Information Processing & Management Vol. 20, No. 1–2, pp. 183–192, 1984
Printed in Great Britain.

0306–4573/84 $3.00 + .00
Pergamon Press Ltd.

SOFTWARE METRICS: AN OVERVIEW OF AN EVOLVING METHODOLOGY

H. E. DUNSMORE
Department of Computer Sciences, Purdue University, West Lafayette, IN 47907, U.S.A.

Abstract—A *direct, algorithmic software metric* is a rule for assigning a number or identifier to software. It is calculated algorithmically from the software alone. This paper describes (with examples) *lines of code, cyclomatic complexity* $v(G)$, *average nesting depth*, and the software science *length, effort*, and *time* metrics. Software metrics that are to be used for time, cost, or reliability estimates should be validated statistically via data analyses that take into consideration the application, size, implementation language, and programming techniques employed. Such research should concentrate on metrics useful for large programs. Those that work for some languages may work for similar languages as well, i.e., those in the same "family". Observing and measuring the work of professional programmers constructing software is probably the best means to conduct this type of analysis. We propose that this be a true joint effort, with early involvement between researchers and practitioners to establish what to measure and how to measure them. The critical need is for software metrics that can be calculated early in the software development cycle to estimate the time and cost involved in software construction.

"... when you can measure what you are speaking about, and express it in numbers, you know something about it; but when you cannot measure it, when you cannot express it in numbers, your knowledge is of a meager and unsatisfactory kind ...",

(Lord Kelvin, *Popular Lectures and Addresses*, 1889 [1]).

1. WHAT ARE SOFTWARE METRICS?

The most familiar software measure is the count of the number of "lines of code". From the first small programming tasks in the 1940's to the present mega-line projects, software has been characterized by the trio: "what does it do?"; (b) "what language is it written in?"; and (c) "how many lines of code is it?". Since (a) and (b) can only be described non-quantitatively, (c) has borne the burden through the years of being the predominant, countable, succinct software measure.

But, "lines of code" is such a gross measure that it may not be a very useful means of measuring software. For example, many of us might agree that constructing a 5000-line assembly language program is a more substantial chore than writing a 5000-line Fortran program. Others might believe (since one source language line is usually translated into several assembly lines) that 5000 assembly lines are easier to construct than 5000 Fortran lines. So, how many lines of assembly code *are* equivalent to 5000 Fortran lines? How about in Pascal and Cobol as well? And what is a "line of code" anyway? Is it a simple count of the number of lines in a program listing? Should it include comments and/or other types of non-executable statements (data declarations)? What about compound statements with two or more "lines" on one line?

In the preceding discussion we have purposely used the term "software measurement" instead of "software metric". In current parlance "software measurement" and "software metric" are generally used interchangeably. There are several ways of categorizing such software measures. A few possibilities are shown in Table 1.

There follows an unambiguous definition of a "software metric" that seems congruent with the predominant use of this term:

A *software metric* is a rule for assigning a number or identifier to software in order to characterize the software.

Table 1. Some ways of categorizing software measures

Product measure - pertains only to the software produced in the development or maintenance process (e.g., such measures as lines of code, typical execution time, number of predicates) [2]	*Process measure* - pertains to the characteristics of the development or maintenance process (e.g., team members' communication paths, software engineering techniques employed, development time) [2]
Direct measure - derived from the software code alone (e.g., lines of code, $v(G)$, number of variables)	*Indirect measure* - can be derived from any other information pertaining to the software (but not the software itself) (e.g., team size, implementation language, programmers' years of experience)
Algorithmic measure - derived via an algorithm that ensures that any arbitrary observer would obtain the same value (e.g., $v(G)$, number of procedures, total programmer months expended, ... if each is computed algorithmically)	*Subjective measure* - derived via some process that requires a subjective decision on the part of the observer that does *not* ensure that arbitrary observers obtain the same values (e.g., "modifiability" of a program on a scale from 0 to 10, manager's estimate of programmer time expended)

This definition clearly permits such measures as "lines of code" (and other direct, algorithmic measures) to be called software metrics. Furthermore, such an item as a "manager's estimate of maintainability" (which may be an indirect measure and almost certainly is a subjective measure) can also be called a software metric.

