Seven ± Two Software Measurement Conundrums

Bill Curtis

TeraQuest Metrics, Austin, Texas
Software Engineering Institute

Anita Carleton

Software Engineering Institute
Carnegie Mellon University

Abstract

Since computer science has traditionally considered itself more an application of discrete logic than of continuous functions, measurement has not played as important a role in software as it has in other areas of engineering. Since almost no computer science programs teach courses in measurement theory, statistics, experimental design, or related areas, software engineering has not benefited from rigorous development as a quantitative science. This article discusses several current problems facing the quantitative study of software engineering that present challenges to the software measurement community.

Introduction

When the senior author first started doing research in software measurement, he was often greeted by computer science colleagues with, "Software metrics...oh yeah, that's what you do if you're a lousy programmer!" During the late 1970s and early 1980s his research in software measurement was mostly confined to the backwaters of computer science. In the mid-1980s, he switched from taking quantitative to qualitative measures of software phenomena, such as interviews and cognitive protocol analyses. As his measurements became less precise, his results became increasingly popular. This is the kind of outcome that does wonders for tenure, but little for one's self-respect as a scientist.

In trying to help organizations establish a sound quantitative basis for measuring software development and maintenance, we continue to struggle with recurrent problems in software measurement. Resolving these problems will require new paradigms in different areas of software measurement and a different way of looking at the measurement endeavor.

These recurrent problems seem to revolve around the simplistic application of measurement techniques to software problems. It is too easy to make the unfortunate assumption that one size fits all. This assumption is as painful in software measurement as it is in fitting shoes. In this article we will briefly discuss the consequences of assuming that:

- one paradigm fits all measures
- one equation fits all validations
- one statistic fits all populations
- one result fits all samples
- one measure fits all programs
- one definition fits all uses
- complexity is simple

One Paradigm Fits All Measures

Several of the most popular approaches to software measurement share a common underlying structure, even through they were proposed for entirely different purposes. Some popular measurement paradigms for measuring size and estimating effort both share a common underlying framework. For instance, consider COCOMO and function points. COCOMO (Boehm, 1981) uses an adjusted size measure to predict the effort required to build a system. Function points (Albrecht, 1979,1984: Garmus, 1989) provides a measure that avoids some of the often criticized problems of lines of code by counting weighted functional elements of a program such as internal logical files and external inquiries.

In COCOMO the estimated size of a system is multiplied by a constant and raised it to an exponent. In function points the functional elements in the program are weighted and summed. Having developed these basic calculations, neither measure seems comfortable that it will relate very well with the criterion of interest. The equations for both COCOMO and function points take their core calculation and tweak the dickens out of it with every factor that might remotely affect its relationship with an intended criterion. COCOMO uses 15 factors (product, computer, personnel, and project attributes), while function point calculations are more parsimonious using only 14 factors (general system characteristics). These adjustment factors are then aggregated and applied

arithmetically to the core equation. Both measures have the form:

Core calculation
 Aggregated adjustment factors
= Estimate of criterion

COCOMO is thought of as an effort estimator for costing projects and function points is thought of as a measure of software size that estimates the business functionality provided in the program. However, structurally these are both very similar measurement paradigms. Both count up and weight attributes of the program and then further weight them with attributes of the environment surrounding the program. Both equations admit implicitly that their underlying relationships are heavily affected by their context. In software, there are few if any pure measures. Most good measures must be adjusted by unique environmental influences. For this reason most cost estimators report that they must calibrate standard cost estimating models such as COCOMO, SLIM, or Checkpoint to the particular characteristics of the organization in which they are using the model.

Occasionally, COCOMO is mistakenly described as a regression-based cost estimating model. The equation underlying COCOMO has no relation to statistical regression analysis. In fact the methods used for weighting and calculating adjustment factors in both COCOMO and function points have no basis in any statistically-based predictive model. Boehm (1981) to his credit admits this. Although the originators of both equations developed their weighting schemes through evaluating relationships in their data, both equations are stated in terms that do not allow statistically based adjustments to the equations based on conducting validity studies in other samples. Thus, some of the most widely used software measurement paradigms encourage us to continue tweaking the data rather than embracing more mathematically tractable alternatives. If we publicly engaged in regression analysis, the adjustment factors would probably drop from the 14 or more used in most models to less than 5 (Vosburgh et al., 1984).

