Software Metrics

SEI Curriculum Module SEI-CM-12-1.1

December 1988

Everald E. Mills
Seattle University

Carnegie Mellon University
Software Engineering Institute

This work was sponsored by the U.S. Department of Defense.

The Software Engineering Institute (SEI) is a federally funded research and development center, operated by Carnegie Mellon University under contract with the United States Department of Defense.

The SEI Education Program is developing a wide range of materials to support software engineering education. A **curriculum module** identifies and outlines the content of a specific topic area, and is intended to be used by an instructor in *designing* a course. A **support materials** package includes materials helpful in *teaching* a course. Other materials under development include textbooks and educational software tools.

SEI educational materials are being made available to educators throughout the academic, industrial, and government communities. The use of these materials in a course does not in any way constitute an endorsement of the course by the SEI, by Carnegie Mellon University, or by the United States government.

SEI curriculum modules may be copied or incorporated into other materials, but not for profit, provided that appropriate credit is given to the SEI and to the original author of the materials.

Requests for additional information should be addressed to the Director of Education, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania 15213.

Comments on SEI materials are solicited, and may be sent to the Director of Education, or to the module author.

Everald E. Mills
Software Engineering Department
Seattle University
Seattle, Washington 98122

This technical report was prepared for the

SEI Joint Program Office
ESD/AVS
Hanscom AFB, MA 01731

The ideas and findings in this report should not be construed as an official DoD position.
It is published in the interest of scientific and technical information exchange.

Review and Approval

This report has been reviewed and is approved for publication.

FOR THE COMMANDER

Karl H. Shingler
SEI Joint Program Office

Software Metrics

Acknowledgements

I would like to express my appreciation to Norm Gibbs, Director of Education at the Software Engineering Institute, and to his staff, for their generous support in my development of this curriculum module. Special thanks go to Lionel Deimel and Linda Pesante for their assistance with the final form of the document.

Other individuals who were especially helpful were Karola Fuchs and Sheila Rosenthal with library materials, and Allison Brunvand with administrative details and logistics.

Contents

Software Metrics

Module Revision History

Version 1.1 (December 1988)	General revision and updating, especially of bibliography
Version 1.0 (October 1987)	Draft for public review

Software Metrics

Capsule Description

Effective management of any process requires quantification, measurement, and modeling. Software metrics provide a quantitative basis for the development and validation of models of the software development process. Metrics can be used to improve software productivity and quality. This module introduces the most commonly used software metrics and reviews their use in constructing models of the software development process. Although current metrics and models are certainly inadequate, a number of organizations are achieving promising results through their use. Results should improve further as we gain additional experience with various metrics and models.

Philosophy

It has been noted frequently that we are experiencing a software crisis, characterized by our inability to produce correct, reliable software within budget and on time. No doubt, many of our failures are caused by the inherent complexity of the software development process, for which there often is no analytical description. These problems can be ameliorated, however, by improving our software management capabilities. This requires both the development of improved software metrics and improved utilization of such metrics.

Unfortunately, the current state of software metrics is confused. Many metrics have been invented. Most of these have been defined and then tested only in a limited environment, if at all. In some cases, remarkable successes have been reported in the initial application or validation of these metrics. However, subsequent attempts to test or use the metrics in other situations have yielded very different results. One part of the problem is that we have failed to identify a commonly accepted set of software properties to be measured. As a result, the same metric has been used to measure very different software properties. Moreover, we have virtually no theoretical models and a multitude of metrics, only a few of which have enjoyed any widespread use or acceptance.

Faced with this situation, the author has chosen to indicate the great diversity of metrics that have been proposed and to discuss some of the most common ones in detail. In the process, the underlying assumptions, environment of application, and validity of various metrics are examined. The author believes that current metrics and models are far from perfect, but that properly applied metrics and models can provide significant improvements in the software development process.

Objectives

The following is a list of possible educational objectives based upon the material in this module. Objectives for any particular unit of instruction may be drawn from these or related objectives, as may be appropriate to audience and circumstance. (See *Teaching Considerations* for further suggestions.)

Cognitive Domain

1. (Knowledge) The student can define the basic terminology and state fundamental facts about software metrics and process models. (For example, identify the metrics and models that have been proposed and used by significant numbers of people.)
2. (Comprehension) The student can explain the metrics and models discussed in the module and summarize the essential characteristics of each.
3. (Application) The student can calculate the values of the metrics discussed for

specific examples of software products or processes. (For example, compute LOC or *v(G)* for specific programs or apply the COCOMO model to the development process for a specified product.)

4. (Analysis) The student can identify the essential elements of a given metric or model, describe the interrelationships among its various elements, and discuss the circumstances or environments in which its use is appropriate.
5. (Synthesis) The student can develop a plan for a metrics program for a software development organization, using prescribed metrics.
6. (Evaluation) The student can evaluate a metrics program by analyzing the metrics and models in use and making judgments concerning their application in a particular environment.

Affective Domain

1. The student will realize the difficulty and effort involved in establishing precise, reliable software metrics and models.
2. The student will appreciate the importance of software metrics in the control and management of the software development process.
3. The student will be more likely to support implementation and use of appropriate software metrics.

Prerequisite Knowledge

The following are recommended prerequisites for the study of software metrics:

1. Knowledge of basic statistics and experimental design.
2. Basic understanding of commonly used software life cycle models, at least to the level covered in an introductory senior- or graduate-level software engineering course.
3. Experience working as a team member on a software development project.

The reason for the statistical prerequisite may not be immediately obvious. Exploring and validating software metrics requires sound statistical methods and unbiased experimental designs. The student needs to understand the fundamentals of experiment design, know what methods are available for data analysis, and be able to select appropriate techniques in specific circumstances. Furthermore, the student needs to understand the concept of statistical significance and how to test for it in the analyses usually performed to validate software metrics. Of particular interest are various correlation techniques, regression analysis, and statistical tests for significance.

The need for familiarity with the typical software development cycle and experience with software development should be self-evident.

These prerequisites are, in the author's view, essential for attaining the cognitive objectives listed above. Prerequisites for any particular unit of instruction, of course, depend upon specific teaching objectives. (See *Teaching Considerations*.)

Module Content

Outline

I. Introduction
  1. The "Software Crisis"
  2. The Need for Software Metrics
  3. Definition of Software Metrics
  4. Classification of Software Metrics
  5. Measurement Scales for Software Metrics
  6. Current State of Software Metrics

II. Product Metrics
  1. Size Metrics
    a. Lines of Code
    b. Function Points
    c. *Bang*
  2. Complexity Metrics
    a. Cyclomatic Complexity—$v(G)$
    b. Extensions to $v(G)$
    c. Knots
    d. Information Flow
  3. Halstead's Product Metrics
    a. Program Vocabulary
    b. Program Length
    c. Program Volume
  4. Quality Metrics
    a. Defect Metrics
    b. Reliability Metrics
    c. Maintainability Metrics

III. Process Metrics, Models, and Empirical Validation
  1. General Considerations
  2. Empirical Models
  3. Statistical Models
  4. Theory-Based Models
    a. Rayleigh Model
    b. Software Science Model—Halstead
  5. Composite Models
    a. COCOMO—Boehm
    b. SOFTCOST—Tausworthe
    c. SPQR Model—Jones
    d. COPMO—Thebaut
    e. ESTIMACS—Rubin
  6. Reliability Models

IV. Implementation of a Metrics Program
  1. Planning Process
    a. Defining Objectives
    b. Initial Estimates of Effort and Cost
  2. Selection of Model and Metrics
    a. Projected Ability to Meet Objectives
    b. Estimated Data Requirements and Cost
  3. Data Requirements and Database Maintenance
    a. Specific Data Required
    b. Data Gathering Procedures
    c. Database Maintenance
    d. Refined Estimates of Efforts and Costs
  4. Initial Implementation and Use of the Model
    a. Clarification of Use
    b. Responsible Personnel
  5. Continuing Use and Refinement
    a. Evaluating Results
    b. Adjusting the Model

V. Trends in Software Metrics

Annotated Outline

I. Introduction
  1. The "Software Crisis"

It has been estimated that, by 1990, fully one half of the American work force will rely on computers and software to do its daily work. As computer hardware costs continue to decline, the demand for new applications software continues to increase at a rapid rate. The existing inventory of software continues to grow, and the effort required to maintain it continues to increase as well. At the same time, there is a significant shortage of qualified software professionals. Combining these factors, one might project that at some point in the not-too-distant future, every American worker will have to be involved in software development and maintenance. Meanwhile, the software development scene is often characterized by:

- schedule and cost estimates that are grossly inaccurate,

- software of poor quality, and
- a productivity rate that is increasing more slowly than the demand for software.

This situation has often been referred to as the "software crisis" [Arthur85].

2. The Need for Software Metrics

The software crisis must be addressed and, to the extent possible, resolved. To do so requires more accurate schedule and cost estimates, better quality products, and higher productivity. All these can be achieved through more effective software management, which, in turn, can be facilitated by the improved use of software metrics. Current software management is ineffective because software development is extremely complex, and we have few well-defined, reliable measures of either the process or the product to guide and evaluate development. Thus, accurate and effective estimating, planning, and control are nearly impossible to achieve [Rubin-83]. Improvement of the management process depends upon improved ability to identify, measure, and control essential parameters of the development process. This is the goal of software metrics—the identification and measurement of the essential parameters that affect software development.

Software metrics and models have been proposed and used for some time [Wolverton74, Perlis81]. Metrics, however, have rarely been used in any regular, methodical fashion. Recent results indicate that the conscientious implementation and application of a software metrics program can help achieve better management results, both in the short run (for a given project) and in the long run (improving productivity on future projects) [Grady87]. Most software metrics cannot meaningfully be discussed in isolation from such metrics programs. Better use of existing metrics and development of improved metrics appear to be important factors in the resolution of the software crisis.

3. Definition of Software Metrics

It is important to further define the term *software metrics* as used in this module. Essentially, *software metrics* deals with the measurement of the software product and the process by which it is developed. In this discussion, the software product should be viewed as an abstract object that evolves from an initial statement of need to a finished software system, including source and object code and the various forms of documentation produced during development. Ordinarily, these measurements of the software process and product are studied and developed for use in modeling the software development process. These metrics and models are then used to estimate/predict product costs and schedules and to measure productivity and product quality. Information gained from the metrics and the model can then be used in the management and control of the development process, leading, one hopes, to improved results.

Good metrics should facilitate the development of models that are capable of predicting process or product parameters, not just describing them. Thus, ideal metrics should be :

- simple, precisely definable—so that it is clear how the metric can be evaluated;
- objective, to the greatest extent possible;
- easily obtainable (*i.e.*, at reasonable cost);
- valid—the metric should measure what it is intended to measure; and
- robust—relatively insensitive to (intuitively) insignificant changes in the process or product.

In addition, for maximum utility in analytic studies and statistical analyses, metrics should have data values that belong to appropriate measurement scales [Conte86, Basili84].

It has been observed that the fundamental qualities required of any technical system are [Ferrari86]:

- functionality—correctness, reliability, etc.;
- performance—response time, throughput, speed, etc.; and
- economy—cost effectiveness.

So far as this author can discern, *software metrics*, as the term is most commonly used today, concerns itself almost exclusively with the first and last of the above characteristics, *i.e.*, functionality and economy. Performance is certainly important, but it is not generally included in discussions of software metrics, except regarding whether the product meets specific performance requirements for that product. The evaluation of performance is often treated extensively by those engaged in *performance evaluation* studies, but these are not generally included in what is referred to as *software metrics* [Ferrari86].

It is possible that, in the future, the scope of *software metrics* may be expanded to include performance evaluation, or that both activities may be considered part of a larger area that might be called *software measurement*. For now, however, this module will confine itself to software metrics as defined above.

4. Classification of Software Metrics

Software metrics may be broadly classified as either *product metrics* or *process metrics*. *Product metrics* are measures of the software product at any stage of its development, from requirements to installed system. Product metrics may measure the complexity of the software design, the size of the final program (either source or object code), or the number of pages of documentation produced. *Process metrics*, on the other hand, are measures of the software de-

velopment process, such as overall development time, type of methodology used, or the average level of experience of the programming staff.

In addition to the distinction between product and process metrics, software metrics can be classified in other ways. One may distinguish *objective* from *subjective* properties (metrics). Generally speaking, objective metrics should always result in identical values for a given metric, as measured by two or more qualified observers. For subjective metrics, even qualified observers may measure different values for a given metric, since their subjective judgment is involved in arriving at the measured value. For product metrics, the size of the product measured in lines of code (LOC) is an objective measure, for which any informed observer, working from the same definition of LOC, should obtain the same measured value for a given program. An example of a subjective product metric is the classification of the software as "organic," "semi-detached," or "embedded," as required in the COCOMO cost estimation model [Boehm81]. Although most programs might be easy to classify, those on the borderline between categories might reasonably be classified in different ways by different knowledgeable observers. For process metrics, development time is an example of an objective measure, and level of programmer experience is likely to be a subjective measure.

