This paper provides an overview of a new approach to the measurement of software. The measurements are based on the count of operators and operands contained in a program. The measurement methodologies are consistent across programming language barriers. Practical significance is discussed, and areas are identified for additional research and validation.

A perspective on software science

by K. Christensen, G. P. Fitsos, and C. P. Smith

Measurement of programs is still a fairly subjective process. We can measure size, based on *line of code* or *number of statements*, but acceptance of these measures is not universal. Acceptance of lines of code, as an example, seems to be based on the view that although lines of code may be an imprecise measure, it is something that can be enumerated, and until something better is discovered we will continue to use it. There is a veiled invitation here to find something better.

Measurement of *program complexity* has not gained the level of acceptance of size measurement, probably because it is a more elusive object to quantify. Most current activity is oriented to the counting of decision nodes in a program. Although the use of decision nodes to measure complexity may seem subjective, there is evidence to suggest a connection between decision nodes and complexity. Structured programming concepts, for example, organize programs to minimize the effects of decision nodes. This suggests a tendency to accept the notion of complexity and reinforces the tendency to measure it with decision nodes.

Both size and complexity are measured after the fact. That is, measurement is not possible until the code has been written. Elements of measurements can be considered if logic is outlined before code has been written. However, measurements are rarely made until after writing the code. Even then, measurements tend to be defense mechanisms against problems identified by other means, such as late schedules and high defect levels.

A more ideal situation would be to use measurements that can lead to the optimization of program organization while the program is being written or while it is being designed. In other engineering disciplines, measurement is an inherent part of the optimization process. Software engineering also needs a measurement discipline that each programmer can understand and can relate to choices made while designing and coding a program.

A new approach to code measurement was suggested by Halstead[1] in which lines of code are broken down into atomic particles of *operators* and *operands*. The relationships between the particles then provide more than one dimension for measurement. In contrast to the use of numbers of statements, the following are three general advantages of the operator-operand approach:

- An explainable methodology for calibrating a measurement instrument.
- A more nearly universal measure, since the approach is consistent across the boundaries of programming languages.
- The ability to relate some of the effects of programming style to measured quantities.

Some aspects of the approach may prove to be imprecise, but the concepts are interesting. If shown to be practical, the power of the approach could be a significant step toward an engineering-like code measurement methodology.

Measurement definitions

The particles of a given source program are operators and operands. *Operands* have values that are changed or are used as reference for change (constants and variables). *Operators* are the operation codes, delimiters, arithmetic symbols, punctuation, etc., that operate on or with operands. There are also operators such as branches, DO WHILE, IF THEN, etc., that control the sequence of operation.

Examples of operand and operator types are the following:

- Variable name—operand.
- Literal—operand.
- Arithmetic symbol—operator.
- Punctuation—operator.
- End of statement delimiter—operator.

There is a methodology for validating rules and calibrating a measurement program.[2] Although the methodology is not discussed here, some of the not-so-obvious rules are the following:

- Parentheses and brackets always come in pairs, and a compiler diagnoses correct pairing. Each pair is counted as a single "grouping" operator.
- GO TO statements are concatenated with the address of the GO TO to form a single operator.
- IF and THEN are combined into a single operator since one is unlikely without the other.
- IF THEN and ELSE are also combined as a single operator. (Thus, IF THEN ELSE and IF THEN are two separate and distinct operators.)
- Each of the possible combinations of DO UNTIL, DO WHILE, etc. is combined as a single operator, but each combination is separate from the others.

A general observation is that the rules seem to combine lines of code (end of statement delimiter), decision nodes (IF THEN ELSE, DO WHILE, DO UNTIL, etc.), as well as operation codes, variables, and punctuation. We may question whether their use is properly weighted, but we cannot help but notice that they are all included.

basic measures

The following are the four basic program measures or metrics:

- η_1 Number of unique operators used.
- η_2 Number of unique operands used.
- N_1 Number of times operators are used.
- N_2 Number of times operands are used.