One Equation Fits All Validations

Software measurement is loaded with marvelously intricate equations for estimating and representing all manner of phenomena. The most elaborately posed were the series of equations that formed Halstead's (1977) Software Science. Equations such as those for Software Science, function points, and module connectivity (Henry and Kafura, 1981) aggregate and manipulate numerous elements of a computer program, be they tokens, lines, data, files, or inquiries.

Although it is good to see that software metricians have not allowed any possible element of a program to escape their attention, it is not parsimonious to force feed all of them into an equation. The entire basis for regression-based statistical methods is to take a large set of predictors and reduce them to a smaller set that adequately accounts for the relationships observed in the sample.

For instance, Curtis, et al. (1979a,b) found that $\eta 2$ accounted for as much variation in performance as did the many elaborate software science equations of which it was a component. Similarly, Kitchenham (1993) found that a smaller set of the components comprising function points accounted for the relationships obtained with the fully calculated equation. In the face of such results one has to question the value of overly elaborate software equations. While clever and elaborate theories are written to justify the equations, a much simpler model seems to account adequately for the observed relationships.

In most other fields where quantitative methodology is required as part of one's academic preparation, it is ordinary practice to use statistical techniques such as regression analysis to reduce the number of variables or elements that must be measured to estimate a criterion of interest. Without this empirical discipline, software measurement has substituted the elaborate construction of theory for the quantitative evaluation of relationships. If not needed empirically, at least these intricate equations provide stimulating debates at software conferences.

One Statistic Fits All Populations

Most software metric studies have been performed on single, or at best limited, samples of programs. Rarely have studies been performed across large samples of programs, except when analyzing project level data (total size and effort, etc.) for cost estimating studies. Consequently, the software measurement community has no idea how to describe the population of programs in the world, or even how to accurately characterize the major sub-populations.

Several large multi-company databases of software measures have been developed. These include the DACS database collected by RADC and the databases collected by Capers Jones, Howard Rubin, and Larry Putnam. However, these databases have seen limited use by the research community. For instance, the RADC database suffers from inconsistent definitions of data items, making it impossible to compare results across organizations.

Data from the repositories such as those collected by Jones and Rubin have not been reported in ways that allow ordinary population statistics to be determined. First, only means are usually reported from these data. Other distributional statistics such as standard deviations or skewness are rarely reported. Second, these programs are drawn from a self-selected sampling of organizations that reported data to the commercial services of Jones, Rubin, or Putnam, and it impossible to accurately determine the characteristics of the broader population. In particular, the Jones and Rubin data most often come from traditional MIS applications, and Putnam's data most often comes from aerospace, although other types of systems are occasionally sampled in each of these databases.

Claiming that statistics from such databases represent national or industry trends stretches well beyond what may be legitimately claimed from these data. Without better quantitative characterizations of population and sub-populations of computer programs in the world, it is impossible to know how far results from a sample may be generalized. The field is left with making sweeping statements from minimal samples that would not be considered adequate in other areas of science. As Mark Twain marveled about science from the kind of statements we make in software, "where else can one get such a marvelous return in conjecture from such a modest investment of fact?"

Nevertheless, the uses made of these commercial databases have been beneficial in establishing the value that could be attained from having more accurate population data in software engineering. There is a strong demand for benchmarking data, and the software measurement research community has not responded to this need. In fact the research community has largely ignored its own need for population data to determine the limits for generalizing results.

One Result Fits All Samples

Over the past several years we have reviewed numerous software measurement papers that presented inconsistent results calculated on different programs. For instance, consider the following simulated results that approximate those from several studies we have read where a software metric was being correlated with a performance measure across modules in several different large programs:

Sample	Correlation
Program 1	high
Program 2	modest
Program 3	insignificant

The interpretations given such results vary with the predisposition of the authors reporting the data. Most would say something like, "In general the metric is related to performance."

Huh??? With results that differ across samples or programs, there is no 'in general'. Sometimes the measure is related to the criterion and sometimes it isn't. And we sure wish we understood why. This is the one question the authors don't answer too often. Unfortunately, referees don't always require that it be answered either.