Another way in which metrics can be categorized is as *primitive* metrics or *computed* metrics [Grady87]. Primitive metrics are those that can be directly observed, such as the program size (in LOC), number of defects observed in unit testing, or total development time for the project. Computed metrics are those that cannot be directly observed but are computed in some manner from other metrics. Examples of computed metrics are those commonly used for productivity, such as LOC produced per person-month (LOC/person-month), or for product quality, such as the number of defects per thousand lines of code (defects/KLOC). Computed metrics are combinations of other metric values and thus are often more valuable in understanding or evaluating the software process than are simple metrics.

Although software metrics can be neatly categorized as primitive objective product metrics, primitive subjective product metrics, etc., this module does not strictly follow that organization. Rather, the discussion reflects areas where most of the published work has been concentrated; no exhaustive coverage of all possible types of software metrics is attempted here. As is evident below, a great deal of work has been done in some areas, such as objective product metrics, and much less in other areas, such as subjective product metrics.

5. Measurement Scales for Software Metrics

Software metric data should be collected with a specific purpose in mind. Ordinarily, the purpose is for use in some process model, and this may involve using the data in other calculations or subjecting them to statistical analyses. Before data are collected and used, it is important to consider the type of information involved. Four basic types of measured data are recognized by statisticians—nominal, ordinal, interval, and ratio. (The following discussion of these types of data is adapted from [Conte86], beginning on page 127.)

The four basic types of data are described by the following table:

Type of Data	*Possible Operations*	*Description of Data*
Nominal	=, ≠	Categories
Ordinal	<, >	Rankings
Interval	+, -	Differences
Ratio	/	Absolute zero

Operations in this table for a given data type also apply to all data types appearing below it.

Examples of software metrics can be found for each type of data.

As an example of *nominal data*, one can measure the type of program being produced by placing it in to a category of some kind—database program, operating system, etc. For such data, we cannot perform arithmetic operations of any type or even rank the possible values in any "natural order." The only possible operation is to determine whether program **A** is of the same type as program **B**. Such data are said to have a *nominal* scale, and the particular example given can be an important parameter in a model of the software development process. The data might be considered either subjective or objective, depending upon whether the rules for classification allow equally qualified observers to arrive at different classifications for a given program.

Ordinal data, by contrast, allow us to rank the various data values, although differences or ratios between values are not meaningful. For example, programmer experience level may be measured as *low*, *medium*, or *high*. (In order for this to be an objective metric, one must assume that the criteria for placement in the various categories are well-defined, so that different observers always assign the same value to any given programmer.)

Data from an *interval* scale can not only be ranked, but also can exhibit meaningful differences between values. McCabe's complexity measure [McCabe76]

might be interpreted as having an interval scale. Differences appear to be meaningful; but there is no absolute zero, and ratios of values are not necessarily meaningful. For example, a program with complexity value of 6 is 4 units more complex than a program with complexity of 2, but it is probably not meaningful to say that the first program is three times as complex as the second.

Some data values are associated with a *ratio* scale, which possesses an absolute zero and allows meaningful ratios to be calculated. An example is program size, in lines of code (LOC). A program of 2,000 lines can reasonably be interpreted as being twice as large as a program of 1,000 lines, and programs can obviously have zero length according to this measure.

It is important to be aware of what measurement scale is associated with a given metric. Many proposed metrics have values from an interval, ordinal, or even nominal scale. If the metric values are to be used in mathematical equations designed to represent a model of the software process, metrics associated with a ratio scale may be preferred, since ratio scale data allow most mathematical operations to be meaningfully applied. However, it seems clear that the values of many parameters essential to the software development process cannot be associated with a ratio scale, given our present state of knowledge. This is seen, for example, in the categories of COCOMO.

6. Current State of Software Metrics

The current state of software metrics is not very satisfying. In the past, many metrics and a number of process models have been proposed [Mohanty81, Kafura85, Kemerer87, Rubin87]. Unfortunately, most of the metrics defined have lacked one or both of two important characteristics :

- a sound conceptual, theoretical basis
- statistically significant experimental validation

Most metrics have been defined by an individual and then tested and used only in a very limited environment. In some cases, significant successes have been reported in the validation or application of these metrics. However, subsequent attempts to test or use the metrics in other environments have yielded very different results. These differences are not surprising in view of the lack of clear definitions and testable hypotheses. Nevertheless, discrepancies and disagreements in reported results have left many observers with the sense that the field of software metrics is, at best, insufficiently mature to be of any practical use.

The metrics field has no clearly defined, commonly accepted set of essential software properties it attempts to measure; however, it does have a large number of metrics, only a few of which have enjoyed any widespread use or acceptance. Even in the case of widely studied metrics, such as LOC, Halstead's metrics, and McCabe's cyclomatic complexity, it is not universally agreed what they measure. In various reported studies, attempts have been made to correlate these metrics with a number of software properties, including size, complexity, reliability (error rates), and maintainability [Curtis79a, Curtis79b, Kafura85, Li87, Potier82, Woodfield81]. Thus, it is little wonder that software practitioners are wary of any claims on behalf of software metrics.

Many apparently important software metrics, such as type of product or level of programming expertise, must be considered subjective metrics at this time, although they may be defined more objectively in the future. These metrics are difficult to construct because of the potentially large number of factors involved and the problems associated with assessing or quantifying individual factors. As a result, little definitive work has been done to reduce the uncertainty associated with these metrics.

As for the proposed process models, few of these have a significant theoretical basis. Most are based upon a combination of intuition, expert judgment, and statistical analysis of empirical data. Overall, the work has failed to produce any single process model that can be applied with a reasonable degree of success to a variety of environments. Generally, significant recalibration is required for each new environment in order to produce useful results. Furthermore, the various models often use widely different sets of basic parameters. Thus, even a relatively small set of universally useful metrics has not yet emerged.

As a result of the above considerations, it is very difficult to interpret and compare quoted metric results, especially if they involve different environments, languages, applications, or development methodologies. Even with an apparently simple metric, such as LOC, differences in underlying definitions and counting techniques may make it impossible to compare quoted results [Jones86]. If different programming languages are involved, metrics involving LOC values can, if not carefully interpreted, lead to incorrect conclusions and thereby conceal the real significance of the data. For example, the (computed) productivity metric LOC per unit-time (LOC/month, for example) and cost per LOC ($/LOC) are often used. However, if they are not interpreted carefully, these metrics can suggest that assembly language programmers are more productive than high-level language programmers (higher LOC/month and lower $/LOC), even though the total programming cost is usually lower when using a high-level language. Similarly, defects per LOC and cost per defect values have often been

used as quality or productivity indicators. As in the above case, when programming languages at different levels are involved, these metrics may obscure overall productivity and quality improvements by systematically yielding lower defect per LOC and cost per defect values for lower-level languages, even though total defects and costs are actually higher.

Despite these problems, it appears that the judicious, methodical application of software metrics and models in limited environments can aid significantly in improving software quality and productivity [Basili87, Grady87]. In many cases, relatively simple metrics such as LOC and McCabe's complexity metric, $v(G)$, have been found to be reasonably good predictors of other characteristics, such as defect counts, total effort, and maintainability [Grady87, Li87, Rombach87]. Thus, although useful metrics and models cannot yet be pulled off the shelf and used indiscriminately, careful application of some of the metrics and models already available can yield useful results if tuned to a particular environment. These results will improve further as we gain additional experience with current models and achieve better understanding of the underlying metrics and their application to the software process.

II. Product Metrics

Most of the initial work in product metrics dealt with the characteristics of source code. As we have gained experience with metrics and models, it has become increasingly apparent that metric information available earlier in the development cycle can be of greater value in controlling the process and results. Thus, for example, a number of papers have dealt with the size or complexity of the software design [Troy81, Henry84, Yau85]. More recently, Card and Agresti have devised a metric for architectural design complexity and compared it with subjective judgments and objective error rates [Card88].

A number of product metrics are discussed below. These examples were chosen because of their wide use or because they represent a particularly interesting point of view. No attempt has been made in this module to provide examples of metrics applicable to each work product of the software development cycle. Rather, the examples discussed reflect the areas where most work on product metrics has been done. References have been provided for readers who are interested in pursuing specialized metrics.

1. Size Metrics

A number of metrics attempt to quantify software "size." The metric that is most widely used, LOC, suffers from the obvious deficiency that its value cannot be measured until after the coding process has been completed. Function points and system *Bang* have the advantage of being measurable earlier in the development process—at least as early as the design phase, and possibly earlier. Some of Halstead's metrics are also used to measure software size, but these are discussed later.

a. Lines of Code

Lines of code (or LOC) is possibly the most widely used metric for program size. It would seem to be easily and precisely definable; however, there are a number of different definitions for the number of lines of code in a particular program. These differences involve treatment of blank lines and comment lines, non-executable statements, multiple statements per line, and multiple lines per statement, as well as the question of how to count reused lines of code. The most common definition of LOC seems to count any line that is not a blank or comment line, regardless of the number of statements per line [Boehm-81, Jones86].

LOC has been theorized to be useful as a predictor of program complexity, total development effort, and programmer performance (debugging, productivity). Numerous studies have attempted to validate these relationships. Examples are those of [Woodfield81] comparing LOC, McCabe's $v(G)$, and Halstead's E as indicators of programming effort and [Curtis79a] and [Curtis-79b] comparing LOC with other metrics, as indicators of programmer performance.

In a recent study, Levitin concludes that LOC is a poorer measure of size than Halstead's program length, N, discussed below [Levitin86].

b. Function Points

Albrecht has proposed a measure of software size that can be determined early in the development process. The approach is to to compute the total *function points* (FP) value for the project, based upon the number of external user inputs, inquiries, outputs, and master files. The value of FP is the total of these individual values, with the following weights applied: inputs: 4, outputs: 5, inquiries: 4, and master files: 10. Each FP contributor can also be adjusted within a range of $\pm$ 35% for specific project complexity [Albrecht-83]. Function points are intended to be a measure of program size and, thus, effort required for development. Examples of studies that validate this metric are those of Albrecht [Albrecht83] (comparing LOC and FP as predictors of development effort) and Behrens [Behrens83] (attempting to correlate FP values with productivity and development effort in a production environment). A more recent study has been reported by Knafl and Sacks [Knafl86].

c. *Bang*

DeMarco defines system *Bang* as a function metric, indicative of the size of the system. In effect, it measures the total functionality of the software system delivered to the user. *Bang* can be calculated from certain algorithm and data primitives available from a set of formal specifications for the software. The model provides different formulas and criteria for distinguishing between complex algorithmic versus heavily data-oriented systems. Since *Bang* measures the functionality delivered to the user, DeMarco suggests that a reasonable project goal is to maximize "*Bang* per Buck"—*Bang* divided by the total project cost [DeMarco82].

2. Complexity Metrics

Numerous metrics have been proposed for measuring program complexity—probably more than for any other program characteristic. The examples discussed below are some of the better known complexity metrics. A recent study by Li and Cheung compares 31 different complexity metrics, including most of those discussed below [Li87]. Another recent study by Rodriguez and Tsai compares LOC, $v(G)$, Kafura's information flow metric, and Halstead's volume, V, as measures of program size, complexity, and quality [Rodriguez86]. Attempts to devise new measures of software complexity continue, as evidenced by recent articles [Card88, Harrison87].

As noted for size metrics, measures of complexity that can be computed early in the software development cycle will be of greater value in managing the software process. Theoretically, McCabe's measure [McCabe76] is based on the final form of the computer code. However, if the detailed design is specified in a program design language (PDL), it should be possible to compute $v(G)$ from that detailed design. This is also true for the information flow metric of Kafura and Henry [Kafura81]. It should be noted here that Halstead's metrics [Halstead77] are often studied as possible measures of software complexity.

a. Cyclomatic Complexity—$v(G)$

Given any computer program, we can draw its control flow graph, G, wherein each node corresponds to a block of sequential code and each arc corresponds to a branch or decision point in the program. The cyclomatic complexity of such a graph can be computed by a simple formula from graph theory, as $v(G) = e - n + 2$, where e is the number of edges, and n is the number of nodes in the graph. McCabe proposed that $v(G)$ can be used as a measure of program complexity and, hence, as a guide to program development and testing. For structured programs, $v(G)$ can be computed without reference to the program flow graph by using only the number of decision points in the program text [McCabe76]. McCabe's cyclomatic complexity metric has been related to programming effort, debugging performance, and maintenance effort. The studies by Curtis and Woodfield referenced earlier also report results for this metric [Curtis79b, Woodfield81, Harrison82].

b. Extensions to $v(G)$

Myers noted that McCabe's cyclomatic complexity measure, $v(G)$, provides a measure of program complexity but fails to differentiate the complexity of some rather simple cases involving single conditions (as opposed to multiple conditions) in conditional statements. As an improvement to the original formula, Myers suggests extending $v(G)$ to $v'(G) = [l{:}u]$, where l and u are lower and upper bounds, respectively, for the complexity. This formula gives more satisfactory results for the cases noted by Myers [Myers77].

