For a significantly large data base, obtain the statistical variance between observed and calculated programming times.

4) Perform quantitative studies on the effect of E of existing programs on the rate of error discovery by new programmers. This should yield a measure of their fluency (and concentration).

5) Investigate the possibility that programming aptitude might be estimated by a software analysis of a technical prose paragraph written by a candidate for programmer training.

The area of *program testing* might benefit by further investigations of the software relationships. This might require

1) Sharpening the definition of "Delivered" bugs.

2) Development of a large data base, with samples from most of the widely used languages.

3) Repetition of experiments to determine the expected variance between observed and calculated error rates.

4) Analysis of modularity, following Zislis' [11] use of software science for program testing.

5) Use of software error rate relations in predicting remaining errors as a function of expected errors and errors removed.

The area of *job scheduling* is related to that of programmer productivity, but requires other information as well. Consequently, it involves a number of additional points.

1) Because any two independent software parameters determine all others, it follows that the task specifications and the language to be used determine, in principle, the time to be required. In practice,

however, it is not that simple. For example, even a concentrating programmer, fluent in the language, without computer memory constraints, must start with a complete problem statement. Furthermore, a problem statement which contains no contradictions or ambiguities may be "complete" for one programmer, but not for another. Nevertheless, the existence of a basic relationship between the number of conceptually unique input/output operands (η_2^*), the language level (λ), and the time (T) suggests that an intensive investigation is now possible and warranted.

2) It has been observed that the product $LV = V^*$ is invariant when a given algorithm is translated from one language to another. But within any one language, L decreases as the potential volume increases. Consequently, a given language can be characterized by its language level (λ), defined as

$$\lambda = LV^*$$

Algebraically, this results in

$$T = V/SL = V^{*3}/S\lambda^2$$

and consequently λ is a parameter of considerable interest. While the mean value of λ appears to lie somewhere near one for a number of programming languages, it has a large variance which appears to increase as the mean increases. Because the data so far available is based on small samples of small programs, it can not be used with confidence. Therefore, statistical determinations of means and variances of λ for any language of interest should be made. This study might well include investigations of the effect of different programming methodologies within a single language. It could also be extended to an analysis of any proposed new language. It could then serve in trade-off studies on cost versus benefits of change to a higher level language, or the question of special purpose versus general purpose languages.

With respect to the problems of *optimizing implementation costs versus maintenance costs*, it appears quite likely that the quantitative approach provided by software science can be of considerable value. This results from recent work of Gordon [6], who has shown that the measure of elementary discriminations E is in an interesting sense ambiguous.

In the usual case of program implementation, E does indeed measure the time required to develop the program. If, however, additional time is then spent in improving or increasing the legibility of the program, the effect is to reduce the final value of E, rather than to increase it. The final value of E for an improved program then represents not the total effort to write it, but a measure of the effort to understand it -- a measure of clarity.

This suggests that a quantitative measure of clarity could be made before a program is polished. Then, the amount of effort which could advantageously be used in increasing the clarity could be determined on the basis of the needs of maintenance.

References:

[1] Bell, D. E., and J. E. Sullivan. "Further Investigations into the Complexity of Software". MITRE Technical Report 2874, Vol. II, June 30, 1974.

[2] Bohrer, Robert. "Halstead's Criterion and Statistical Algorithms." Proceedings of the Eighth Annual Computer Science/Statistics Interface Symposium, Los Angeles, February 1975, pp. 262-266.

[3] Elci, Atilla. "Factors Effecting the Program Size of Control Functions of Operating Systems." Ph.D. Thesis, Purdue University, December 1975.

[4] Elshoff, James L. "Measuring Commercial PL/1 Programs Using Halstead's Criteria", ACM SIGPLAN Notices Vol. 7, No. 5, May 1976.

[5] Funami, Yasuo, and M. H. Halstead. "Software Physics Analysis of Akiyama's Debugging Data", Proc. MRI XXIV International Symposium: Software Engineering. New York. Polytechnic Press, 1976.

[6] Gordon, R. D., "A Measure of Mental Effort Related to Program Clarity", Ph.D. Thesis, Purdue University, August 1977.

[7] Halstead, M. H. "Elements of Software Science", Elsevier North-Holland Publishers. New York 1977.

[8] Halstead, M. H. "A Quantitative Connection Between Computer Programs and Technical Prose," Digest of Papers from COMPCON 77 Fall, IEEE Computer Society.

[9] Love, L. T. and A. B. Bowman, "An Independent Test of the Theory of Software Physics". ACM SIGPLAN Notices, Vol 11. No. 11, November 1976, pp 42-49.

[10] Ostapko, Daniel L. "On Deriving a Relation Between Circuits and Input/Output by Analyzing an Equivalent Program". ACM SIGPLAN Notices, Vol 8, No. 6, June 1974, pp 18-24.

[11] Zislis, Paul M. "Semantic Decomposition of Computer Programs: An Aid to Program Testing", ACTA Informatica, Vol 4, 1975 pp 245-269.