Vocabulary (η) of a given program is defined as the sum of unique operators and operands used in that program, and is a measure of the repertoire of elements that a programmer must deal with to implement the program. Thus Vocabulary is defined as follows:

$$\text{Vocabulary} = \eta = \eta_1 + \eta_2.$$

Length (N) of a given program is defined as the sum of the operator usage and the operand usage. Intuitively, length is a measure of program size, and measures the number of times a programmer deals with each of the programming elements. Length is expressed as follows:

$$\text{Length} = N = N_1 + N_2. \tag{1}$$

length and vocabulary relationships

Halstead suggests a relationship such that Length can be estimated from Vocabulary. The formula for *Estimated Length* ($\hat{N}$) based on Vocabulary is the following:

$$\text{Estimated Length} = \hat{N} = \eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2. \tag{2}$$

Construction of an experiment to test Equation 2 is relatively straightforward. One need only measure the basic metrics for a set of programs, calculate Estimated Length ($\hat{N}$) from Equation 2, and compare the result with observed Length (N) from Equation 1. This experiment has been conducted a number of

Table 1 Summary of experiments correlating Estimated Length ($\hat{N}$) and Observed Length (N)

Language	*Number of programs*	*Correlation coefficient*	*Cited reference*
FORTRAN	429	0.95	1
PL/I	120	0.98	8
COBOL	264	0.90+	6
System/370 assembly language	994	0.90+	4
PL/S	643	0.90+	4
COBOL	24	0.92	9
APL	29	0.96	9
RPG	371	0.94	10

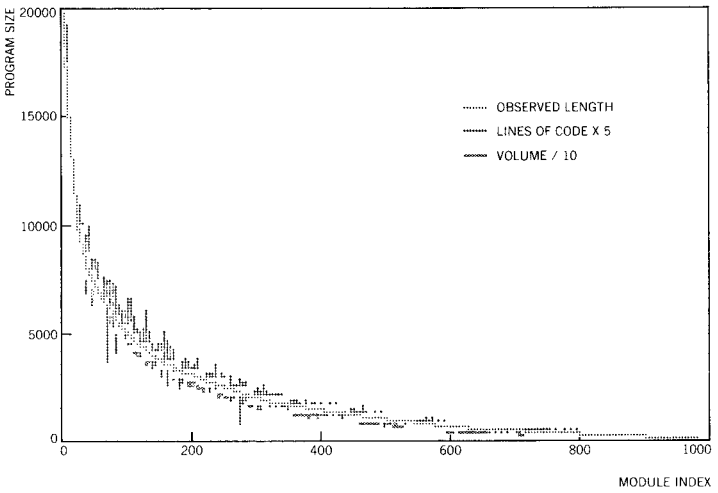
times, and the results are summarized in Table 1. These results are sufficient to have practical significance. It should be pointed out that not every program complies with the rules. There can be wide differences between Estimated Length and observed Length, but for a large population of programs there is reasonable correlation. Halstead[3] took the view that software science is similar to actuarial statistics, in that, for example, one might find that men at age 65 have a life expectancy of 14 years. This, however, is no guarantee for any particular 65-year-old individual. In other words, the accuracy of the actuarial prediction is completely adequate, but its precision might be too poor for any individual—person or program.

To be accurate, Smith,[4] Fitsos,[5] and Shen and Dunsmore[6] have observed that Estimated Length tends to be low for large programs and high for small programs. The formula, Equation 2, for Estimated Length seems to be most accurate in the range 2000 to 4000 units of length. Feuer and Fowlkes[7] also report that the length equation overestimates the actual length 80 percent of the time for 197 PL/I programs. (Most of these are small programs, i.e., below 2000 units of length). This behavior also seems to be language-independent, as has been observed for the three languages, COBOL, System/370 assembler language, and PL/S, and may be true for PL/I.

Having shown that Length can be estimated from Vocabulary with reasonable accuracy, we can formulate our first general rule as follows:

Rule 1. Length (N) of a program is a function of Vocabulary (η) for that program.

Figure 1 Program size expressed as observed Length, lines of code, and Volume ordered by Length (N) for a total of 992 System/370 assembler language modules



Program size

Although length of a program can be considered a measure of program size, Halstead suggests a two-dimensional approach that considers the number of times elements are used in a program versus the repertoire of elements from which selections must be made. This notion is expressed as *Volume* (V) of a program and is a function of the number of selections required (N) from a Vocabulary (η), which is given as follows:

$$\text{Volume} = V = N \log_2 \eta. \tag{3}$$

To understand the value of Equation 3, experiments were conducted at the IBM Santa Teresa Programming Laboratory and documented by Smith.[4] The usual measure of program size at the Laboratory is lines of code, and a single tool is used to measure programs written in both System/370 assembler language and PL/S. Lines of code, Length, and Volume are plotted in Figures 1 and 2. Table 2 summarizes the correlation coefficients.