Stetter proposed that the program flow graph be expanded to include data declarations and data references, thus allowing the graph to depict the program complexity more completely. If H is the new program flow graph, it will generally contain multiple entry and exit nodes. A function $f(H)$ can be computed as a measure of the *flow complexity* of program H. The deficiencies noted by Myers are also eliminated by $f(H)$ [Stetter84].

c. Knots

The concept of program *knots* is related to drawing the program control flow graph with a node for every statement or block of sequential statements. A knot is then defined as a necessary crossing of directional lines in the graph. The same phenomenon can also be observed by simply drawing transfer-of-control lines from statement to statement in a program listing. The number of knots in a program has been proposed as a measure of program complexity [Woodward79].

d. Information Flow

The information flow within a program structure may also be used as a metric for program complexity. Kafura and Henry have proposed such a measure. Basically, their method counts the number of local information flows entering (fan-in) and exiting (fan-out) each procedure. The procedure's complexity is then defined as:

$$c = [\ \textit{procedure length}\ ] \cdot [\ \textit{fan–in} \cdot \textit{fan–out}\ ]^2$$

[Kafura81]. This information flow metric is compared with Halstead's E metric and McCabe's

cyclomatic complexity in [Henry81]. Complexity metrics such as $v(G)$ and Kafura's information flow metric have been shown by Rombach to be useful measures of program maintainability [Rombach87].

3. Halstead's Product Metrics

Most of the product metrics proposed have applied to only one particular aspect of the software product. In contrast, Halstead's software science proposed a unified set of metrics that apply to several aspects of programs, as well as to the overall software production effort. Thus, it is the first set of software metrics unified by a common theoretical basis. In this section, we discuss the program vocabulary (n), length (N), and volume (V) metrics. These metrics apply specifically to the final software product. Halstead also specified formulas for computing the total effort (E) and development time (T) for the software product. These metrics are discussed in Section III.

a. Program Vocabulary

Halstead theorized that computer programs can be visualized as a sequence of tokens, each token being classified as either an operator or operand. He then defined the *vocabulary*, n, of the program as:

$$n = n_1 + n_2 \ ,$$

where n_1 = the number of unique operators in the program and
n_2 = the number of unique operands in the program.

Thus, n is the total number of unique tokens from which the program has been constructed [Halstead77].

b. Program Length

Having identified the basic tokens used to construct the program, Halstead then defined the program length, N, as the count of the total number of operators and operands in the program. Specifically:

$$N = N_1 + N_2 \ ,$$

where N_1 = the total number of operators in the program and
N_2 = the total number of operands in the program.

Thus, N is clearly a measure of the program's size, and one that is derivable directly from the program itself. In practice, however, the distinction between operators and operands may be nontrivial, thus complicating the counting process [Halstead77].

Halstead theorized that an estimated value for N, designated N', can be calculated from the values of n_1 and n_2 by using the following formula:

$$N' = n_1 \log_2 n_1 + n_2 \log_2 n_2 \ .$$

Thus, N is a primitive metric, directly observable from the finished program, while N' is a computed metric, which can be calculated from the actual or estimated values of n_1 and n_2 before the final code is actually produced. A number of studies lend empirical support to the validity of the equation for the computed program length, N'. Examples are reported by Elshoff and can also be found in Halstead's book. Other studies have attempted to relate N and N' to other software properties, such as complexity [Potier82] and defect rates [Elshoff76, Halstead77, Levitin86, Li87, Shen85].

c. Program Volume

Another measure of program size is the program volume, V, which was defined by Halstead as:

$$V = N \cdot \log_2 n \ .$$

Since N is a pure number, the units of V can be interpreted as bits, so that V is a measure of the storage volume required to represent the program. Empirical studies by Halstead and others have shown that the values of LOC, N, and V appear to be linearly related and equally valid as relative measures of program size [Christensen81, Elshoff78, Li87].

4. Quality Metrics

One can generate long lists of quality characteristics for software—correctness, efficiency, portability, maintainability, reliability, etc. Early examples of work on quality metrics are discussed by Boehm, McCall, and others [Boehm76, McCall77]. Unfortunately, the characteristics often overlap and conflict with one another; for example, increased portability (desirable) may result in lowered efficiency (undesirable). Thus, useful definitions of general quality metrics are difficult to devise, and most computer scientists have abandoned efforts to find any single metric for overall software quality.

Although a good deal of work has been done in this area, it exhibits less commonality of direction or definition than other areas of metric research, such as software size or complexity. Three areas that have received considerable attention are: program correctness, as measured by defect counts; software reliability, as computed from defect data; and software maintainability, as measured by various other metrics, including complexity metrics. Examples from these areas are discussed briefly below.

Software quality is a characteristic that, theoretically at least, can be measured at every phase of the soft-

ware development cycle. Cerino discusses the measurement of quality at some of these phases in [Cerino86].

a. Defect Metrics

The number of defects in the software product should be readily derivable from the product itself; thus, it qualifies as a product metric. However, since there is no effective procedure for counting the defects in the program, the following alternative measures have been proposed :

- number of design changes
- number of errors detected by code inspections
- number of errors detected in program tests
- number of code changes required

These alternative measures are dependent upon both the program and the outcome or result of some phase of the development cycle.

The number of defects observed in a software product provides, in itself, a metric of software quality. Studies have attempted to establish relationships between this and other metrics that might be available earlier in the development cycle and that might, therefore, be useful as predictors of program quality [Curtis79b, Potier82, Shen85, Rodriguez86].

b. Reliability Metrics

It would be useful to know the probability of software failure, or the rate at which software errors will occur. Again, although this information is inherent in the software product, it can only be estimated from data collected on software defects as a function of time. If certain assumptions are made, these data can then be used to model and compute software reliability metrics. These metrics attempt to measure and predict the probability of failure during a particular time interval, or the mean time to failure (MTTF). Since these metrics are usually discussed in the context of developing a reliability model of the software product's behavior, more detailed discussion of this model is deferred to the section on process models. Significant references in this area are [Ruston79], [Musa75], and [Musa87].

c. Maintainability Metrics

A number of efforts have been made to define metrics that can be used to measure or predict the maintainability of the software product [Yau80, Yau85]. For example, an early study by Curtis, *et al.*, investigated the ability of Halstead's effort metric, E, and $v(G)$ to predict the psychological complexity of software maintenance tasks [Curtis-79a]. Assuming such predictions could be made accurately, complexity metrics could then be profitably used to reduce the cost of software maintenance [Harrison 82]. More recently, Rombach has published the results of a carefully designed experiment that indicates that software complexity metrics can be used effectively to explain or predict the maintainability of software in a distributed computer system [Rombach87]. A similar study, based on three different versions of a medium-sized software system that evolved over a period of three years, relates seven different complexity metrics to the recorded experience with maintenance activities [Kafura87]. The complexity metrics studied included both measures of the internal complexity of software modules and measures of the complexity of interrelationships between software modules. The study indicates that such metrics can be quite useful in measuring maintainability and in directing design or redesign activities to improve software maintainability.

III. Process Metrics, Models, and Empirical Validation

1. General Considerations

Software metrics may be defined without specific reference to a well-defined model, as, for example, the metric LOC for program size. However, more often metrics are defined or used in conjunction with a particular model of the software development process. In this curriculum module, the intent is to focus on those metrics that can best be used in models to predict, plan, and control software development, thereby improving our ability to manage the process.

Models of various types are simply abstractions of the product or process we are interested in describing. Effective models allow us to ignore uninteresting details and concentrate on essential aspects of the artifact described by the model. Preference should be given to the simplest model that provides adequate descriptive capability and some measure of intuitive acceptability. A good model should possess predictive capabilities, rather than being merely descriptive or explanatory.

In general, models may be analytic-constructive or empirical-descriptive in nature. There have been few analytic models of the software process, the most notable exception being Halstead's software science, which has received mixed reactions. Most proposed software models have resulted from a combination of intuition about the basic form of relationships and the use of empirical data to determine the specific quantities involved (the coefficients of independent variables in hypothesized equations, for example).

Ultimately, the validity of software metrics and

models must be established by demonstrated agreement with empirical or experimental data. This requires careful attention to taking measurements and analyzing data. In general, the work of analyzing and validating software metrics and models requires both sound statistical methods and sound experimental designs. Precise definitions of the metrics involved and the procedures for collecting the data are essential for meaningful results. Small-scale experiments should be designed carefully, using well-established principles of experimental design. Unfortunately, validation of process models involving larger projects must utilize whatever data can be collected. Carefully controlled large experiments are virtually impossible to conduct. Guidance in the area of data collection for software engineering experiments is provided by Basili and Weiss [Basili84].

A knowledge of basic statistical theory is essential for conducting meaningful experiments and analyzing the resulting data. In attempting to validate the relationships of a given model, one must use appropriate statistical procedures and be careful to interpret the results objectively. Most studies of software metrics have used some form of statistical correlation, often without proper regard for the theoretical basis or limitations of the methods used. In practice, software engineers lacking significant background in statistical methods should consider enlisting the aid of a statistical consultant if serious metric evaluation work is undertaken.

Representative examples of software models are presented below. For papers that compare the various models, see [Kemerer87] and [Rubin87].

Teaching Consideration: *In any given unit of instruction based on this module, at least one or two examples of each type of model should be covered in some detail.*

2. Empirical Models

One of the earliest models used to project the cost of large-scale software projects was described by Wolverton of TRW in 1974. The method relates a proposed project to similar projects for which historical cost data are available. It is assumed that the cost of the new project can be projected using this historical data. The method assumes that a waterfall-style life cycle model is used. A 25×7 structural forecast matrix is used to allocate resources to various phases of the life cycle. In order to determine actual software costs, each software module is first classified as belonging to one of six basic types—control, I/O, etc. Then, a level of difficulty is assigned by categorizing the module as new or old and as easy, medium, or hard. This gives a total of six levels of module difficulty. Finally, the size of the module is estimated, and the system cost is determined from historical cost data for software with similar size, type, and difficulty ratings [Wolverton74].

3. Statistical Models

C. E. Walston and C. P. Felix of IBM used data from 60 previous software projects completed by the Federal Systems Division to develop a simple model of software development effort. The metric LOC was assumed to be the principal determiner of development effort. A relationship of the form

$$E = aL^b$$

was assumed, where L is the number of lines of code, in thousands, and E is the total effort required, in person-months. Regression analysis was used to find appropriate values of parameters a and b. The resulting equation was

$$E = 5.2L^{0.91} .$$

Nominal programming productivity, in LOC per person-month, can then be calculated as L/E. In order to account for deviations from the derived form for E, Walston and Felix also tried to develop a productivity index, I, which would increase or decrease the productivity, depending upon the nature of the project. The computation of I was to be based upon evaluations of 29 project variables (culled from an original list of 68 possible determiners of I) [Walston77].

4. Theory-Based Models

Few of the proposed models have substantial theoretical bases. Two examples that do are presented below.

a. Rayleigh Model

L. H. Putnam developed a model of the software development process based upon the assumption that the personnel utilization during program development is described by a Rayleigh-type curve such as the following:

$$y = \frac{Kte^{-t^2/2T^2}}{T^2} ,$$

where y = the number of persons on the project at any time, t;
K = the area under the Rayleigh curve, equal to the total life cycle effort in person-years; and
T = development time (time of peak staffing requirement).

Putnam assumed that either the overall staffing curve or the staffing curves for individual phases of the development cycle can be modeled by an equation of this form. He then developed the following relationship between the size of the software product and the development time [Putnam78, Putnam80]:

$$S = CK^{1/3}T^{4/3} ,$$

where S = the number of source LOC delivered;

K = the life-cycle effort in person-years; and
C = a state-of-technology constant.

b. Software Science Model—Halstead

The software science equations can be used as a simple theoretical model of the software development process. The effort required to develop the software is given by the equation $E = V/L$, which can be approximated by:

$$E = \frac{n_1 n_2 [n_1 \log_2 n_1 + n_2 \log_2 n_2] \log_2 n}{2n_2}.$$

The units of E are *elementary mental discriminations*. The corresponding programming time (in seconds) is simply derived from E by dividing by the Stroud number, S:

$$T = E / S \ .$$

The value of S is usually taken as 18 for these calculations. If only the value of length, N, is known, then the following approximation can be used for computing T:

$$T = \frac{N^2 \log_2 n}{4S} \ ,$$

where n can be obtained from the relationship $N = n \log_2 (n/2)$ [Halstead77, Woodfield81].