There are several possible causes of the differing correlations simulated above. Among them are that:

1) the distributions of data are non-normal and restricted the possible range of the correlations,
2) the relationships only hold under certain conditions and these conditions vary across the programs,
3) there are certain characteristics of some programs that mask the relationship between the variables that would have otherwise appeared in the correlations,
4) individual differences variation among the programmers working on the program has overpowered the variation due to the metric in determining the value of criterion, and
5) these particular measures are only appropriate for detecting the hypothesized relationships in some programs, and different measures or operational definitions would be needed to observe the relationships in the program studied.

The tragedy in walking away from such results without deeper investigation of the variance in the correlations is that scientific breakthroughs often occur when equivocal or anomalous results surprise the researcher and cause deeper investigation of the data. Great science starts when researchers ask why the results weren't straightforward as hypothesized. Science is interesting because the world presents us with so much more than just main effects in the data. There are myriad interaction effects that confound the most carefully plotted theory. Tracking down the causes of these interactions is the essence of making original contributions to science, and of occasionally redefining the explanatory paradigm.

For instance, Curtis et al. (1979b) dug into the data underlying zero-order correlations between software metrics and performance criteria and found both program and individual differences masking the expected results. Although their adjustments to the data were open to serious scrutiny, they later cross-validated their enhanced results in an independent study (Curtis, et al., 1979a)

where they had more effectively controlled for such differences. Although they supported their hypothesis that software metrics were related to performance criteria, they became quite wary of the impact that individual differences among programmers could have on software data.

The assumption underlying too many software measurement studies is that all samples or programs are created equal in terms of the phenomena underlying their development and structure. We know from practical experience that this assumption is not true. Metric studies often do not account for process differences in the development of large programs that may have substantial impacts on the relationships observed. The explanation is simple. A criterion has many factors affecting its value, only some of which are captured by the software measure under investigation. As these factors vary across samples and programs, the observed relationship between the metric and the criterion will vary.

Our fundamental software measurement paradigms have not changed in over a decade because we are not digging into the data with inquisitive minds. No variation in results across programs or samples should go uninvestigated. It is unfortunate that tenure is more a matter of publication than inquisitiveness.

One Measure Fits All Programs

In the beginning programs were all measured by their size in lines or instructions. This is not surprising since lines are easy to count—or so we thought. Most early cost estimating equations were based on a size measure, generally lines of code. Jones (1978) demonstrated an unsettling paradox in the use of line of code measures when comparing results across different programming languages. Others (Vosburgh, et al., 1984) have demonstrated excellent predictive results even when reporting line of code measures across several languages. Nevertheless, Jones' paradox presents serious challenges to interpreting lines of code results when used across languages that differ strongly in the computational paradigm they implement.

As a solution to this software size measurement paradox, Albrecht (1979, 1984) presented a measure of software size that was based on the functional characteristics of the program. These functional characteristics included internal logical files, external interface files, external inputs, external outputs, and external inquiries. These are sensible functional program elements for transaction processing systems.

In their zeal to eradicate the lines of code paradox, many pundits have recommended that all software be measured with function points. This recommendation has been resisted in many application domains because those building sophisticated scientific, engineering, system, or embedded software applications did not believe that these were the functional elements that presented the most challenge in their development. The much maligned lines of code measure continues to be the most frequently used indicator of size in these non-transaction processing applications. Jones (1992) proposed a measure called feature points to use outside of transaction processing, but it has not been developed sufficiently for evaluation and use.

In the early 1980s Albrecht and Gaffney (1983) demonstrated that function points correlated very highly with lines of code. This relationship was reassuring for those who used function points because they could be counted before lines of code became available. However, it was disquieting to those who believed that function points measured something theoretically different from lines of code and something that lines of code failed to indicate. Function points is therefore one among several useful indicators of size, and one highly correlated with other indicators.

All of these measures, (i.e., function points, lines of code, tokens, or decision structures) are indicators of a primary underlying construct, program size. Regardless of which program constructs are measured, the primary attribute they indicate is size. Basili, et al. (1983) could not find a size measure that proved better than lines of code for predicting performance criteria in scientific programs.

Function points is a useful size measure for transaction processing programs. However it has not been defined using elements that are relevant to many areas of software development beyond the transaction processing paradigm. Function points may not solve some important measurement paradoxes that emerge when size measures are compared across different application domains, even for programs written in the same language.