It is unfortunate that consistent rules for measuring lines of code have not been developed for other languages. This prevents extensive validation of the observations that follow. It is possible, however, to speculate that Length and/or Volume may be more universal size measures than lines of code, since the definition of operator and operand is consistent across language barriers. Table 2 indicates that the following rules are true for at least assembler language and PL/S. Feuer and Fowlkes[7] also observed a

Figure 2 Program size expressed as observed Length, lines of code, and Volume ordered by Length (N) for a total of 643 PL/S modules

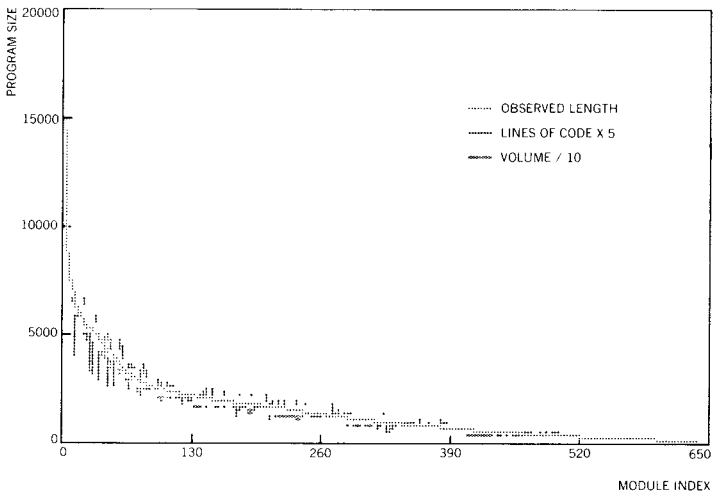


Table 2 Correlation coefficients for lines of code, Length, and Volume as measures of program size

Assembler language correlation coefficient between lines of code and length	0.995
Assembler language correlation coefficient between lines of code and volume	0.992
PL/S correlation coefficient between lines of code and length	0.952
PL/S correlation coefficient between lines of code and volume	0.929

0.95 percent correlation coefficient between PL/I clauses and Length, indicating that the rules may also be true for PL/I.

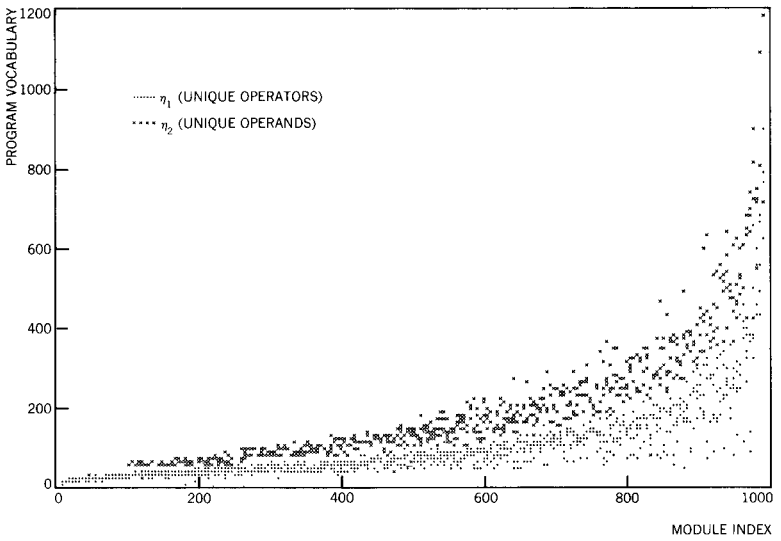
Rule 2. Lines of code, Length, and Volume are equally valid as relative measures of program size.

Combining Rule 1 and Rule 2 we can also state Rule 3.

Rule 3. Program size, measured in any of the three terms, is a function of Vocabulary (η).