In the remainder of this paper we shall consider only direct, algorithmic software metrics:

> A *direct, algorithmic software metric* is a software metric that is calculated algorithmically from the software code *only*.

We believe that software metrics should be based on the software itself and should be objective measures. This allows for automatic calculation from the source code at minimum expense and ensures that arbitrary observers obtain the same values (when using the same algorithmic method).

Software metrics are often used to explain, estimate, or predict the amount of effort necessary to construct, comprehend, debug, or modify software. They may also be used for assessing the quality of software (i.e., reliability, ease-of-use, maintainability, and portability) and for allocating resources (i.e., time, personnel, and hardware).

We shall also establish the following definition:

> A *software metrician* is a person who develops software metrics and/or investigates the usefulness of them.

In this paper we shall consider some simple (but representative) direct, algorithmic software metrics. We discuss the computation of and give examples for lines of code, cyclomatic complexity $v(G)$, average nesting depth, and the Software Science length, effort, and time metrics. We then proceed to the problem of establishing empirical support to suggest that software metrics really "characterize" software. We conclude with an assessment of the future of software metrics research.

2. SOME SIMPLE SOFTWARE METRICS

This section examines a few direct, algorithmic software metrics. In addition, it presents a computation algorithm and an example for each. As an ongoing illustration, consider the Fortran subroutine in Fig. 1. This routine sorts an integer array *X* of size *N* (no greater than 100) into ascending order by using the interchange sort technique. It was written by the author in under 10 minutes. This subroutine will be used to illustrate the metrics defined in the remainder of this section.

2.1 *Lines of code*

As we mentioned at the outset, the most familiar software measure is the count of the number of "lines of code" (also known as LOC or, for a thousand lines of code, KLOC)[3]. Although on the surface this may seem a very simple direct, algorithmic software metric, there is no general agreement about what constitutes a "line of code". If it is just a count of the number of lines, then Fig. 1 contains 15 lines of code. But, most software metricians agree that the LOC count should not include comments or certain other types of non-executable statements because these are really internal documentation and not "purely" lines of the program. The only executable statements in Fig. 1 are in lines 4–14 leading to an LOC count of 11.

What about languages that allow compounding with two or more statements on one line or languages that allow a single statement to be extended over two or more lines? For example, in Fig. 2 (containing a bit of a program in an imaginary language) we may argue that there are alternatively 3 lines of code (denoted by the 3 semicolons), 5 (simply counting every line), or even 6 (every line + 2 for line 1).

If LOC is to be computed in order to estimate effort, we believe that an appropriate definition is that:

> A *line of code* (*LOC*) is any line of program text that is not a comment or blank line, regardless of the number of statements or fragments of statements on the line. This specifically includes all lines containing program headers, declarations, executable and non-executable statements.

```
Line                                                        Level
----                                                        -----
 1        SUBROUTINE SORT (X,N)
 2        INTEGER X(100),N,I,J,SAVE,IM1
 3C  THIS ROUTINE SORTS ARRAY X INTO ASCENDING ORDER
 4        IF(N.LT.2) GO TO 200                                 1
 5            DO 210 I=2,N                                     2
 6                IM1=I-1                                      3
 7                DO 220 J=1,IM1                               3
 8                    IF(X(I).GE.X(J)) GO TO 220               4
 9                        SAVE=X(I)                            5
10                        X(I)=X(J)                            5
11                        X(J)=SAVE                            5
12   220          CONTINUE                                     3
13   210      CONTINUE                                         2
14   200  RETURN                                               1
15        END
```

Fig. 1. A Fortran subroutine that sorts an array into ascending order.

```
Line
----
 1      X <— 0;  Y <—1;
 2      IF X<Y
 3        THEN
 4          R <— X/Y
 5        FI;
```

Fig. 2. A program segment with an unknown number of lines of code.