5. Composite Models

As experience has been gained with previous models, a number of more recent models have utilized some combination of intuition, statistical analyses, and expert judgment. These have been labeled "composite models" by Conte, *et al*. Several models are listed below [Conte86].

a. COCOMO—Boehm

This is probably the best known and most thoroughly documented of all software cost estimating models. It provides three levels of models: basic, intermediate, and detailed. Boehm identifies three modes of product development—organic, semidetached, and embedded—that aid in determining the difficulty of the project. The developmental effort equations are all of the form:

$$E = a S^b m \ ,$$

where a and b are constants determined for each mode and model level;
S is the value of source LOC; and
m is a composite multiplier, determined from 15 cost-driver attributes.

Boehm suggests that the detailed model will provide cost estimates that are within 20% of actual values 70% of the time, or PRED(0.20) = 0.70 [Boehm81, Boehm84].

b. SOFTCOST—Tausworthe

Tausworthe, of the Jet Propulsion Laboratory, attempted to develop a software cost estimation model using the best features of other relatively successful models available at the time. His model incorporates the quality factors from Walson-Felix and the Rayleigh model of Putnam, among other features. It requires a total of 68 input parameters, whose values are deduced from the user's response to some 47 questions about the project. Latest reports suggest that this model has not been tested or calibrated adequately to be of general interest [Tausworthe81, Conte86].

c. SPQR Model—Jones

T. Capers Jones has developed a software cost estimation model called the Software Productivity, Quality, and Reliability (SPQR) model. The basic approach is similar to that of Boehm's COCOMO model. It is based on 20 reasonably well-defined and 25 not-so-well-defined factors that influence software costs and productivity. SPQR is a commercial product, but it is not as thoroughly documented as some other models. The computer model requires user responses to more than 100 questions about the project in order to formulate the input parameters needed to compute development costs and schedules. Jones claims that it is possible for a model such as SPQR to provide cost estimations that will come within 15% of actual values 90% of the time, or PRED(0.15) = 0.90 [Jones86].

d. COPMO—Thebaut

Thebaut proposed a software development model that attempts to account specifically for the additional effort required when teams of programmers are involved on large projects. Thus, the model is not appropriate for small projects. The general form of the equation for the effort, E, is assumed to be:

$$E = a + bS + cP^d \ ,$$

where a, b, c, and d are constants to be determined from empirical data via regression analysis;
S is the program size, in thousands of LOC; and
P is the average personnel level over the life of the project.

Unfortunately, this model requires not one but two input parameters whose actual values are not known until the project has been completed. Furthermore, the constants b and c are dependent upon the complexity class of the software, which is not easily determined. This model presents an interesting form, but it needs further development

and calibration to be of widespread interest. In view of its stage of development, no estimates of its predictive ability are in order [Conte86, Thebaut84].

e. ESTIMACS—Rubin

Rubin has developed a proprietary software estimating model that utilizes gross business specifications for its calculations. The model provides estimates of total development effort, staff requirements, cost, risk involved, and portfolio effects. At present, the model addresses only the development portion of the software life cycle, ignoring the maintenance or post-deployment phase. The ESTIMACS model addresses three important aspects of software management—estimation, planning, and control.

The ESTIMACS system includes the following modules:

- **System development effort estimator.** This module requires responses to 25 questions regarding the system to be developed, development environment, etc. It uses a database of previous project data to calculate an estimate of the development effort.
- **Staffing and cost estimator.** Inputs required are: the effort estimation from above, data on employee productivity, and salary for each skill level. Again, a database of project information is used to compute the estimate of project duration, cost, and staffing required.
- **Hardware configuration estimator.** Inputs required are: information on the operating environment for the software product, total expected transaction volume, generic application type, etc. Output is an estimate of the required hardware configuration.
- **Risk estimator.** This module calculates risk using answers to some 60 questions on project size, structure, and technology. Some of the answers are computed automatically from other information already available.
- **Portfolio analyzer.** This module provides information on the effect of this project on the total operations of the development organization. It provides the user with some understanding of the total resource demands of the projects.

The ESTIMACS system has been in use for only a short time. In the future, Rubin plans to extend the model to include the maintenance phase of the software life cycle. He claims that estimates of the total effort are within 15% of actual values [Rubin83]. The ESTIMACS model is compared with the GECOMO, JS-2, PCOC, SLIM, and SPQR/10 models in [Rubin83] and [Rubin87].

6. Reliability Models

A number of dynamic models of software defects have been developed. These models attempt to describe the occurrence of defects as a function of time, allowing one to define the reliability, R, and mean time to failure, MTTF. One example is the model described by Musa, which, like most others of this type, makes four basic assumptions:

- Test inputs are random samples from the input environment.
- All software failures are observed.
- Failure intervals are independent of each other.
- Times between failures are exponentially distributed.

Based upon these assumptions, the following relationships can be derived:

$$d(t) = D(1 - e^{-bct}) \ ,$$

where D is the total number of defects;
b, c are constants that must be determined from historical data for similar software;
$d(t)$ is the number (cumulative total) of defects discovered at time t:

$$\mathrm{MTTF}(t) = \frac{e^{bct}}{cD} \ .$$

As in many other software models, the determination of b, c and D is a nontrivial task, and yet a vitally important one for the success of the model [Ruston79, Musa75, Musa80, Musa87].

IV. Implementation of a Metrics Program

There is growing evidence, both from university research and from industry experience, that the conscientious application of software metrics can significantly improve our understanding and management of the software development process. For example, a number of software estimating models have been developed to aid in the estimation, planning, and control of software projects. Generally, these models have been developed by calibrating the estimating formulas to some existing database of previous software project information. For new projects that are not significantly different from those in the database, reasonably accurate predictions (say, ±20%) are often possible [Boehm81, Jones86]. However, numerous studies have shown that these models cannot provide good estimates for projects that may involve different environments, languages, or methodologies [Kemerer87, Rubin87]. Thus, great care must be taken in selecting a model and recalibrating it, if necessary, for the new application environment.

The selective application of software metrics to specific phases of the software development cycle can also be productive. For example, certain complexity metrics have been shown to be useful in guiding software design or redesign (maintenance) activities [Kafura87, Rombach87, Yau85].

Encouraging reports on the use of metrics are coming from industry also. A recent example is that of Grady and Caswell on the experience of Hewlett-Packard [Grady87]. They describe HP's experience implementing a corporate-wide software metrics program designed to improve project management, productivity, and product quality. The program described appears to be helping to achieve these goals, both in the short run (on individual projects) and in the long run (with improved productivity on future projects). This program may serve as a model for other organizations interested in improving their software development results. The HP experience in establishing an organizational software metrics program provides a number of useful insights, including the following:

- In addition to planning carefully for the technical operation of the metrics program, the idea of such a program must be "sold" to all individuals involved, from top management, who must find the resources to support it, to entry level programmers, who may feel threatened by it.
- Although some short-range benefits may be realized on current projects, the organization should expect to collect data for at least three years before the data are adequate for measuring long-term trends.

Outlined below is a general procedure for implementing an organizational software metrics program. The details of implementing such a program will, of course, vary significantly with the size and nature of the organization. However, all of the steps outlined are necessary, in one form or another, to achieve a successful implementation of a metrics program.

The implementation plan that follows is presented as a sequence of distinct steps. In practice, the application of this plan will probably involve some iteration between steps, just as occurs with the application of specific software development life cycle models. Although it is not stated explicitly, those responsible for establishing the metrics program must be concerned at each step with communicating the potential benefits of the program to all members of the organization and selling the organization on the merits of such a program. Unless the organization as a whole understands and enthusiastically supports the idea, the program will probably not achieve the desired results.

1. Planning Process

The implementation of a metrics program requires careful planning.

a. Defining Objectives

What are the objectives of the proposed program? What is it supposed to achieve, and how?

A specific approach which can be used to plan the software metrics program is the Goal/Question/Metric (GQM) paradigm developed by Basili, *et al*. [Basili84, Basili87]. This paradigm consists of identifying the goals to be achieved by the metrics program and associating a set of related questions with each goal. The answers to these questions should make it possible to identify the quantitative measures that are necessary to provide the answers and, thus, to reach the goals.

b. Initial Estimates of Effort and Cost

Metrics programs are not free; they may require major commitments of resources. For this reason, it is especially important to sell the idea of such a program to top management. Estimates of program costs should be made early in the planning process, even though they may be very crude, to help the organization avoid major surprises later on. These effort/cost estimates will need continuous refinement as the project proceeds.

(i) Initial Implementation

What are the costs associated with the initial start-up of the program?

(ii) Continuing Costs

Organizations must expect to incur continuing costs to operate the metrics program. These include, for example, the costs of collecting and analyzing data and of maintaining the metrics database.

2. Selection of Model and Metrics

A specific model and set of metrics is selected, based upon the objectives defined and cost considerations identified. Given a choice of several models that seem capable of meeting the objectives and cost requirements, the simplest model that is not intuitively objectionable should be chosen. The GQM paradigm provides a practical procedure for the selection of software metrics [Basili84, Basili87]. Important considerations in this selection process are the following items.

a. Projected Ability to Meet Objectives

Metrics and models available should be compared with respect to their apparent ability to meet the objectives (goals) identified.

b. Estimated Data Requirements and Cost

Models identified as capable of meeting the objectives of the organization should be compared in terms of data requirements and associated cost

of implementation. As indicated above, parsimonious models are to be preferred, if adequate.

3. Data Requirements and Database Maintenance

Once a specific model has been chosen, the data requirements and cost estimates must be spelled out in detail and refined. At this point, care must be taken to collect enough, but not too much data. Often, the initial reaction is to to collect masses of data without regard to how it will be used, "just in case it might be useful." The result is usually extinction by drowning in data. For this part of the task, the work of Basili and Weiss on collecting data for software engineering projects may be especially helpful [Basili84]. Steps include the following considerations:

a. Specific Data Required

Data must be gathered throughout the software life cycle. The specific information required at each phase must be identified.

b. Data Gathering Procedures

Once the necessary data have been identified, the specific methods and procedures for gathering the data must be described, and responsible personnel identified.

c. Database Maintenance

The database of metric data becomes an important corporate resource. Funds, procedures, and responsibilities for its maintenance must be spelled out.

d. Refined Estimates of Efforts and Costs

The information generated in the preceding steps should now make it possible to compute fairly accurate estimates of the effort and costs involved in implementing and continuing the software metrics program.

4. Initial Implementation and Use of the Model

Assuming that the above steps have been carried out successfully and that the estimated costs are acceptable, the program can now be initiated. The following items should be re-emphasized at this time:

a. Clarification of Use

The intended use of the metrics program should have been made clear early on. However, it is appropriate to restate this clearly when the program is initiated. Are the metrics to be used only for project management purposes? What about their use as tools for evaluating personnel?

b. Responsible Personnel

Specific obligations of personnel who gather, maintain, or analyze the data should be made very clear. It is impossible to collect some types of data after the project has been completed.

5. Continuing Use and Refinement

For the metrics program to be successful, it must be continuously applied and the results reviewed periodically. The following steps are involved:

a. Evaluating Results

Results should be carefully summarized and compared with what actually happened. This is often not done because "there wasn't enough time."

b. Adjusting the Model

Most models in use require a calibration process, adapting the values of multiplicative constants, etc., to the empirical data for the environment of application. Based upon the results achieved, these calibration constants should be reviewed and adjusted, if appropriate, especially over a long period of use, during which the environment itself may change significantly.

V. Trends in Software Metrics

Current trends in the software metrics area are encouraging. Metrics are being applied more widely, with good results in many cases. The limitations of existing models have been recognized, and people are becoming more realistic in their expectations of what these models can provide. There is a growing awareness that metrics programs pay off, but not without some investment of both time and resources. As the benefits of software metrics programs become more evident, the establishment of such a program will become essential for software development organizations to remain competitive in this area.

As our experience with metrics grows, better data will become available for further research. This, in turn, will make it possible to develop better metrics and models. Although it is generally still too costly to run carefully controlled experiments on large-scale software projects, better experimental data are becoming available, and for larger projects than in the past. Such data should provide better insight into the problems of large software efforts. Results already available have improved our understanding of the metrics currently in use and have provided insight into how to select better metrics.