It is not too speculative to acknowledge that Length and Volume are equally valid measures of size for any language, since it has

Figure 3 Program Vocabulary in terms of unique operators and operands ordered by Length (N) for a total of 992 System/370 assembler language modules

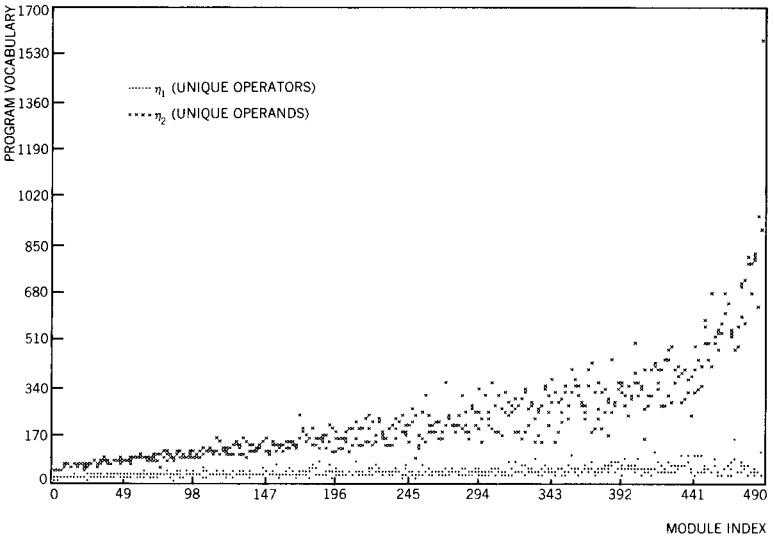


already been shown that Length is a function of Vocabulary and that Volume is a function of Length and Vocabulary. What remains, then, is to gain confidence in these metrics as measures of size. Acceptance probably depends on the availability of easy-to-use analyzers and sufficient experience to gain confidence.

Program vocabulary

Since Vocabulary is so important to the previous measures, it is helpful to decompose it. Fitsos[5] has plotted the elements of Vocabulary by program size, as shown in Figures 3 and 4 for 992 assembler language programs and 490 PL/S programs. These programs are ordered by Length with the largest on the right. By observation we see that for assembly language programs η_1 and η_2 both tend to increase with program Length. For PL/S programs, η_1 tends to be flat whereas η_2 increases with program Length. This flat η_1 characteristic is significant, and it is important to understand the cause. There are spikes on the η_1 line that may represent deviations from an implied norm. Figure 5 shows the distribution of number of programs having each value of η_1. By analyzing programs with η_1 greater than the means plus one standard deviation, we can search for abnormalities. What Fitsos found was that, for programs with high η_1, the predominant reason was a high usage of GO TO.

Figure 4 Program Vocabulary in terms of unique operators and operands ordered by Length (N) for a total of 490 PL/S program modules



Recalling the rules for counting GO TO (i.e., GO TO concatenated with address to form a unique operator), the explanation for a flat η_1 is clarified. If one accepts the absence of GO TO as an indicator of compliance with structured programming rules, then, for PL/S, η_1 tends to be a constant for structured programs. It is difficult to imagine assembly language programs that do not use branch instructions. In fact, structured programs cannot be written in assembly language without adding macros to avoid the use of branches. One could argue that the addition of macros changes the language into something else, i.e., a higher level of language.

From these considerations combined with Rule 3 we can formulate the following additional rule:

Rule 4. For structured programs, program size is a function of the Vocabulary of Operands (η_2).

In more general terms, it seems that program size is determined by the data that must be processed by the program. Whether Rule 4 is true for all languages remains to be demonstrated with further experimentation. In Figure 6, we have plotted vocabulary detail for 34 PL/I programs measured by Elshoff.[8] Here, η_1 tends to be flat, but the sample is not very large, and the observations concerning GO TO cannot be tested using published information. The authors believe the rule has promise for PL/I since PL/S is a subset of PL/I. More concrete proof would be comforting.