This definition is congruent with the fact that effort must be expended on all source statements (including declarations), not just those that will be executed. Using this definition, the program in Fig. 1 has only one line (line 3—a comment) that should not be counted; this yields 14 lines of code. Figure 2 has 5 lines of code.

It should be obvious, then, that counting lines of code is not the simple, exacting, exercise that one might imagine. On the other hand, if one uses a standard definition (like the one above) consistently, then we believe that LOC is probably the most intuitive measure of program size.

2.2 *Cyclomatic complexity* ($v(G)$)

If the static control-flow graph of a program is constructed, then there are graph-theoretic metrics that can be computed. Figure 3 is the control-flow graph for the program in Fig. 1. Note that the nodes have been labelled with line numbers that refer back to Fig. 1.

The best known graph-theoretic metric is McCabe's "cyclomatic complexity" $v(G)$[4]. This measure is a count of the number of distinct paths in the program. It is computed by the formula

$$v(G) = e - n + 2$$

where e is the number of edges (i.e. potential control paths) and n is the number of nodes (i.e. individual statements or collection of sequential statements). In Fig. 3, $e = 15$ and $n = 12$; thus $v(G) = 5$. The metric $v(G)$ is even simpler to compute than it appears on the

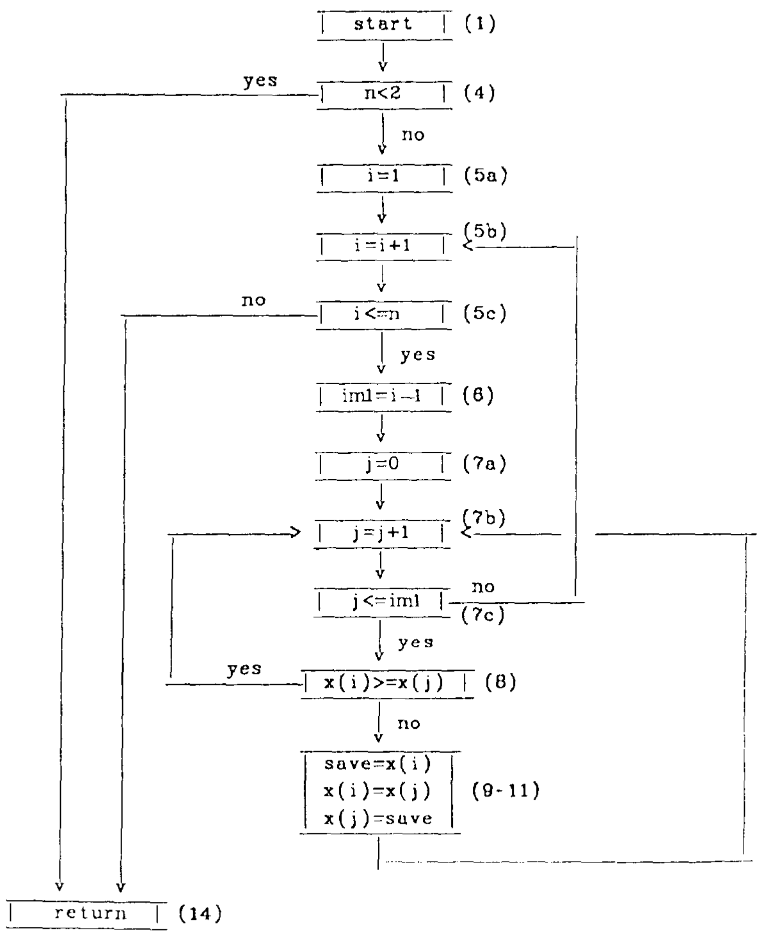

Fig. 3. The control-flow graph for the program in Fig. 1.

surface. Since control can only be interrupted by loops or transfers and since these all depend on the truth or falsehood of a predicate, one need only count the number of predicates in the program π and use the simpler formula

$$v(G) = \pi + 1.$$

In Fig. 3 note that there are explicit or implicit predicates on lines 4, 5, 7, and 8. Thus, $\pi = 4$ and, as before, $v(G) = 5$.