Function points is one instance of a measure from the domain of functional software measures. There are other functional measures, but they must be defined in terms of the primary functional elements that constitute a specific application domain. If there are important functional elements, or attributes of elements that account for a significant portion of the variance in some development criterion that are not assessed in counting function points, then function points is not an adequate size measure for a given domain of application. The success of the functional measurement paradigm revolves

around a successful application domain analysis as a critical input to defining new functional measures.

If a true functional measurement paradigm can be applied in software development, then it will end the notion that there is a single size measure that can be used for assessing all programs. This would be a radical departure from the dominant paradigm in software measurement, which led us to spend the last two decades trying to define the optimum software size measure. The new paradigm would follow existing trends in computer science research that gave up on trying to define architectural principles that could be applied across all programs.

In this new functional software measurement paradigm, different application areas will have uniquely defined functional measures tailored for use with the primary functional elements of that domain. For instance, in test software the test vectors will be among the program entities counted as functional elements. However, simple counts of vectors may have to be adjusted to account for important attributes that vary among classes of vectors. Just as the study of software systems has moved to investigating domain-specific software architectures, so the software measurement paradigm should move to the creation of domain-specific software measures.

If a functional software measurement paradigm is adopted, we will probably produce domain specific measurement results from our research for the next decade. This may help us better limit the appropriate generalization of our results. While this will not satisfy those who search for a grand field theory of software measurement, it will probably produce more useful research results and more practical software measures. Hopefully we can see many areas of software benefit from a new developments in software measurement the way transaction processing applications have benefited from function points.

One Definition Fits All Uses

Related to the problem of applying a specific size measure to all manner of programs, is the problem of assuming that a single operational definition of any measure is adequate for all uses of that measure. For instance, when measuring lines of code people have argued vociferously about whether the count should include only executable statements, or also include data declarations, compiler directives, comments, etc. Further, they argue about whether reused lines should be counted or not.

This argument takes a different twist depending on how the measure will be used. For instance, if a manager wants to estimate the amount of time that will be required to build a system, then reused code should not be included as a straightforward count of lines. Rather, this code should be weighted by a fraction so that its inclusion only causes the effort to go up by the amount required to integrate the reused code. On the other hand, if a manager wants to account for the productivity gain due to reused code, then this code should be added in without weighting to see how much less effort in involved by being able to reuse previous components.

Therefore, the operational definition (Curtis, 1980) of a software measure depends on its use. This observation is behind the Goal-Question-Metric paradigm presented by Basili and Weiss (1984). Jones (1986) demonstrated that differences in the operational definition used in counting lines of code can cause the result to vary by an order of magnitude.

A major problem faced in using historical data, even when collected from projects conducted at a single site, is that few people fully describe the operational definitions they used in collecting data. Thus, variations resulting from different counting rules introduce error into research results or predictions based on these data. The software community needs some standard methods to help account for and communicate the operational definitions used in collecting software measures.

In order to help the Department of Defense establish a common basis for collecting and interpreting software data on the many systems it develops, the Software Process Measurement group at the Software Engineering Institute developed and released a set of Core DoD Software Measures in 1992. Actually, they did not actually release a set of defined measures, but rather a framework for indicating the elements counted in each of 4 software measurement areas; size, effort, schedule, and defects (Carleton, et al., 1992; Florac, et al., 1992; Goethert, et al., 1992; Park, et al., 1992). These measurement frameworks were based on inputs from several national working groups focusing on the measurement problems faced in each domain.

These frameworks provide worksheets on which the operational definitions used in developing counts of size, effort, schedule, and defects for a system or its components can be indicated. Thus, it is possible to vary the operational definition of a measure across different components of a systems based on unique characteristics or uses. The operational definition is captured as part of the measurement profile permanently attached to that component. Historical data can be interpreted more accurately because the decisions behind the operational

definitions of the available measures are clearly represented to future users of the data. If the data are recorded at a sufficient level of granularity, the counts can be altered based on different operational definitions in the future to serve the purpose of a different analysis.