Figure 5 Number of programs versus value of η_1 for 490 PL/S program modules (mean = 46, mode = 38, and median = 42)

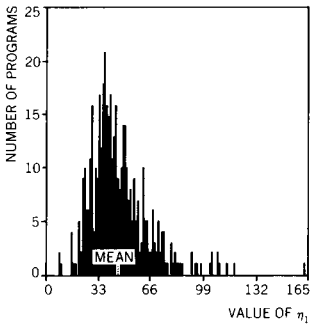
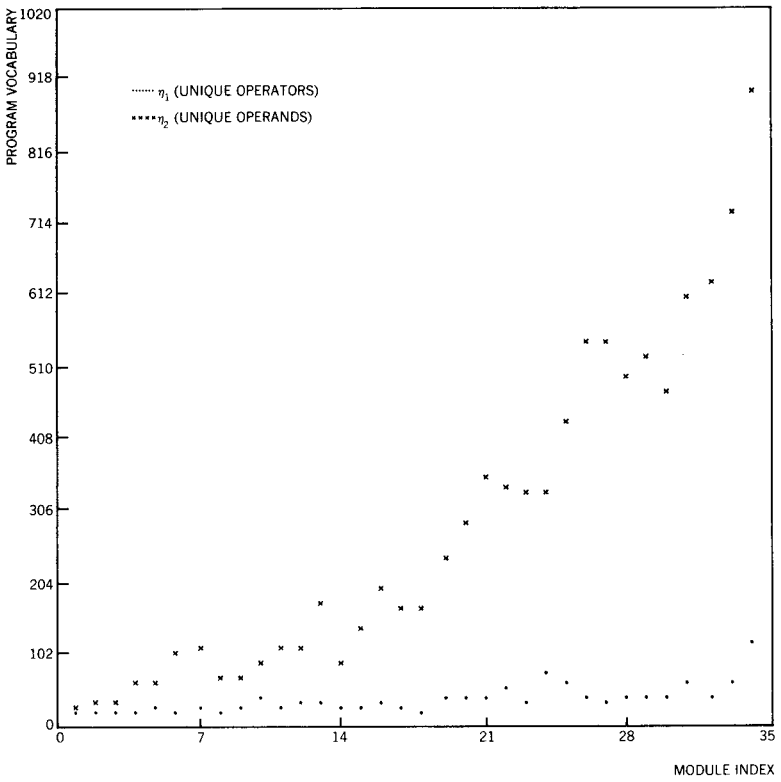


Figure 6 Program Vocabulary in terms of unique operators and operands ordered by Length (N) for a total of 34 PL/I programs



Given that Rule 4 is shown to be correct for all programming languages, and also given that Length and/or Volume become accepted as measures of size, the implications are encouraging. Most program design activity creates a detailed layout of data elements that must be processed. Given further a desire to contain program size below some optimum level, detailed data layouts can be used to estimate program size. The effect of data organization on program size can be predicted, and the reorganization of planned program packages can be addressed before coding activity begins. This methodology is certainly more explainable than current techniques.

Program difficulty

Halstead defined *Difficulty* (D) as a metric that expresses the difficulty of writing code. It includes considerations for decision nodes, the repertoire of operators that a programmer must deal

with, and the conciseness with which he deals with variables. It appears to be a measure of "ease of reading" as well as a measure of "ease of writing." The expression for the Difficulty of writing code is the following:

$$\text{Difficulty} = D = \frac{\eta_1}{2}\frac{N_2}{\eta_2}. \tag{4}$$

Although an explanation for the basis of Equation 4 is tedious, an intuitive frame of reference is helpful. The ratio $\eta_1/2$ includes consideration of the difficulty of dealing with a large vocabulary of operators. Since no program can be written with less than two operators (function call and end of statement) the ratio measures the distance from absolute zero, so to speak. Aside from previous discussions on the use of GO TO, consider a reference card for the list of operation codes available in an assembler language in comparison to the list of key word operators in such languages as FORTRAN, COBOL, and PL/I. The higher-level languages remove concerns for register usage, word boundaries, and data representation. A smaller repertoire of operators is required to implement the same algorithm in a high-level language.

The ratio N_2/η_2 represents the average number of times operands are used. In a program where each operand is used only once this ratio is 1. The more frequently a variable is changed in a program, the more difficult it is to retain its current value in one's mind.

Difficulty (D), therefore, affects the effort required to code an algorithm, to inspect and review it, and to understand it later when alteration is required. Difficulty does not measure whether a program needs to be the way it is. High Difficulty can come about because of the skill level of the programmer, the poor structure of the program, or the absence of experience with a particular language. It can also possibly be a function of the complexity of the algorithm.