2.3 *Average nesting depth*

Another simple measure of the control structure complexity of a program is "average nesting depth"[5]. In order to compute this metric, every executable statement in a program must be assigned a nesting level. A simple recursive procedure for doing this is described as follows:

(1) The first executable statement is assigned nesting level 1.

(2) If statement a is at level l and statement b simply follows sequentially the execution of statement a, then the nesting level of statement b is l also.

(3) If statement a is at level l and statement b is in the range of a loop or a conditional transfer governed by statement a, then the nesting level of statement b is $l + 1$.

Notice that all executable statements in Fig. 1 have been assigned a nesting level via this procedure. In order to determine the average nesting depth, sum all statement nesting levels and divide by the number of statements. For our program (with 11 executable statements) the sum is 34 and the average nesting depth is 3.1.

2.4 *Software science length, effort and time metrics*

All software consists of ordered collections of tokens. Each token is an atomic element of the coded algorithm. For example, line 4 in Fig. 1 contains the seven tokens IF, (), N, .LT., 2, GO TO, and 200. Notice that "()" is considered one token because "("is always paired with")". In the same fashion "GO TO" is a single token.

Halstead[6] proposed several software metrics that depend entirely on the tokens in a program. In his "Software Science", tokens are dichotomized into "operators" and "operands" where an "operand" is a variable, constant, name, or label and an "operator" is anything else. The following terms are defined in software science:

$$\eta_1 = \textit{number of unique operators}$$

$$\eta_2 = \textit{number of unique operands}$$

$$N_1 = \textit{total occurrences of operators}$$

$$N_2 = \textit{total occurrences of operands.}$$

Figure 4 shows a sample software science analysis of subroutine SORT. Unfortunately, there is no general agreement among metricians on how to count specific operators and operands[7]. Several things done in Fig. 4 would raise objections from software science purists. But, it is meant only as a simple, illustrative example.

It is only coincidence that there are $\eta_1 = 13$ different operators and $\eta_2 = 13$ different operands in Fig. 4. For most programs, the number of unique operators and unique operands are not equal. (In fact, for very large programs $\eta_2 \gg \eta_1$). In this case (i.e. Fig. 4), $N_1 = 38$ and $N_2 = 42$.

The total number of unique tokens (called the Vocabulary η in software science) is defined by

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

Operators	occurrences	Operands	occurrences
SUBROUTINE	1	SORT	1
()	10	X	8
,	8	N	4
INTEGER	1	100	1
IF	2	I	6
.LT.	1	J	5
GO TO	2	SAVE	3
DO	2	IM1	3
=	6	2	2
-	1	200	2
.GE.	1	210	2
CONTINUE	2	1	2
RETURN	1	220	3
	38		42

Fig. 4. A software science analysis of subroutine SORT.

The program length N (i.e. total number of tokens used) is

$$N = N_1 + N_2.$$

For our program, $\eta = 26$ and $N = 80$. The software science metric N may be convertible to LOC via the relationship LOC $= N/c$ where the constant c is language-dependent. For Fortran it is suggested to be about 7; notice that $80/7 \approx 11$ which is fairly close to the actual LOC in Fig. 1.

There is an effort metric E defined in software science as

$$E = N \log_2 \eta \times \frac{\eta_1}{2} \times \frac{N_2}{\eta_2}.$$

The units in which the effort metric is expressed are "elementary mental discriminations" (emd's). Since software science assumes that the typical programmer can perform 18 emd's/second[6] (i.e. 1080 emd's/minute), the effort metric can be converted to time T via

$$T\,(\text{minutes}) = \frac{E}{1080}.$$

For our program, $E \approx 7897$ and $T \approx 7$ minutes. Since the author created this routine in under 10 minutes, the T metric is a fairly-good time estimate in this case.