Finally, although there are still a large number of metrics in use or under active investigation, a smaller set of metrics is emerging as having more practical utility in the measurement of the software development process. An economical set of metrics capturing the essential characteristics of software may yet emerge from this smaller, more useful set.

Software engineering is still a very young discipline. There are encouraging signs that we are beginning to understand some of the basic parameters that are most influential in the processes of software production.

Teaching Considerations

General Comments

In the past, the software metrics area has been characterized by a multitude of candidate metrics, surrounded by sometimes exaggerated and often conflicting claims. As a result, many people, especially practicing software professionals, have formed strong opinions about the validity or practicality of software metrics. Anyone intending to teach a course in this area should be aware of this controversial atmosphere. Depending upon the students involved, the instructor will have to take special care to present the material as objectively as possible, pointing out shortcomings where appropriate, but still trying to emphasize the positive potential of the field.

Textbooks

Although there is a fairly extensive literature on software metrics, textbooks are only now beginning to appear. The only one available as of fall 1988 that even begins to cover all of the topics in this module adequately is that by Conte, Dunsmore, and Shen [Conte86]. One may expect that the appearance of this text and the continuing interest and research on metrics will result in a number of new texts in this area in the next few years. Thus, anyone teaching a course in metrics should first consult with publishers for their most recent offerings.

If the instructor would like to place more emphasis on the implementation of software metrics programs, the recent book by Grady and Caswell, relating the experience of the Hewlett-Packard, might be considered as a supplementary text [Grady87].

Possible Courses

The material presented in this module may be used in various ways to meet the needs of different audiences. Depending upon the total time to be devoted to the course and upon student backgrounds, software metrics might be taught in a graduate course of 2 or 3 quarter- (or semester-) hours or in short, intensive tutorials (possibly non-credit) lasting from a few hours to a few days. Also, a unit on software metrics might be incorporated into a broader software engineering course. The objectives and prerequisites listed earlier could apply to a 2- or 3-hour credit course. Clearly, these objectives are not likely to be achieved in an intensive, shorter course. Below are described two possible courses based on the material contained in this curriculum module.

Graduate-Level University Course, 2 to 3 Quarter Hours

A graduate-level university course on software metrics could be based on this module. A lecture-based 2–quarter-hour course or a similar course augmented with a significant project and carrying 3 hours credit could cover the material presented here. It is appropriate to ask students to read some of the significant research papers on software metrics in such a course.

Coverage. For the 2–quarter-hour course, class time (20 to 25 hours) might be allocated as follows:

I. Introduction (1-2 hours)
II. Product Metrics (8-10 hours)
III. Process Metrics, Models, and Empirical Validation (8-10 hours)
IV. Implementation of a Metrics Program (2 hours)
V. Trends in Software Metrics (1 hour)

Objectives. For a 2–quarter-hour course, the first five cognitive domain objectives should be achievable. For a 3–quarter-hour course, all six objectives should be targeted. Whether or not these objectives can actually be achieved is largely a function of student background.

Prerequisites. For a 2– or 3–quarter-hour course, students should have all of the background listed under *Prerequisite Knowledge*. Students less well prepared will have difficulty in achieving all course objectives. Although not specifically noted under *Prerequisite Knowledge*, it is assumed that students have a solid mathematics background, at least through differential and integral calculus.

Intensive 4-Hour to 6-Hour Tutorial

For non-university audiences, an intensive course of 4 to 6 hours could be based on this material. It might be a non-credit offering aimed at some specific audience, such as software project managers.

Coverage. The coverage should concentrate on topics most appropriate for the particular audience. For example, project managers can be assumed to be more interested in the implementation of a metrics program than in the details of complexity metrics.

Objectives. For a course of this type, appropriate objectives might be only objectives 1 and 2 in the cognitive domain. If students have good technical backgrounds, objective 3 might also be appropriate.

Prerequisites. The background required can be reduced from that required of a student in a normal university course. For example, the statistics background might well be waived. Of course, the objectives that can be achieved depend heavily upon the background of the students.

Resources/Support Materials

Instructors should seek software tools for studying and computing software metrics. What is available will depend upon local circumstances. Many universities have software available for computing the simpler metrics, such as LOC, $v(G)$, and the Halstead metrics. However, these facilities may be difficult to use or not available in the most desirable computing environment. Thus, instructors will have to search out the best tools for their particular situations.

In addition, it is highly desirable that some computerized software metrics model be available for student experimentation. It may be possible to acquire commercially available cost-estimating tools for use in a class environment for little or no cost.

There are other resources that may also be used in presenting and discussing this material. For example, personnel from local industry who are most knowledgeable in the use and application of software metrics in their organization can be asked to provide assistance in preparing lectures or even to deliver guest lectures. Depending upon their circumstances and background, students may be asked to report on or make recommendations regarding the use of software metrics in their work environments.

Exercises

For a credit course, it is assumed that students will be assigned homework problems related to the metrics and models discussed in class. These exercises should include the use of automated tools to compute metrics for some representative examples of software, if such tools are available. However, it is essential that students do some manual calculations of metrics such as LOC, $v(G)$, and Halstead's metrics. By doing so, they will acquire a much better understanding of certain fundamental problems, for example, the difficulty in defining LOC or the counting rules for Halstead's metrics.

Depending upon their backgrounds and the time available, students might also be asked to do a project that implements or modifies a metrics computation or process model. Students might also be asked to work as a team to design a complete metrics program for implementation in some particular software development environment.

Bibliography

Albrecht83

Albrecht, A. J. and J. E. Gaffney, Jr. "Software Function, Source Lines of Code, and Development Effort Prediction: A Software Science Validation." *IEEE Trans. Software Eng. SE-9*, 6 (Nov. 1983), 639-648.

Abstract: *One of the most important problems faced by software developers and users is the prediction of the size of a programming system and its development effort. As an alternative to "size," one might deal with a measure of the "function" that the software is to perform. Albrecht [1] has developed a methodology to estimate the amount of the "function" the software is to perform, in terms of the data it is to use (absorb) and to generate (produce). The "function" is quantified as "function points," essentially, a weighted sum of the numbers of "inputs," "outputs," "master files," "inquiries" provided to, or generated by, the software. This paper demonstrates the equivalence between Albrecht's external input/output data flow representative of a program (the "function points" metric) and Halstead's [2] "software science" or "software linguistics" model of a program as well as the "soft content" variation of Halstead's model suggested by Gaffney [7].*

Further, the high degree of correlation between "function points" and the eventual "SLOC" (source lines of code) of the program, and between "function points" and the work-effort required to develop the code, is demonstrated. The "function point" measure is thought to be more useful than "SLOC" as a prediction of work effort because "function points" are relatively easily estimated from a statement of basic requirements for a program early in the development cycle.

The strong degree of equivalency between "function points" and "SLOC" shown in the paper suggests a two-step work-effort validation procedure, first using "function points" to estimate "SLOC" and then using "SLOC" to estimate the work-effort. This approach would provide validation of application development work plans and work-effort estimates early in the development cycle. The approach would also more effectively use the existing base of knowledge on producing "SLOC" until a similar base is developed for "function points."

The paper assumes that the reader is familiar with the fundamental theory of "software science" measurements and the practice of validating estimates of work-effort to design and implement software applications (programs). If not, a review of [1]-[3] is suggested.

This paper presents a comparison of SLOC and function points as predictors of software development effort, using Halstead's software science as theoretical support for the use of function points. One useful feature of this paper is an appendix (more than three pages) that provides a detailed explanation of how to apply the function point methodology.

Arthur85

Arthur, L. J. *Measuring Programmer Productivity and Software Quality.* New York: John Wiley, 1985.

Table of Contents

1 Measurement
2 Productivity
3 Quality
4 Complexity Metrics
5 Correctness Metrics
6 Efficiency Metrics
7 Flexibility Metrics
8 Integrity Metrics
9 Interoperability Metrics
10 Maintainability Metrics
11 Portability Metrics
12 Reliability Metrics
13 Reusability Metrics
14 Structure Metrics
15 Testability Metrics
16 Usability Metrics
17 Application of Software Metrics
18 Programming Style
19 IBM Assembly Language Code (ALC)
20 COBOL Metrics
21 PL/I Metrics
22 Implementing Software Measurement
Bibliography
Appendix A. ALC Reserved Words
Appendix B. COBOL Reserved Words
Appendix C. PL/I Reserved Words

The author presents a set of eleven software quality metrics, including correctness, efficiency, maintainability, and reliability. These eleven metrics are then described as functions of a more basic set of some 22 different software quality criteria. The author then discusses these metrics in some detail, with specific applications to various programming languages. This book may be of more interest to practitioners than to serious students of software metrics. There appears to be little new material, and the presentation is somewhat redundant.

Basili80

Basili, V. R. *Tutorial on Models and Metrics for Software Management and Engineering.* New York: IEEE Computer Society Press, 1980.

This is a tutorial on quantitative methods of software management and engineering. A quantitative methodology is needed to evaluate, control, and predict software development and maintenance costs. This quantitative approach allows cost, time, and quality tradeoffs to be made in a systematic manner. The tutorial focuses on numerical product-oriented measures such as size, complexity, and reliability and on resource-oriented measures such as cost, schedules, and resources. Twenty articles from software engineering literature are reprinted in this document. The articles are organized into the following sections: resource models, changes and errors, product metrics, and data collection. Successful application of the techniques, however, requires a thorough knowledge of the project under development and any assumptions made. Only then can these techniques augment good managerial and engineering judgement.

Basili84

Basili, V. R. and D. M. Weiss. "A Methodology For Collecting Valid Software Engineering Data." *IEEE Trans. Software Eng. SE-10*, 6 (Nov. 1984), 728-738.

Abstract: *An effective data collection method for evaluating software development methodologies and for studying the software development process is described. The method uses goal-directed data collection to evaluate methodologies with respect to the claims made for them. Such claims are used as a basis for defining the goals of the data analysis, defining a set of data categorization schemes, and designing a data collection form.*

The data to be collected are based on the changes made to the software during development, and are obtained when the changes are made. To ensure accuracy of the data, validation is performed concurrently with software development and data collection. Validation is based on interviews with those people supplying the data. Results from using the methodology show that data validation is a necessary part of change data collection. Without it, as much as 50 percent of the data may be erroneous.

Feasibility of the data collection methodology was demonstrated by applying it to five different projects in two different environments. The application showed that the methodology was both feasible and useful.

This article describes an effective data collection method for studying the software development process and evaluating software development methodologies. The paper describes the Goal/Questions/Metric paradigm for data collection.

Basili87

Basili, V. R. and H. D. Rombach. *TAME: Integrating Measurement into Software Environments.* TR-1764, University of Maryland, Computer Science Department, 1987.

Abstract: *Based upon a dozen years of analyzing software engineering processes and products, we propose a set of software engineering process and measurement principles. These principles lead to the view that an Integrated Software Engineering Environment (ISEE) should support multiple process models across the full software life cycle, the technical and management aspects of software engineering, and the planning, construction, and feedback and learning activities. These activities need to be tailored to the specific project under development and they must be tractable for management control. The tailorability and tractability attributes require the support of a measurement process. The measurement process needs to be top-down, based upon operationally defined goals. The TAME project uses the goal/question/metric paradigm to support this type of measurement paradigm. It provides for the establishment of project specific goals and corporate goals for planning, provides for the tracing of these goals throughout the software life cycle via feedback and post mortem analysis, and offers a mechanism for long range improvement of all aspects of software development. The TAME system automates as much of this process as possible, by supporting goal development into measurement via models and templates, providing evaluation and analysis of the development and maintenance processes, and creating and using databases of historical data and knowledge bases that incorporate experience from prior projects.*

Ten software process principles and fourteen software measurement principles, based upon a dozen years of research in the area, are presented. The Goal/Questions/Metric paradigm for designing software measurement systems is also discussed. TAME stands for Tailoring A Measurement Environment.

Behrens83

Behrens, C. A. "Measuring the Productivity of Computer Systems Development Activities with Function Points." *IEEE Trans. Software Eng. SE-9*, 6 (Nov. 1983), 648-652.

Abstract: *The function point method of measuring application development productivity developed by Albrecht is reviewed and a productivity improvement measure introduced. The measurement methodology is then applied to 24 development projects.*

Size, environment, and language effects on productivity are examined. The concept of a productivity index which removes size effects is defined and an analysis of the statistical significance of results is presented.

This is a report of a relatively successful attempt to correlate function point values with productivity and effort values in a production environment.

Boehm76

Boehm, B. W., J. R. Brown, and M. Lipow. "Quantitative Evaluation of Software Quality." *Proc. 2nd Intl. Conf. on Software Engineering.* Long Beach, Calif.: IEEE Computer Society, Oct. 1976, 592-605. Reprinted in [Basili80], 218-231.