The important notion here is that operational definitions of software measures need to be captured and preserved along with the data if they are to be useful either as historical data for management use or as a basis for empirical research. Basili, et al. (1983) and Vosburgh, et al. (1984) indicated the importance of validating the measures that are collected against the original operational definitions. In both cases substantial increases in the predictive value of the data were obtained with data that had been validated against operational definitions and collection methods before being entered into a statistical analysis.

Complexity Is Simple

For almost two decades software metricians have been trying to quantify the complexity of a software component (Zuse, 1991). The two most popular complexity metrics remain two of the first proposed, Halstead's (1977) effort measure [E] and McCabe's (1976) cyclomatic complexity [v(G)]. Curtis, et al. (1979a,b) demonstrated that these measures correlated highly with each other and with lines of code. Basili, et al. (1983) were unable to demonstrate that such measures were better than lines of code at predicting any of several performance criteria.

Unfortunately, most people who propose complexity metrics never provide a definition of what complexity is. As a result they propose measures that when studied empirically, primarily measure size. In fact few of these measures capture important aspects of complexity other than the effects that can be accounted for by size alone.

Evangelist (1983) demonstrated that some complexity measures actually increase in value when programs are restructured to decrease their complexity to programmers. That is, many of these measures are inconsistent with the teachings of good programming style (Kernighan & Plauger, 1974). This anomaly strongly argues that the current crop of complexity measures is measuring size far more than any other attribute of software.

Data by Curtis, et al. (1979b, 1989), Curtis (1981), Boehm (1981), Basili and Hutchens (1983), and Valett and McGarry (1989) have demonstrated the overwhelming impact of individual differences among programmers on their performance. Such data suggest that what is complex for one programmer may not be complex for another, an observation that surprises no one, but one which the traditional complexity paradigm fails to explain. In fact, Curtis, et al. (1979b) demonstrated that at least some of these differences depend on the breadth of a programmer's experience. Curtis, Krasner, and Iscoe (1988) reported that some of these differences revolved around the depth of knowledge a programmer had about the application domain of the program they were working with. Differences in the knowledge a programmer brings to working with a program are not captured in any software complexity metrics that have been proposed.

Based on theoretical problems such as these, Curtis (1979) proposed the following definition of complexity 15 years ago this month:

> *"Complexity is an attribute of the interaction between two systems that describes the resources one system will expend in interacting with the other."*

This definition does not treat complexity as a measure derived exclusively from quantifying attributes of the software. For instance, it treats psychological complexity as a phenomenon emerging from an interaction between attributes of the software and attributes of the programmer (Curtis, 1986).

The traditional paradigm for explaining complexity must be abandoned. It only accounts for a few of the factors that interact to create a phenomenon that programmers experience as complexity. For this reason, the current crop of metrics are not, accurately speaking, complexity metrics. They are measures of software attributes such as tokens. decisions, or data flows that may contribute to a program being complex for a given programmer. However, whether the program is complex for a given programmer is partly determined by how much experience the programmer has with similar algorithms, data structures, applications, and other attributes of the program. The impact of this experience factor is unaccounted for in existing complexity paradigms.

Work in cognitive psychology and cognitive science suggests that creating a true complexity measure will be extremely difficult. The software measurement community should learn from the difficulty experienced by reading researchers who have failed to create a simple measure of the complexity of a paragraph. Quite clearly the complexity of a paragraph is partly determined by the knowledge the reader has of the material discussed in the paragraph. This same phenomena occurs in working with programs. Since most existing software complexity metrics correlate so highly with lines of code, it is not clear that they offer any unique contribution to predicting

how hard a program will be to work with beyond that available from simpler measures of program size.

One interesting use of existing software measures suggested by Sunohara, et al. (1981) was to study programs on a scatter plot where tokens or lines are plotted on one axis and decision structure is plotted on the other. In general there was a strong correlation among programs on these measures. However, those programs that deviated strongly from the regression line were often pathological modules. These modules would either have lots of tokens will little decision structure, or lots of decisions for very few tokens.

These anomalous modules seem to have represented a more complex challenge for the programmer developing them. In fact, Basili and Hutchens (1983) demonstrated dramatic differences in the ability of programmers to manage increases in module size. Some have little difficulty while others loose control of larger modules very quickly. Once a programmer has lost control of the design of a module, the kind of anomalies detected in Sunohara et al's. data can occur.