3. HOW CAN EMPIRICAL DATA SUPPORT THE USE OF SOFTWARE METRICS?

There is no established criterion for "validating" a software metric. Obviously, most of the measures defined in the last section may be computed on any program. Ideally, we would like each metric to be algorithmically-convertible into time, cost, and reliability estimates that coincide closely with actual times, costs, and reliabilities. For example, if a time measure based on LOC estimated 400 hours for development of a program which actually took 410 hours, then we would be encouraged that the metric is useful. But, what if it actually took 500 hours? Is 400 *good enough* as an estimate?

Furthermore, validation cannot be performed by arbitrarily choosing one (or even a few) good examples. For instance, the time predictor might estimate within 10% of the actual time for 8 projects chosen at random, but it might be in error by 50% on another and 200% on yet another. Could you trust such an estimator? There is no simple answer to this question. From a practical standpoint, the only solution is to validate metrics *statistically*. This can be done by collecting data on several software projects. The estimated values (i.e. time, cost, size, etc.) are then compared to the actual values. If the correlation is significantly-different from zero and if the estimations are generally close to their actual counterparts (e.g. average relative error no greater than 10%), then a convincing argument can be made that the metric is a valid one.

But, what are appropriate software projects for such an investigation? Software can differ along several dimensions—including size, implementation language and application. The software metrician must be concerned with the problems of "external validity"[8]. That is, if a metric is validated for a particular application, language, and program size, is that result generalizable to any other programs? In particular, are metrics that work for 50-line programs useful for 500,000-line programs? Do they generalize across different languages? To what extent do the peculiarities of the application affect the relationship between metric estimates and actual values?

There is a problem of scale with many software metrics. They are often created from observing small programs, and their usefulness is often supported by data from only small (e.g. single-module) programs. There are serious questions as to whether any explanatory or predictive capability exists for them when applied to large, multi-module programs. For large-scale software, measures that emphasize the interconnectivity among modules (e.g. Kafura and Henry's information flow metrics[9]) may be more useful.

Since it is infeasible to validate a software metric for every possible size, language, and application, some sensible compromises must be made. First, we believe that results based on "toy" programs (of less than 100 lines) are not generally representative. But, since even very large systems are composed of separate modules, we may be able to defend the generalizability of results based on realistic modules on the order of about 100 lines of code each.

We still do not know enough about programming languages to be able to rule definitively on the relationships among them. Some languages are sufficiently similar in data structures, control structures, and syntax that they may be considered members of the same "family" of languages. Assembly language is probably in a family of its own. Fortran, Pascal, Algol, and perhaps even Cobol may be considered members of another family (but not the only ones in this family and some people would not even put these four in the same family). It is not unreasonable to suggest that a software metric found useful (for example) for Pascal programs would also be useful for Fortran programs. We propose that metrics validated with one language will be useful with others in the same family, unless it can be shown empirically that the use of metrics does *not* generalize among family members.

Furthermore, we think that software metricians should try to gather data from *any* realistic applications. We propose that data from all programs should be considered together during the present research, unless empirical results suggest that some applications are fundamentally-different from others. Perhaps later we will be sophisticated enough to take into account characteristics of the application, but we consider this a minor factor now when compared with the more difficult problems concerning size and language. We will have more to say about this in the next section.

4. WHAT IS THE FUTURE OF SOFTWARE METRICS?

It should be evident from the preceding section that software metricians need to work diligently to develop some basic relationships concerning generalizability across sizes of programs, types of applications, and languages. Our intuition is that some candidate software metrics that seem unrelated to the size and cost of small programs may yet be useful for large software projects. Small programs are a special case in software development and maintenance. Generally, they are produced or altered in the most expedient manner (i.e. "quick and dirty") without regard for standard software practices, and thus cost and effort may not be as predictable an activity as with larger programs. Rather than spending a good deal of time confirming this hypothesis, we suggest that software metricians should concentrate on metrics useful for large programs, since these are the ones for which better size and cost estimators are actually needed.