In order to test the validity of the Difficulty measure, it is reasonable to assume that productivity rates and defect levels bear some relationship to difficulty. A combination of program size and difficulty intuitively should correlate with either defects or productivity. Formulation of experiments is another matter, however. Productivity measurement has been done in a number of areas using lines of code as a unit of measure. What these studies have shown, more than anything else, is that productivity varies considerably among persons. Where defect tracking systems have been installed, some individuals have been observed to be more "error-prone" than others. In large organizations where such tracking systems are installed, there is also the practical matter of accepting a certain error rate in data collection systems to avoid the stigma of regimentation.

Table 3 Examples of code impurities and their effect on Difficulty

Impurity	*Example cases*	*Difficulty* N_2/η_2
I. Complementary operation	1. A=B×B+B−B	5/2
	2. A=B×B	3/2
II. Ambiguous operands	1. A=B+C;A=A×A	6/3
	2. D=B+C;A=D×D	6/4
III. Synonymous operands	1. A=B+C;D=B+C;E=A×D	9/5
	2. A=B+C;E=A×A	6/4
IV. Common subexpressions	1. A=(B+C)×(B+C)	5/3
	2. X=B+C;A=X×X	6/4
V. Unwarranted assignment	1. X=B+C;A=X^2	6/5
	2. A=$(B+C)^2$	4/4
VI. Unfactored expressions	1. A=B^2+2×B×C+C^2	8/4
	2. A=$(B+C)^2$	4/4

Until better proof can be obtained we can only test intuition against a decomposition of the various elements making up the Difficulty measure. As was discussed earlier, η_1 increases as the use of GO TO increases. It has already been shown that for PL/S and possibly for PL/I high η_1 values are caused by the absence of structured programming practices.

code impurities

With respect to N_2/η_2, six code "impurities" have been identified.[1] Table 3 lists these six impurities with examples. In all examples, both Case 1 and Case 2 yield the same computational result, although Case 2 yields a lower N_2/η_2 ratio.

I. *Complementary operations* are the same as unreduced expressions. Simplifying the expression (by reducing it) results in a lower N_2/η_2 ratio.

II. *Ambiguous Operands* involves the use of the same variable to mean different things. In Case 1, the variable A acquires two different values. If other instructions intervene between the two statements, one might lose sight of the current meaning of A. If A is referenced by other program statements in the interim, the reader must refresh his memory as to the current value of A to be certain of correct usage. A case in point is the common practice of using the same variable name to index all loops. On the surface this may save storage space, but it may also penalize readability.

III. *Synonymous operands* involves the assignment of the same value to more than one variable name.
IV. *Common subexpressions* are subexpressions that are used more than once in a program. If a variable is set to the value of the subexpression at the first occurrence, the variable can be used in all other occurrences.
V. *Unwarranted assignments* involves the assignment of a variable to a subexpression that is used only once in a program. The extreme example is assignment of a value to a variable where the variable is never referenced. Unwarranted assignments is not a classification that contradicts common subexpressions (IV).
VI. *Unfactored expressions* are easy to understand but sometimes hard to perceive in the midst of a coding effort. (Factoring is the same concept as used in mathematics.)

One may create his own examples of each of the impurities to test the readability of the result. In each case, the simplified expressions have the lowest N_2/η_2 ratio. We conclude from our studies and experience that programs that are coded as simply and concisely as possible have the lowest Difficulty (D). It is interesting to note that these impurities are also related to optimization in compiler development.

In summary, program difficulty can be measured with the Difficulty metric, and the reason for high difficulty can be quickly diagnosed. A high value for η_1 is most probably caused by unstructured programs. A high value for the ratio of N_2/η_2 also raises difficulty and is probably caused by an extreme use of one or more of the six impurities. The value of this metric is in its capability for measuring programs for potential error-proneness much earlier in the development cycle than at present. Self-measurement can be performed by the individual programmer at the occurrence of the first clean compile. This provides considerably more calendar time for corrective action than dependency on test results.