Abstract: *The study reported in this paper establishes a conceptual framework and some key initial results in the analysis of the characteristics of software quality.*

The software quality characteristics delineated in this article are also discussed in [Perlis81], where they are compared to those of McCall, *et al.*

Boehm81

Boehm, B. W. *Software Engineering Economics.* Englewood Cliffs, N. J.: Prentice-Hall, 1981.

Table of Contents

This is a classic text on software engineering economics. It presents an excellent, detailed discussion of the use of selected software metrics in one particular software development process model, *i.e.*, COCOMO, which was developed by the author. Otherwise it is not appropriate as a text for this module; its scope is much too limited, and the book is now somewhat out of date.

Boehm84

Boehm, B. W. "Software Engineering Economics." *IEEE Trans. Software Eng. SE-10*, 1 (Jan. 1984), 4-21.

Abstract: *This paper summarizes the current state of the art and recent trends in software engineering economics. It provides an overview of economic analysis techniques and their applicability to software engineering and management. It surveys the field of software cost estimation, including the major estimation techniques available, the state of the art in algorithmic cost models, and the outstanding research issues in software cost estimation.*

The cost estimation techniques identified are: algorithmic models, expert judgment, analogy, Parkinson's principle, price-to-win, top-down, and bottom-up. Although Parkinson's principle and price-to-win are identified as unacceptable methods, it is acknowledged that, of the other methods, none

is demonstrably superior. Thus, since the methods tend to complement one another, best results will probably come from using some combination of the other techniques. The following algorithmic cost estimation models are discussed: Putnam's SLIM, The Doty Model, RCA PRICE S, COCOMO, Bailey-Basili, Grumman SOFTCOST, Tausworthe Deep Space Network (DSN) model, and the Jensen model. Finally, the author identifies seven major issues needing further research—including size estimation, size and complexity metrics, cost-driver attributes, cost model analysis and refinement, models of project dynamics, models for software evolution, and software data collection.

Card87a

Card, D. N. and W. W. Agresti. "Resolving the Software Science Anomaly." *J. Syst. and Software 7*, 1 (March 1987), 29-35.

Abstract: *The theory of software science proposed by Halstead appears to provide a comprehensive model of the program construction process. Although software science has been widely criticized on theoretical grounds, its measures continue to be used because of apparently strong empirical support. This study reexamined one basic relationship proposed by the theory: that between estimated and actual program length. The results show that the apparent agreement between these quantities is a mathematic artifact. Analyses of both Halstead's own data and another larger data set confirm this conclusion. Software science has neither a firm theoretical nor empirical foundation.*

The anomaly referred to in the title is that although a high correlation between the actual (observed) program length and estimated (calculated) program length appears to be supported by empirical studies, no solid theoretical basis has been established for such a relationship. The authors resolve the anomaly by demonstrating that the two quantities are defined in such a way that one is mathematically dependent upon the other. Thus, the strong empirical support previously reported apparently has not been established either.

Card87b

Card, D. N. and W. W. Agresti. "Comments on Resolving the Software Science Anomaly." *J. Syst. and Software 7*, 1 (March 1987), 83-84.

Abstract: *Refer to the abstract for* [Card87a]

The authors offer a rationale for [Card87a], pointing out that users of software analysis tools based upon software science metrics may not be aware—but should be—of the lack of theoretical and empirical basis for these metrics.

Card88

Card, D. N. and W. W. Agresti. "Measuring Software Design Complexity." *J. Syst. and Software 8*, 3 (June 1988), 185-197.

Abstract: *Architectural design complexity derives from two sources: structural (or intermodule) complexity and local (or intramodule) complexity. These complexity attributes can be defined in terms of functions of the number of I/O variables and fanout of the modules comprising the design. A complexity indicator based on these measures showed good agreement with a subjective assessment of design quality but even better agreement with an objective measure of software error rate. Although based on a study of only eight medium-scale scientific projects, the data strongly support the value of the proposed complexity measure in this context. Furthermore, graphic representations of the software designs demonstrate structural differences corresponding to the results of the numerical complexity analysis. The proposed complexity indicator seems likely to be a useful tool for evaluating design quality before committing the design to code.*

The measure proposed by the authors expresses the total complexity of a software design as the sum of the structural (intermodule) complexity and the local (intramodule) complexity. The number of modules, number of I/O variables, and degree of fanout are important factors in determining the complexity. An important consideration for this metric is that all the required information is available at design time and before code is produced. The approach is similar to that described in [Harrison87].

Cerino86

Cerino, D. A. "Software Quality Measurement Tools And Techniques." *Proc. COMPSAC 86.* Washington, D. C.: IEEE Computer Society, Oct. 1986, 160-167.

Abstract: *This paper describes research being performed by RADC, to develop quality measurement computer based tools to support quality evaluation during each activity of the software life cycle. Current work has provided a baseline quality measurement tool to monitor the overall quality and resource expenditures of developing software by collecting (semi-automated), storing, and analyzing software measurement data for software acquisition and software project personnel. This tool is being used in the prediction and assessment of developing software. In the future, this tool will evolve into a metrics researcher's workbench tuned for software development personnel and will be completely automated. Efforts are also underway to specify the data collection mechanisms which can be embedded within software engineering environment tools. All three approaches are presented.*

This paper reports on work being done at Rome Air Development Center to develop automated tools to support software quality evaluation during each activity of the development life cycle.

Christensen81

Christensen, K., G. P. Fitsos, and C. P. Smith. "A Perspective on Software Science." *IBM Systems J. 20*, 4 (1981), 372-387.

Abstract: *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.*

The authors review Halstead's software science. They conclude that software science "offers a methodology not only for making measurements, but also for calibrating the measuring instruments."

Conte86

Conte, S. D., H. E. Dunsmore, and V. Y. Shen. *Software Engineering Metrics and Models.* Menlo Park, Calif.: Benjamin/Cummings, 1986.

Table of Contents

The basic outline of this book is similar to that of this module. It is intended to be used as a textbook, and covers most of the topics shown in the module outline.

Cote88

Cote, V., P. Bourque, S. Oligny, and N. Rivard. "Software Metrics: An Overview of Recent Results." *J. Syst. and Software 8*, 2 (March 1988), 121-131.

Abstract: *The groundwork for software metrics was established in the seventies, and from these earlier works, interesting results have emerged in the eighties. Over 120 of the many publications on software metrics that have appeared since 1980 are classified and presented in five tables that comprise, respectively, (1) the use of classic metrics, (2) a description of new metrics, (3) software metrics through the life cycle, (4) code metrics and popular programming languages, and (5) various metric-based estimation models.*

This is an excellent overview of the software metrics literature, especially for the period 1981 through 1986. It cites and classifies over 120 publications. Six classic papers prior to 1981 are also included, beginning with McCabe's 1976 paper. Especially with the five tables described above, this paper should prove invaluable to anyone interested in consulting the literature for this period.

Coulter83

Coulter, N. S. "Software Science and Cognitive Psychology." *IEEE Trans. Software Eng. SE-9*, 2 (March 1983), 166-171.

Abstract: *Halstead proposed a methodology for studying the process of programming known as software science. This methodology merges theories from cognitive psychology with theories from computer science. There is evidence that some of the assumptions of software science incorrectly apply the results of cognitive psychology studies. Halstead proposed theories relative to human memory models that appear to be without support from psychologists. Other software scientists, however, report empirical evidence that may support some of those theories. This anomaly places aspects of software science in a precarious position. The three conflicting issues discussed in this paper are 1) limitations of short-term memory and a number of subroutine parameters, 2) searches in human memory and programming effort, and 3) psychological time and programming time.*

This paper is a review of Halstead's theory, and critical discussion of Halstead's use of relevant theories from the field of psychology.

Curtis79a

Curtis, B., S. B. Sheppard, P. Milliman, M. A. Borst, and T. Love. "Measuring the Psychological Complexity of Software Maintenance Tasks with the Halstead and McCabe Metrics." *IEEE Trans. Software Eng. SE-5*, 2 (March 1979), 96-104.

Abstract: *Three software complexity measures (Halstead's* E*, McCabe's* v(G)*, and the length as measured by a number of statements) were compared to a programmer performance on two software maintenance tasks. In an experiment on understanding, length and* v(G) *correlated with the percent of statements correctly recalled. In an ex-*

periment on modification, most significant correlations were obtained with metrics computed on modified rather than unmodified code. All three metrics correlated with both the accuracy of the modification and the time to completion. Relationships in both experiments occurred primarily in unstructured rather than structured code, and in code with no comments. The metrics were also most predictive of performance for less experienced programmers. Thus, these metrics appear to assess psychological complexity primarily where programming practices do not provide assistance in understanding the code.

This paper investigates the extent to which the Halstead (*E*) and McCabe (*v*(*G*)) metrics assess the psychological complexity of understanding and modifying software. The authors claim that "Halstead's metric ... was proposed as an absolute measure of psychological complexity (i.e., number of mental discriminations)." Furthermore, McCabe's measure, although not formulated in psychological terms, "may prove to be a correlated measure of psychological complexity." Two experiments were performed, using professional programmers: 1) understanding an existing program and 2) accurately implementing modifications to it. Each experiment involved 36 programmers with an average of more than 5 years of professional experience. Results in the first experiment indicated that the Halstead and McCabe metrics correlated well with each other (0.84), but not with LOC (0.47, 0.64). Correlations with measured performances were not as high, ranging from -0.10 for *E*, to -0.61 for LOC. After adjustments in the data, correlations were -0.73 (*E*), -0.21 (*v*(*G*)), and -0.65 (LOC). In the second experiment, results indicated that all three metrics correlated well with each other (0.85 to 0.97). By comparison, correlations with performance were not high (< 0.57), but the authors claim that "their magnitudes are typical of significant results reported in human factors experiments."

Curtis79b

Curtis, B., S. B. Sheppard, and P. Milliman. "Third Time Charm: Stronger Prediction of Programmer Performance by Software Complexity Metrics." *Proc. 4th Int. Conf. on Software Engineering.* New York: IEEE, Sept. 1979, 356-360.

Abstract: *This experiment is the third in a series investigating characteristics of software which are related to its psychological complexity. A major focus of this research has been to validate the use of software complexity metrics for predicting programmer performance. In this experiment we improved experimental procedures which produced only modest results in the previous two studies. The experimental task required 54 experienced Fortran programmers to locate a single bug in each of three programs. Performance was measured by the time to locate and successfully correct the bug. Much stronger results were obtained than in earlier studies. Halstead's* E *proved to be the best predictor of performance, followed by McCabe's* v(G) *and the number of lines of code.*

This paper is a report on the third in a series of experiments on software complexity metrics, specifically McCabe's *v*(*G*), Halstead's *E*, and LOC. Intercorrelations of metrics, when applied at the subroutine level, were: 0.92 for *E*:*v*, 0.89 for LOC:*E* and 0.81 for LOC:*v*. Intercorrelations of metrics, when applied at the program level, were : 0.76 for *E*:*v*, 0.56 for LOC:*E* and 0.90 for LOC:*v*. Correlations of these metrics with measured performances ranged from 0.52 to 0.75. These results are considerably better than those attained in previous experiments, *e.g.*, as reported in [Curtis79a].

DeMarco82

DeMarco, T. *Controlling Software Projects: Management, Measurement & Estimation.* New York: Yourdon Press, 1982.

Table of Contents

Appendix B. Notational Conventions for Specification and Design Models
Appendix C. A Tailored Primer on Statistics
Appendix D. Sample Program to Compute Code Volume

This is primarily a book on software project management. However, it recognizes the importance of models and metrics in this process, and much of the book deals with these topics. Of particular interest is the development of specification metrics that are available early in the development cycle.

Elshoff76

Elshoff, J. L. "Measuring Commercial PL/I Programs Using Halstead's Criteria." *ACM SIGPLAN Notices 11*, 5 (May 1976), 38-46.

Abstract: *In 1972 Halstead first reported his investigation into natural laws of algorithm analogous to laws of natural or physical sciences. The basic idea is to separate the physical structure of algorithms from the logical structure of algorithms. His theory has been refined since that time. Furthermore, the theory has been applied to algorithms in different languages and different environments. In this study, Halstead's criteria are applied to 154 PL/I programs. This sample contains the largest algorithms to be measured by his methods to date. A subset of 120 of the programs has been measured previously by other techniques which describe the basic attributes of the programs herein discussed.*

The correlation between observed program length, *N*, and calculated program length is investigated. Of the 154 programs, 34 have been developed using structured programming techniques, while the other 120 were not. Correlations between observed and calculated values of *N* are reported to be 0.985 (for the structured programs) and 0.976, respectively.