It seems reasonable that the application involved and the implementation language should be related to size and effort. We suggest that software metricians develop and validate their metrics on representative software written in the languages most appropriate for each application. Then, as replications are attempted for other software and other languages, we will be able to find what role these factors really play. If the metrics in question perform well for various applications and various languages, then these factors may be disregarded in employing them. But, if not, then we will need to modify existing software metrics so that they will be able to take into account characteristics of the problem and implementation language as well.

Perhaps the method for gathering empirical data that will best lead to external validity is to enlist the aid of the industrial world in software metrics research. It seems that organizations with large-scale software projects should have a two-handed stake in this type of research. First, they can provide us with data from actual software construction projects and maintenance tasks that can be used for metric development and validation. Second, they will be the primary beneficiaries of this research. If metrics can be developed that are good estimators of cost and effort, then these will be valuable tools for their own programming managers and team leaders.

Our experience conducting metrics research at Purdue University has suggested that, for real progress to be made, there must be a true joint effort between software metricians and practitioners. Working alone, the researcher may have a wealth of ideas, but little data upon which to test them. But, if the metrician simply asks for any data that was collected during an industry software project, then the quality and usefulness of that data may be very questionable.

For example, suppose that a metrician is offered a collection of data consisting of the following items from a major software development project for each of 45 programs that comprised a large system: (1) name of the program; (2) short description; (3) implementation language; (4) lines of code; (5) development hours (i.e. development time). Does "lines of code" include all source lines or only executable statements? Was it actually counted during development or estimated in some manner? Do "development hours" include all work-hours (even those only marginally-related to project development, e.g. coffee breaks)? Was "development hours" actually collected during development or estimated later?

Obviously, the software metrician would benefit far more from early involvement with the project. He could request that time measures be collected objectively during development. He could ask for source copies of the 45 programs from which several metrics (in addition to lines of code) could be computed algorithmically.

From a research standpoint, it is very interesting to find metrics that may be computed from completed software and which yield a time or cost estimate that (statistically) agrees with the actual time or cost to produce the programs. But, in an economic sense, the use of software metrics in this manner is not very rewarding. The programmer, analyst, or manager faced with a software development or maintenance task wants to know *before* the software is completed how much time or cost will be involved. It is of limited interest to find out *after the fact* that the time or cost required is explainable by a metric.

Thus, the critical need is for software metricians to apply themselves to developing metrics that can be used early in the software development cycle. Ideally, we would like to have a metric that could be applied to a functional specification that could be used to accurately estimate the time and cost of development of the program. Note that this will also require a software metric that cannot be calculated from the code alone (i.e. an "indirect" metric). The type of metric desired could be calculated from a "program specification language"-type representation of the program.

Another factor that may need to be considered for some metrics is "programming technique", particularly if a measure is to be used for estimation purposes. When a programmer develops software using a top-down process, the programs may be fundamentally different from those produced using a bottom-up process. Furthermore, some programmers are required to document all data structures (including specifying all interconnections with other modules) before writing code for any module. Their programs (and programming process) may well be quite different from those programmers who construct module code before the headers for all modules are constructed.

It may be that the specification stage is too early for useful prediction, since typically at this point one has developed neither data structures nor algorithms. Perhaps a little further along in the development process there may be more adequate information (e.g. fairly-complete data structures and some algorithm development) that may permit a metric to be applied which can be used to predict the time and cost remaining. This approach is being explored in the Department of Computer Sciences at Purdue University[10]. An early-estimation metric (or metrics) could allow the programmer/analyst/manager to determine if the proposed software is worth the expense involved or, if so, if it can be delivered on time.

5. WHAT ARE SOME SUGGESTIONS FOR ESTABLISHING USEFUL SOFTWARE METRICS?

Table 2 contains the suggestions proposed in this paper for establishing useful software metrics. Direct, algorithmic software metrics are rules for assigning numbers or identifiers to software. These are calculated algorithmically from the software code only. Proponents argue that software metrics may be used to explain or predict such software characteristics as size, time, and quality. There is no agreed-upon criterion for validating such a metric, but we have proposed that some type of statistical, empirical evidence is mandatory. Preferably, this evidence should be collected from realistic applications considering (and perhaps categorizing by) size, type of application, language, and even programming technique. We suggest that data should be collected from large projects with one or more modules—each on the order of about 100 lines of code. Anything less falls into the "toy" program category, from which generalization may be dangerous. We further contend that metrics found useful for a program written in a language in one "family" should be considered for use with programs written in other languages in the same family. Furthermore, software metricians should try to generalize across applications if possible.