Practical application of metrics

Practical application of metrics depends on the availability of tools to measure programs. The tools should be easy to use, they should be fast, and they should provide more than the basic measurements. For example, a dictionary of operators and operands with frequency of use of each has been shown to be a powerful aid to diagnosing the reasons for high measured values. Given high Difficulty for a program, a high usage of GO TO is easily diagnosed. If variables have a high frequency of use, a dictionary quickly identifies those most frequently used.

Table 4 Summary of Language Level experiments

Language	*Language Level λ*	*Standard deviation*	*Number of programs*	*Cited references*
COBOL (for experienced programmers)	2.07	0.90	16	6
PL/S	2.05	1.14	643	4
BASIC	2.04	1.57	32	11
COBOL (for students)	1.82	0.73	23	6
PL/I	1.53	0.92	120	8
COBOL	1.40	0.69	24	9
ALGOL	1.21	0.74	14	1
FORTRAN	1.14	0.81	14	1
PILOT	0.92	0.43	14	1
System/360, System/370 assembler language	0.91	0.79	993	4
CDC 6500 assembler language	0.88	0.42	—	1
APL	0.81	0.60	29	9

For a large system of separately compiled program modules, a ranked list of modules by size or by difficulty can help narrow the search for modules deserving of concentrated attention. This type of ranking has been performed on two projects at the Santa Teresa Programming Laboratory. In each case, it was found that a small set of programs had a significantly higher Difficulty value than the population as a whole. The intuition of project members was found to confirm the high Difficulty values of the programs. As a result, the programs with high values of Difficulty were subjected to a more intensive review for possible problems. Although this experience is not proof of applicability, it does provide an example of practical use of programming metrics.

Other measures

Other measures which are extensions of the size and difficulty metrics have not been treated in this article for reasons of brevity. They are mentioned here, however, with a minimum of explanation so as to round out the concepts.

Table 5 Information Content for programs for Euclid's algorithm for finding the greatest common divisor

Language	*Information Content I*	*Cited reference*
PL/I	12.9	9,1
FORTRAN	10.5	9,1
CDC assembler	12.2	9,1
ALGOL 68	11.9	1
Table lookup	12.0	1
Potential HLL	11.6	1
BASIC	10.5	11
APL	10.0	12

A metric for *Effort* (E) to code a program should intuitively be a function of size (i.e., Volume V and Difficulty D), and may be expressed as follows:

Effort $= E = D \times V$.

There have been experiments to correlate Effort to defect levels and productivity, the results of which are encouraging enough to continue experimentation and refinement of the experimental method.

A measure of *Language Level* (λ) should be a constant number for a given language regardless of the algorithm being implemented. Language Level relates to Volume and Difficulty as follows:

Language Level $= \lambda = V/D^2$.

Results of experiments with Language Level are not completely understood. Many experiments have been conducted, and the results have been found to be variable. The means (i.e., averages) for large groups of programs seem to correlate with our intuitive belief, but within one language there is extreme variability. There is a suggestion that Language Level does not measure the language so much as it measures how the language is used in a program.

Table 4 is a summary of Language Level values, begun by Halstead[1] and updated with more recent research by Smith,[4] Zweben,[9] and Shen and Dunsmore.[6]

Information Content (I) of a program should be constant for a single algorithm, regardless of language chosen to implement it. The Information Content is also related to Volume and Difficulty, and is expressed as follows:

$$\text{Information Content} = I = \frac{V}{D}.$$

Experiments involving Information Content (I) are not extensive. If it proved to be a practical metric, Information Content would be, in a sense, a measure of the amount of function of a program. One small experiment is the implementation of Euclid's algorithm for finding the greatest common divisor using eight different programming languages. The results of the experiment shown in Table 5 indicate a narrow range of values for I.

Concluding remarks

Numeric measurement of programs, where measurements can be logically related to optimum approaches, has appeal from an engineering standpoint. Although software engineering has come a long way in the sense of establishing disciplines and orderly processes, the use of numbers to aid in understanding the reasons for those disciplines has not made the same progress.

Other engineering disciplines have constraints on design that can often be expressed numerically. The designer of circuit chips, for example, deals with technology limits such as the number of access pins, the number of circuits that can be housed in a chip, and so forth. These limits are in turn derived from other limits—heat dissipation, voltage limits, etc.—that can also be dealt with quantitatively. The limits are understood, and progress occurs when technology finds new ways to expand the limits. As we have discussed in this article, software science and its related metrics are beginning to quantify areas of programming that heretofore have been based on abstract feelings and experience.