Elshoff78

Elshoff, J. L. "An Investigation into the Effects of the Counting Method Used on Software Science Measurements." *ACM SIGPLAN Notices 13*, 2 (Feb. 1978), 30-45.

Abstract: *Professor Maurice Halstead of Purdue University first defined a set of properties of algorithms in 1972. The properties are defined in terms of the number of unique operators, unique operands, total operators, and total operands used to express the algorithm. Since 1972, independent experiments have measured various sets of algorithms and have supported Halstead's theories concerning these properties. Also, new properties have been defined and experiments performed to study them.*

This paper reports a study in which different methods of counting operators and operands are applied to a fixed set of 34 algorithms written in PL/I. Some properties of the algorithms vary significantly depending on the counting method chosen; other properties remain stable. Although no one counting method can be shown to be best, the results do indicate the importance of the counting method to the overall measurement of an algorithm. Moreover, the results provide a reminder of how sensitive some of the measurements are and of how careful researchers must be when drawing conclusions from software science measurements.

The author investigates the effect of variations in counting methods. Eight different counting methods were applied to 34 different PL/I programs. Results: Length (*N*) and volume (*V*) are relatively insensitive, while level (*L*) and effort (*E*) are much more sensitive to the counting method.

Ferrari86

Ferrari, D. "Considerations on the Insularity of Performance Evaluation." *IEEE Trans. Software Eng. SE-12*, 6 (June 1986), 678-683.

Abstract: *It is argued that systems performance evaluation, in the first 20 years of its existence, has developed in substantial isolation from such disciplines as computer architecture, system organization, operating systems, and software engineering. The possible causes for this phenomenon, which seems to be unique in the history of engineering, are explored. Its positive and negative effects on computer science and technology, as well as on performance evaluation itself, are discussed. The drawbacks of isolated development outweigh its advantages. Thus, instructional and research initiatives to foster the rapid integration of the performance evaluation viewpoint into the mainstream of computer science and engineering are proposed.*

This article discusses the degree of isolation of performance evaluation studies from other computer science/software engineering activities. Although performance evaluation is now considered a separate field, the author questions whether this is desirable and suggests that performance evaluation considerations should be introduced into computer science and engineering courses in general.

Fitzsimmons78

Fitzsimmons, A. and T. Love. "A Review and Evaluation of Software Science." *ACM Computing Surveys 10*, 1 (March 1978), 3-18.

Abstract: *During recent years, there have been many attempts to define and measure the "complexity" of a computer program. Maurice Halstead has developed a theory that gives objective measures of software complexity. Various studies and experiments have shown that the theory's predictions of*

the number of bugs in programs and of the time required to implement a program are amazingly accurate. It is a promising theory worthy of much more probing scientific investigation.

This paper reviews the theory, called "software science," and the evidence supporting it. A brief description of a related theory, called "software physics," is included.

This article is one of the earliest published critical reviews of Halstead's work on software science.

Grady87

Grady, R. B. and D. R. Caswell. *Software Metrics: Establishing a Company-Wide Program.* Englewood Cliffs, N. J.: Prentice-Hall, 1987.

Table of Contents

This book describes the experience of the Hewlett-Packard Company in setting up a company-wide program of software metrics designed to improve the management, productivity, and quality of the software development process. Enough detail is presented that this book should prove useful to other organizations seriously contemplating establishing a software metrics program.

Halstead77

Halstead, M. H. *Elements of Software Science.* New York: Elsevier North-Holland, 1977.

Table of Contents

This book is a classic in software metrics, the original book by Halstead expounding the principles of software science. Principal attractions of the theory as presented here are its high degree of agreement with selected empirical data and its distinction of providing a unified theory of software metrics. Unfortunately, a number of later works have pointed out several difficulties in the formulation of the theory and its empirical validation, *e.g.*, see [Shen83] and [Card87a].

Harrison82

Harrison, W., K. Magel, R. Kluczny, and A. DeKock. "Applying Software Complexity Metrics to Program Maintenance." *Computer 15*, 9 (Sept. 1982), 65-79.

Abstract: *The authors find that predicting software complexity can save millions in maintenance costs, but while current measures can be used to some degree, most are not sufficiently sensitive or comprehensive. They examine some complexity metrics in use.*

This is primarily a survey of more than a dozen complexity measures currently in use. Despite the article's title, little guidance is given on how to apply these to the software maintenance area.

Harrison87

Harrison, W. and C. Cook. "A Micro/Macro Measure of Software Complexity." *J. Syst. and Software 7*, 3 (Sept. 1987), 213-219.

Abstract: *A software complexity metric is a quantitative measure of the difficulty of comprehending and working with a specific piece of software. The majority of metrics currently in use focus on a program's "microcomplexity." This refers to how difficult the details of the software are to deal with. This paper proposes a method of measuring the*

"macrocomplexity," i.e., how difficult the overall structure of the software is to deal with, as well as the microcomplexity. We evaluate this metric using data obtained during the development of a compiler/environment project, involving over 30,000 lines of C code. The new metric's performance is compared to the performance of several other popular metrics, with mixed results. We then discuss how these metrics, or any other metrics, may be used to help increase the project management efficiency.

The authors propose a software complexity metric incorporating both the micro (intra–sub-program level) and macro (inter-program level) complexity contributed by each subprogram. The metric (MMC) is compared with other metrics such as those of Hall and Preisser, Henry and Kafura, McCabe, Halstead, lines of code, and number of procedures. The new metric correlated better (0.82) with the basic error rates than the other five metrics. However, in identifying software modules with exceptional error rates, it did little better than the other metrics, and slightly worse than DSLOC.

Henry81

Henry, S., D. Kafura, and K. Harris. "On the Relationships Among Three Software Metrics." *Performance Eval. Rev. 10*, 1 (Spring 1981), 81-88.

Abstract: *Automatable metrics of software quality appear to have numerous advantages in the design, construction and maintenance of software systems. While numerous such metrics have been defined, and several of them have been validated on actual systems, significant work remains to be done to establish the relationships among these metrics. This paper reports the results of correlation studies made among three complexity metrics which were applied to the same software system. The three complexity metrics used were Halstead's effort, McCabe's cyclomatic complexity and Henry and Kafura's information flow complexity. The common software system was the UNIX operating system. The primary result of this study is that Halstead's and McCabe's metrics are highly correlated while the information flow metric appears to be an independent measure of complexity.*

The results of this study show a high correlation of all three metrics with the number of errors in the software: 0.89 for Halstead's *E*, 0.95 for information flow, and 0.96 for McCabe's metric. In addition, Halstead's metric and McCabe's metrics appear to be highly related to one another (0.84). However, the information flow metric correlates poorly with either the Halstead (0.38) or the McCabe (0.35) metric. This may indicate that while all three metrics are reasonable predictors of error rates, the information flow metric is somewhat orthogonal to the other two complexity metrics.

Henry84

Henry, S. and D. Kafura. "The Evaluation of Software Systems' Structure Using Quantitative Software Metrics." *Software—Practice and Experience 14*, 6 (June 1984), 561-573.

Abstract: *The design and analysis of the structure of software systems has typically been based on purely qualitative grounds. In this paper we report on our positive experience with a set of quantitative measures of software structure. These metrics, based on the number of possible paths of information flow through a given component, were used to evaluate the design and implementation of a software system (the UNIX operating system kernel) which exhibits the interconnectivity of components typical of large-scale software systems. Several examples are presented which show the power of this technique in locating a variety of both design and implementation defects. Suggested repairs, which agree with the commonly accepted principles of structured design and programming, are presented. The effect of these alterations on the structure of the system and the quantitative measurements of that structure lead to a convincing validation of the utility of information flow metrics.*

This is an important paper, in the sense that the information flow metric developed is shown to be related to software complexities and thus to potential problem areas of the UNIX operating system. This information is then used to guide efforts to redesign those portions of the system that appear to be overly complex. Of special note here is the fact that these information flow metrics may be computed and utilized in the software design process, prior to the generation of any program code.

Jones84

Jones, T. C. "Reusability in Programming: A Survey of the State of the Art." *IEEE Trans. Software Eng. SE-10*, 5 (Sept. 1984), 488-494.

Abstract: *As programming passes the 30 year mark as a professional occupation, an increasingly large number of programs are in application areas that have been automated for many years. This fact is changing the technology base of commercial programming, and is opening up new markets for standard functions, reusable common systems, modules, and the tools and support needed to facilitate searching out and incorporating existing code segments. This report addresses the 1984 state of the art in the domains of reusable design, common systems, reusable programs, and reusable modules or subroutines. If current trends toward reusability continue, the amount of reused logic and reused code in commercial programming systems may approach 50 percent by 1990. However, major efforts will be needed in the areas of reusable data, reus-*

able architectures, and reusable design before reusable code becomes a sound basic technology.

The author includes interesting statistics on programmer and software populations in a survey of the current status of this possible key to programming productivity.

Jones86

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

Table of Contents

This book is primarily a study of programming productivity, especially as it might be predicted by the Software Productivity, Quality, and Reliability (SPQR) model developed by the author. It enumerates a total of 20 major and 25 less significant factors that influence productivity, many of which are input to the SPQR model. This book does not possess sufficient breadth in software metrics to serve as a text for this module, but does contain illuminating discussions of some currently used metrics and problems associated with them.

Kafura81

Kafura, D. and S. Henry. "Software Quality Metrics Based on Interconnectivity." *J. Syst. and Software 2*, 2 (June 1981), 121-131.

Abstract: *States a set of criteria that has guided the development of a metric system for measuring the quality of a large-scale software product. This metric system uses the flow of information within the system as an index of system interconnectivity. Based on this observed interconnectivity, a variety of software metrics can be defined. The types of software quality features that can be measured by this approach are summarized. The data-flow analysis techniques used to establish the paths of information flow are explained and illustrated. Finally, a means of integrating various metrics and models into a comprehensive software development environment is discussed. This possible integration is explained in terms of the Gandalf system currently under development at Carnegie Mellon University.*

The authors propose a quality metric for large-scale software products, using the program information flow as a measure of system interconnectivity. Results of application to UNIX systems are discussed.

Kafura85

Kafura, D. and J. Canning. "A Validation of Software Metrics Using Many Metrics and Two Resources." *Proc. 8th Intl. Conf. on Software Engineering.* Washington, D. C.: IEEE Computer Society Press, 1985, 378-385.

Abstract: *In this paper are presented the results of a study in which several production software systems are analyzed using ten software metrics. The ten metrics include both measures of code details, measures of structure, and combinations of these two. Historical data recording the number of errors and the coding time of each component are used as objective measures of resource expenditure of each component. The metrics are validated by showing: (1) the metrics singly and in combination are useful indicators of those components which require the most resources, (2) clear patterns between the metrics and the resources expended are visible when both resources are accounted for, (3) measures of the structure are as valuable in examining software systems as measures of code details, and (4) the choice of which, or how many, software metrics to employ in practice is suggested by measures of "yield" and "coverage".*

The code metrics used were LOC, Halstead's *E*, and McCabe's *v*(*G*). Structure metrics used were Henry and Kafura's information flow, McClure's invocation complexity, Woodfield's review complexity, and Yau and Collofello's stability measure. The three hybrid measures were combinations of LOC with the metrics of Henry and Kafura, Woodfield, and Yau and Collofello, respectively. The authors conclude that "The interplay between and among the resources and factors is too subtle and fluid to be observed accurately by a single metric, or a single resource."

Kafura87

Kafura, D. and G. R. Reddy. "The Use of Software Complexity Metrics in Software Maintenance." *IEEE Trans. Software Eng. SE-13*, 3 (March 1987), 335-343.

Abstract: *This paper reports on a modest study which relates seven different software complexity metrics to the experience of maintenance activities performed on a medium size software system. The seven metrics studied are the same as those used in [Kafura85]. The software system involved is a single user relational database system, written in Fortran. Three different versions of the software system that evolved over a period of three years were analyzed in this study. A major revision of the system, while still in its design phase, was also analyzed.*

The results of the study indicate: 1) that the growth in system complexity as determined by the software metrics agree with the general character of the

maintenance tasks performed in successive versions; 2) the metrics were able to identify the improper integration of functional enhancements made to the system; 3) the complexity values of the system components as indicated by the metrics conform well to an understanding of the system by people familiar with the system; 4) an analysis of the redesigned version of the system showed the usefulness of software metrics in the (re)design phase by revealing a poorly structured component of the system.

This paper reports on a study of seven different complexity metrics as related to experience in software maintenance. The research involved three different versions of a medium-sized system that evolved over a period of three years. Conclusions include the statement that the metrics were able to identify the improper integration of functional enhancements made to the system.