Conclusions

Software is an intellectual artifact. We are fond of saying that software does not follow the laws of physics. Consequently, we should stop trying to measure it as if it obeyed the laws of physics. Software measurement will be hard because the scientific community has not made much progress on measuring knowledge or intellectual artifacts. Within this very challenging context, the software measurement community is called on to produce measures, models, and methods that will at least prove useful for advancing software research or practice. This will require our community to accept that we will not be working with perfect measures and that we need to protect ourselves with our best methodological efforts.

For the field of software measurement to make substantial progress during the remainder of this decade, there are at least seven ± two directions that should be taken.

1) Software measures and models should be developed in ways that are mathematically and statistically tractable. Not only should transformation rules based scales of measurement be observed (Curtis, 1980; Fenton, 1991, Zuse, 1991), but the manipulation of variables in an equation should have some realistic underlying basis in the phenomena being measured or predicted.

2) Software metricians should not develop elaborate equations when the unweighted components from which these equations are developed may provide equal or better prediction. Start with simple measures and only make them more complex when research indicates significant additional prediction can be achieved.

3) It will remain difficult to determine how far any software measurement result can be generalized until serious work has been performed on describing program populations and sub-populations. It is difficult to believe in the validity of existing statistics presented to characterize software phenomena because statistically valid population characterizations are missing.

4) Software metricians must begin to analyze the anomalies in their results rather than glossing over the poor results obtained in some samples with statements about what happens on average. The community needs to know what factors limit the valid use of various software measures.

5) Computer science has surrendered on the notion of characterizing the architecture of programs with broad, general principles. The software measurement community should follow suit. We need to begin developing domain specific software measures that are based on the functional elements and unique performance or architectural attributes of different domains.

6) The operational definition of software measures need to be recorded and maintained along with the data that are collected using them. Frameworks need to be developed that allow software practitioners to operationally define variations on common measures based on the use to which they will be put.

7) The measurement of software attributes should no longer be confused with the concept of software complexity. Software complexity represents an interaction between attributes of the software and attributes of another entity (e.g., a programmer) that interacts with the software. The paradigm for explaining complexity must be changed if the field is to make progress in this area.

Each of these recommendations represents a paradigm shift (some more than others) in either the theory or practice of software measurement. If challenges such as these can be addressed, then the software measurement community will be in a position to lead in computer science research rather than being perceived as a

backwater. Quantification is the essence of science. With renewed effort and some professional risk-taking, the software measurement community could exercise leadership in injecting more science into software engineering.

References

Albrecht, A. (1979). Measuring application development productivity. *Proceedings of the Joint SHARE/GUIDE/IBM Application Development Symposium*. 83-92.

Albrecht, A. (1984). *AD/M Productivity Measurement and Estimate Validation*. Purchase, NY: IBM.

Albrecht, A. & Gaffney, J. (1983). Software function, source lines of code, and development effort prediction: A software science validation. *IEEE Transactions on Software Engineering*, 9(6), 639-647.

Basili, V.R. & Hutchens, D. (1983). An empirical study of a syntactic complexity family. *IEEE Transactions on Software Engineering*, 9(6), 664-672.

Basili, V.R., Selby, R.W., & Phillips, T.-Y. (1983). Metric analysis and data validation across Fortran projects. *IEEE Transactions on Software Engineering*, 9(6), 652-663.

Basili, V.R. & Weiss, D. (1984). A methodology for collecting valid software engineering data. *IEEE Transactions on Software Engineering*, 10(3), 728-738.

Boehm, B. (1981). *Software Engineering Economics*. Englewood Cliffs, NJ: Prentice-Hall.

Carleton, A., Park, R.E., Goethert, W.B., Florac, W.A., Bailey, E.K., & Pfleeger, S.L. (1992). *Software Measurement for DoD Systems: Recommendations for Initial Core Measures (Tech. Rept. CMU/SEI-92-TR-19)*. Pittsburgh: Software Engineering Institute, Carnegie Mellon University.

Curtis, B. (1979). In search of software complexity. *Proceedings of the Workshop on Quantitative Models of Software Reliability, Complexity, and Cost*. Washington, DC: IEEE Computer Society Press.