There are still many areas of software science where validation and refinement are required. A large portion of today's programming effort deals with the modification of existing programs, whereas the metrics of software science deal with an entire program as an entity. As yet, no theoretical approach has been offered to measure modification work.

The authors conclude with a word of caution and encouragement related to rules for counting operators and operands. It is easy to become caught up in the desire to measure libraries of programs in a rush to see results. One should recognize that results may be spurious because of error in the measurement instrument. Those who are involved with programming measurement should learn from other disciplines, where strict and rigorous calibration of measurement instruments is a normal part of experimentation.

Software science offers a methodology not only for making measurements, but also for calibrating the measuring instruments.

CITED REFERENCES AND NOTES

1. M. H. Halstead, *Elements of Software Science*, Elsevier North Holland, Inc., New York (1977).
2. G. P. Fitsos, *Software Science Counting Rules and Tuning Methodology*, Technical Report TR 03.075, IBM Santa Teresa Laboratory, 555 Bailey Avenue, P.O. Box 50020, San Jose, CA 95150 (September 1979).
3. M. H. Halstead, "Software Science—A progress report," *Second Software Life Cycle Management Workshop*, Atlanta, GA, August 21–22, 1978, sponsored by the IEEE and the U.S. Army Computer Systems Command, 174–179, the Institute of Electrical and Electronics Engineers, Inc., 345 East 47th Street, New York, NY 10017.
4. C. P. Smith, *A Software Science Analysis of IBM Programming Products*, Technical Report TR 03.081, IBM Santa Teresa Laboratory, 555 Bailey Avenue, P.O. Box 50020, San Jose, CA 95150 (January 1980).
5. G. P. Fitsos, *Vocabulary Effects in Software Science*, Technical Report TR 03.082, IBM Santa Teresa Laboratory, 555 Bailey Avenue, P.O. Box 50020, San Jose, CA 95150 (January 1980).
6. V. Y. Shen and H. E. Dunsmore, *A Software Science Analysis of COBOL Programs*, Technical Report CSD-TR-348, Purdue University, Department of Computer Science, West Lafayette, IN 47907 (August 6, 1980). Also submitted to the *IEEE Transactions on Software Engineering*.
7. A. R. Feuer and E. B. Fowlkes, "Some results from an empirical study of computer software," *Fourth International Conference on Software Engineering, Proceedings*, Munich, Germany, September 17–19, 1980, pp. 351–355.
8. J. L. Elshoff, "Measuring commercial PL/I programs using Halstead's criteria," *ACM SIGPLAN Notices* **11**, No. 5, 38–46 (May 1976).
9. S. H. Zweben and Fung Kin-Chee, "Exploring software science relations in COBOL and APL," *Comsac 79, IEEE Proceedings*, Chicago, November 1979, pp. 702–709.
10. Unpublished work by S. D. Hartman, IBM General Systems Division, Menlo Park, CA 94025.
11. Unpublished measurements made by the authors of this paper.
12. Measurements made by the authors of this paper using the program on page 218, problem 2 in L. Gilman and A. J. Rose, *APL, An Interactive Approach*, Second Edition Revised Reprinting, John Wiley & Sons, Inc., New York (1976).

GENERAL REFERENCES

C. P. Smith, "A software science analysis of programming size," *ACM 80, Proceedings of the Annual Conference*, Nashville, TN, October 27–29, 1980, pp. 179–185.

G. P. Fitsos, "Vocabulary effects in software science," *Comsac 80, IEEE Proceedings*, Chicago, IL, October 31, 1980, pp. 751–756.

C. P. Smith, *The Application of Halstead's Software Science DIFFICULTY Measure to a Set of Programming Projects*, Technical Report TR 03.124, IBM Santa Teresa Laboratory, 555 Bailey Avenue, P.O. Box 50020, San Jose, CA 95150 (January 1981).

The authors are located at the IBM Santa Teresa Laboratory, 555 Bailey Avenue, P.O. Box 50020, San Jose, CA 95150.

Reprint Order No. G321–5154.