We suggest that software metricians involve themselves early in the data collection process to ensure that as much data as possible is collected algorithmically and that the computation of metrics is consistent with accepted definitions. The suggestions concerning joint software metrics and practitioner research are written from the perspective of a person in academia. The software metrician in industry obviously has even better contacts for influencing, setting up, and conducting such industrial data gathering processes. Finally, we believe that software metrics will be taken much more seriously when they can be used for estimation purposes—to *predict* the time and cost of software projects.

Table 2. Suggestions for establishing useful software metrics

How should software metrics be computed?	algorithmic and direct (from the software itself)
What applications should be observed in software metrics research?	realistic ones (considering size, language, and programming technique that would be typical of actual software of the same type); results should be generalized across applications in present-day research
What size projects should be observed?	large size (several modules with each on the order of at least 100 lines of code)
What languages should be used in these projects?	should be typical of those used in similar "real" applications; metrics found useful with one language may generalize to other languages in its "family"
How much software metrician involvement should there be in designing for and collecting software metrics data for analysis?	metricians should be involved early in and throughout the design phase to ensure unbiased, objective, detailed, complete data is gathered for analysis
To what use should we put software metrics?	to explain or predict time, cost, quality, etc.

Acknowledgements—This research is supported by the U.S. Army Research Office (ARO) through the Army Institute for Research in Management Information and Computer Systems (AIRMICS). The author is indebted to colleagues Vincent Shen, Sam Conte, Andrew Wang, Steve Thebaut, Prudence Zacarias, Carl Burch, Jim Yu (all from the Department of Computer Sciences, Purdue University), Richard Schweikert (Department of Psychology, Purdue University), and LTC J. Steve Davis (AIRMICS) for many discussions that have helped to shape the opinions shared within. A special word of thanks to Thebaut for his careful reading and many excellent suggestions concerning an earlier draft of this paper.

REFERENCES

[1] M. L. Cook, Software metrics: an introduction and annotated bibliography. *ACM SIGSOFT Software Engng Notes* 1982, **7**(2), 41–60.

[2] V. R. Basili and R. W. Reiter, Evaluating automatable measures of software development. *Models and Metrics for Software Management and Engineering* (Edited by V. R. Basili), pp. 93–105. Computer Society Press, Los Alamitos, California (1980).

[3] T. C. Jones, Measuring programming quality and productivity. *IBM Systems J.* 1978, **17**(1), 39–63.

[4] T. J. McCabe, A complexity measure. *IEEE Trans. Software Engng* 1976, **2**(4), 308–320.

[5] H. E. Dunsmore and J. D. Gannon, Analysis of the effects of programming factors on programming effort. *J. Systems Software* 1980, **1**(2), 141–153; also in *Models and Metrics for Software Management and Engineering* (Edited by V. R. Basili), pp. 280–289. Computer Society Press, Los Alamitos, California (1980).

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

[7] S. D. Conte, H. E. Dunsmore and V. Y. Shen, Software science revisited. *IEEE Trans. on Software Engng* 1983, **SE-9**(2), 155–165.

[8] D. T. Campbell and J. C. Stanley, *Experimental and Quasi-Experimental Designs for Research*. Rand McNally, New York (1963).

[9] D. G. Kafura and S. H. Henry, On the relationships among three software metrics. *Proc. ACM Sigmetrics Symp. on Measurement and Evaluation of Software Quality* (1981).

[10] A. S. Wang and H. E. Dunsmore, Back-to-front programming effort prediction. *Proc. Symp. on Empirical Foundations of Information and Software Science*, Atlanta (November 1982).