Curtis, B. (1980). Measurement and experimentation in software engineering. *Proceedings of the IEEE*, 68(9), 1144-1157.

Curtis, B. (1981). Substantiating programmer variability. *Proceedings of the IEEE*, 69 (7), 846.

Curtis, B. (1986). Conceptual issues in software metrics. *Proceedings of the Nineteenth Hawaii International Conference on System Sciences*. 154-157.

Curtis, B., Krasner, H., & Iscoe, N. (1988). A field study of the software design process for large systems. *Communications of the ACM*, 31 (11), 1268-1287.

Curtis, B., Sheppard, S.B., Kruesi-Bailey, E., Bailey, J., & Boehm-Davis, B. (1989). Experimental evaluation of software specification formats. *Journal of Systems and Software*, 9(2), 167-207.

Curtis, B., Sheppard, S.B., & Milliman, P. (1979a). Third time charm: Stronger prediction of programmer performance by software complexity metrics. *Proceedings of the Fourth International Conference on Software Engineering*. Washington, DC: IEEE Computer Society, 356-360.

Curtis, B., Sheppard, S.B., Milliman, P., Borst, M.A., & Love, T. (1979b). Measuring the psychological complexity of software maintenance tasks with the Halstead and McCabe metrics. *IEEE Transactions on Software Engineering*, 5(2), 96-104.

Evangelist, M. (1983). Software complexity metric sensitivity to program restructuring rules. *Journal of Systems and Software*, 3 (3), 231-243.

Fenton, N.E. (1991). *Software Metrics: A Rigorous Approach*. London: Chapman & Hall.

Florac, W.A. (1992). *Software Quality Measurement: A Framework for Counting Problems and Defects (Tech. Rept. CMU/SEI-92-TR-22)*. Pittsburgh: Software Engineering Institute, Carnegie Mellon University.

Garmus, D. (1992). *IFPUG Function Point Counting Practices Manual, Release 3.4*. Burlington, MA: Software Productivity Research.

Goethert, W.B., Bailey, E.K., & Busby, M.B. (1992). *Software Effort and Schedule Measurement: A Framework for Counting Staff Hours and Reporting Schedule Information (Tech. Rept. CMU/SEI-92-TR-21)*. Pittsburgh: Software Engineering Institute, Carnegie Mellon University.

Halstead, M.H. (1977). *Elements of Software Science*. New York: Elsevier.

Henry, S. & Kafura, D. (1981). Software structure metrics based on information flow. *IEEE Transactions on Software Engineering*, 7 (5), 510-518.

Jones, T.C. (1978). Measuring programming quality and productivity. *IBM Systems Journal*, 17 (1), 39-63.

Jones, C. (1986). *Programming Productivity*. New York: McGraw-Hill.

Jones, C. (1992). *Applied Software Measurement: Assuring Productivity and Quality*. New York: McGraw-Hill.

Kernighan, B.W. & Plauger, P.J. (1974). *The Elements of Programming Style*. New York: McGraw-Hill.

Kitchenham, B. (1993). *Proceedings of the First International Workshop on Software Metrics*. Washington, DC: IEEE Computer Society.

McCabe, T. (1976). A complexity measure. *IEEE Transactions on Software Engineering*, 2, 308-320.

Park, R.E. (1992). *Software Size Measurement: A Framework for Counting Source Statements (Tech. Rept. CMU/SEI-92-TR-20)*. Pittsburgh: Software Engineering Institute, Carnegie Mellon University.

Sunohara, T., Takano, A., Uehara, K., & Ohkawa, T. (1981). Program complexity measure for software development management. *Proceedings of the Fifth International Conference on Software Engineering*. Washington, DC: IEEE Computer Society, 100-106.

Valett, J.D. & McGarry, F.E. (1989). A summary of software measurement experiences in the Software Engineering Laboratory. *Journal of Systems and Software*, 9 (2), 137-148.

Vosburgh, J., Curtis, B., Wolverton, R., Albert, B., Malec, H., Hoben, S., & Liu, Y., (1984). Productivity factors and programming environments. *Proceedings of the Seventh International Conference on Software Engineering*. Washington, DC: IEEE Computer Society, 143-152.

Zuse, H. (1991). *Software Complexity: Measures and Methods*. Berlin: de Gruyter.