Kemerer87

Kemerer, C. F. "An Empirical Validation of Software Cost Estimation Models." *Comm. ACM 30*, 5 (May 1987), 416-429.

Abstract: *Practitioners have expressed concern over their inability to accurately estimate costs associated with software development. This concern has become even more pressing as costs associated with development continue to increase. As a result, considerable research attention is now directed at gaining a better understanding of the software-development process as well as constructing and evaluating software cost estimating tools. This paper evaluates four of the most popular algorithmic models used to estimate software costs (*SLIM, COCOMO, *Function Points, and* ESTIMACS*). Data on 15 large completed business data-processing projects were collected and used to test the accuracy of the models'* ex post *effort estimation. One important result was that Albrecht's Function Points effort estimation model was validated by the independent data provided in this study [3]. The models not developed in business data-processing environments showed significant need for calibration. As models of the software-development process, all of the models tested failed to sufficiently reflect the underlying factors affecting productivity. Further research will be required to develop understanding in this area.*

The author compares results of four cost estimation models on a set of 15 large (average of 200 KSLOC) software products, all developed by the ABC consulting firm (anonymous). The models compared were Boehm's COCOMO, Putnam's SLIM, Albrecht's Function Points (FP), and Rubin's ESTIMACS. Although the models were developed and calibrated with very different data, the author seems surprised that the resulting errors in predicted person-months are large (COCOMO 600%, SLIM 771%, FP 102%, ESTIMACS 85%). The author concludes that models developed in different environments do not work well without recalibration for the environment where they are to be applied.

Knafl86

Knafl, G. J. and J. Sacks. "Software Development Effort Prediction Based on Function Points." *Proc. COMPSAC 86.* Washington, D. C.: IEEE Computer Society Press, Oct. 1986, 319-325.

Abstract: *We analyze a published data set used to predict future software development effort in terms of function points. For a full range of COBOL project sizes, a straight line model is inappropriate as is a linear regression model using the software science transform of function points. Confidence bands based on alternate robust models show the untenability of the straight line model. Acceptable uncertainty levels require large prediction bands indicating that function points by itself is insufficient for precise prediction.*

The authors analyze the data set used by Albrecht and Gaffney [Albrecht83]. Their conclusion is that the function point measure by itself is insufficient for precise prediction.

Lassez81

Lassez, J.-L., D. Van der Knijff, J. Shepherd, and C. Lassez. "A Critical Examination of Software Science." *J. Syst. and Software 2*, 2 (June 1981), 105-112.

Abstract: *The claims that software science could provide an empirical basis for the rationalization of all forms of algorithm description are shown to be invalid from a formal point of view. In particular, the conjectured dichotomy between operators and operands is shown not to hold over a wide class of languages. An experiment that investigated discrepancies between the level measure and its estimator is described to show that its failure was due to shortcomings in the theory. One cannot obtain reliable results without tampering with both measure and estimator definitions.*

This paper is a critical analysis of Halstead's theory. The authors conclude that his fundamental hypotheses are not applicable over the broad range claimed by Halstead.

Levitin86

Levitin, A. V. "How To Measure Software Size, and How Not To." *Proc. COMPSAC 86.* Washington, D. C.: IEEE Computer Society Press, Oct. 1986, 314-318.

Abstract: *The paper suggests a list of criteria desirable for a measure of software size. The principal known size metrics - source lines of code, the number of statements, Software Science length and volume, and the number of tokens -- are discussed from the standpoint of these general criteria. The analysis indicates that the number of tokens is superior over the other, much more often used metrics.*

Levitin compares common size metrics, such as LOC, number of statements, and Halstead's measures *n*, *N*, and *V*. He reports that *n* (the number of tokens) is superior to the other metrics as a measure of size.

Li87

Li, H. F. and W. K. Cheung. "An Empirical Study of Software Metrics." *IEEE Trans. Software Eng. SE-13*, 6 (June 1987), 697-708.

Abstract: *Software metrics are computed for the purpose of evaluating certain characteristics of the software developed. A Fortran static source code analyzer, FORTRANAL, was developed to study 31 metrics, including a new hybrid metric introduced in this paper, and applied to a database of 255 programs, all of which were student assignments. Comparisons among these metrics are performed. Their cross-correlation confirms the internal consistency of some of these metrics which belong to the same class. To remedy the incompleteness of most of these metrics, the proposed metric incorporates context sensitivity to structural attributes extracted from a flow graph. It is also concluded that many volume metrics have similar performance while some control metrics surprisingly correlate well with typical volume metrics in the test samples used. A flexible class of hybrid metric can incorporate both volume and control attributes in assessing software complexity.*

The authors report on a study of 31 different complexity metrics applied to a database of 255 FORTRAN programs (all student assignments). They claim that all the other metrics are incomplete and propose a new hybrid metric to fill the gap. The article includes an interesting classification of complexity metrics, shown in the form of a chart.

Lister82

Lister, A. M. "Software Science—The Emperor's New Clothes?" *Australian Computer J. 14*, 2 (May 1982), 66-70.

Abstract: *The emergent field of software science has recently received so much publicity that it seems appropriate to pose the question above. This paper attempts to provide an answer by examining the methodology of software science, and by pointing out apparent anomalies in three major areas: the length equation, the notion of potential volume, and the notion of language level. The paper concludes that the emperor is in urgent need of a good tailor.*

A critical review of Halstead's results and the 1978 review of the same by Fitzsimmons and Love. Halstead's results for *N*, *V**, *L*, and *E* are all criticized.

McCabe76

McCabe, T. J. "A Complexity Measure." *IEEE Trans. Software Eng. SE-2*, 4 (Dec. 1976), 308-320.

Abstract: *This paper describes a graph-theoretic complexity measure and illustrates how it can be used to manage and control program complexity. The paper first explains how the graph-theory concepts apply and gives an intuitive explanation of the graph concepts in programming terms. The control graphs of several actual Fortran programs are then presented to illustrate the correlation between intuitive complexity and the graph-theoretic complexity. Several properties of the graph-theoretic complexity are then proved which show, for example, that complexity is independent of physical size (adding or subtracting functional statements leaves complexity unchanged) and complexity depends only on the decision structure of a program. The issue of using nonstructured control flow is also discussed. A characterization of nonstructured control graphs is given and a method of measuring the "structuredness" of a program is developed. The relationship between structure and reducibility is illustrated with several examples.*

The last section of this paper deals with a testing methodology used in conjunction with the complexity measure; a testing strategy is defined that dictates that a program can either admit of a certain minimal testing level or the program can be structurally reduced.

McCabe's classic paper on the cyclomatic complexity of a computer program. This is an excellent paper. The contents are well-described by the abstract.

McCall77

McCall, J. A., P. K. Richards, and G. F. Walters. *Factors in Software Quality, Vol. I, II, III: Final Tech. Report.* RADC-TR-77-369, Rome Air Development Center, Air Force Systems Command, Griffiss Air Force Base, N. Y., 1977.

This is one of the earliest, often-referenced works on software quality factors. The quality characteristics identified in this report are also discussed in [Perlis81], pages 204-206, where they are compared to those of Boehm *et al.*

Mohanty81

Mohanty, S. N. "Software Cost Estimation: Present and Future." *Software—Practice and Experience 11*, 2 (Feb. 1981), 103-121.

> ***Abstract:*** *The state-of-the-art in software cost estimation is reviewed. The estimated cost of a software system varies widely with the model used. Some variation in cost estimation is attributable to the anomalies in the cost data base used in developing the model. The other variations, it is claimed are due to the presence or absence of certain 'qualities' in the final product. These qualities are measures of 'goodness' in design, development and test-integration phases of software. To consider quality as a driver of software cost, the author suggests an association between cost and quality and proposes a way to use quality metrics to estimate software cost.*
>
> Mohanty reviews the state-of-the-art in software cost estimation. More than 15 models are discussed, including those of Wolverton, Price-S, and Walston/Felix. The author lists 49 factors that influence software development costs.

Musa75

Musa, J. D. "A Theory of Software Reliability and Its Application." *IEEE Trans. Software Eng. 1*, 3 (Sept. 1975), 312-327. Reprinted in [Basili80], 194-212.

> ***Abstract:*** *An approach to a theory of software reliability based on execution time is derived. This approach provides a model that is simple, intuitively appealing, and immediately useful. The theory permits the estimation, in advance of a project, of the amount of testing in terms of execution time required to achieve a specified reliability goal [stated as a mean time to failure (MTTF)]. Execution time can then be related to calendar time, permitting a schedule to be developed. Estimates of execution time and calendar time remaining until the reliability goal is attained can be continually remade as testing proceeds, based only on the length of execution time intervals between failures. The current MTTF and the number of errors remaining can also be estimated. Maximum likelihood estimation is employed, and confidence intervals are also established. The foregoing information is obviously very valuable in scheduling and monitoring the progress of program testing. A program has been implemented to compute the foregoing quantities. The reliability model that has been developed can be used in making system tradeoffs involving software or software and hardware components. It also provides a soundly based unit of measure for the comparative evaluation of various programming techniques that are expected to enhance reliability. The model has been applied to four medium-sized development projects, all of which have completed their life cycles. Measurements taken of MTTF during operation agree well with the predictions made at the end of system test. As far as the author can determine, these are the first times that a software reliability model was used* during *software development projects. The paper reflects and incorporates the practical experience gained.*
>
> The author develops the basic concept of software reliability and discusses its application to actual projects. This is one of the early papers by this author on this subject. Later work is reported in [Musa80] and [Musa87].

Musa80

Musa, J. D. "Software Reliability Measurement." *J. Syst. and Software 1*, 3 (1980), 223-241. Reprinted in [Basili80], 194-212.

> ***Abstract:*** *The quantification of software reliability is needed for the system engineering of products involving computer programs and the scheduling and monitoring of software development. It is also valuable for the comparative evaluation of the effectiveness of various design, coding, testing, and documentation techniques. This paper outlines a theory of software reliability based on execution or CPU time, and a concomitant model of the testing and debugging process that permits execution time to be related to calendar time. The estimation of parameters of the model is discussed. Application of the theory in scheduling and monitoring software projects is described, and data taken from several actual projects are presented.*
>
> This paper further develops the basic concepts of software reliability and its measurement. The author has developed these concepts much more fully in [Musa87].

Musa87

Musa, J. D., A. Iannino, and K. Okumoto. *Software Reliability: Measurement, Prediction, Application.* New York: McGraw-Hill, 1987.

> ***Table of Contents***
>
> *Part I. Overview*
> *1 Introduction to Software Reliability*
> *2 Selected Models*
> *3 Applications*
> *Part II. Practical Applications*
> *4 System Definition*
> *5 Parameter Determination*
> *6 Project-Specific Techniques*
> *7 Application Procedures*
> *8 Implementation Planning*
> *Part III. Theory*
> *9 Software Reliability Modeling*

10 Markovian Models
11 Descriptions of Specific Models
12 Parameter Estimation
13 Comparison of Software Reliability Models
14 Calendar Time Modeling
15 Failure Time Adjustment for Evolving Programs
Part IV. Future Development
16 State of the Art
Appendixes
A. Review of Probability, Stochastic Processes and Statistics
B. Review of Hardware Reliability
C. Review of Software and Software Development for Hardware Engineers
D. Optimization Algorithm
E. Summary of Formulas for Application
F. Glossary of Terms
G. Glossary of Notation
H. Problem Solutions
I. Recommended Specifications for Supporting Computer Programs
References

This book is an extensive treatment of the application of reliability models to software.

Myers77

Myers, G. J. "An Extension to the Cyclomatic Measure of Program Complexity." *ACM SIGPLAN Notices 12*, 10 (Oct. 1977), 61-64.

Abstract: *A recent paper has described a graph-theoretic measure of program complexity, where a program's complexity is assumed to be only a factor of the program's decision structure. However several anomalies have been found where a higher complexity measure would be calculated for a program of lesser complexity than for a more complex program. This paper discusses these anomalies, describes a simple extension to the measure to eliminate them, and applies the measure to several programs in the literature.*

Myers reviews McCabe's *v*(*G*) metric, and proposes an extended metric, replacing the single value of *v*(*G*) by an ordered pair consisting of a lower bound (equal to the number of decision statements plus one) and an upper bound value (equal to the number of individual conditions plus one). Although the new metric seems to deal adequately with the examples cited, the new measure lacks the appeal of a single value for the metric.

Perlis81

Perlis A., F. Sayward, and M. Shaw, eds. *Software Metrics: An Analysis and Evaluation.* Cambridge, Mass.: MIT Press, 1981.

Table of Contents

Preface
The Role of Metrics in Software and Software Development
Summary of Panel Findings
Software Metrics: A Research Initiative
1 Toward a Scientific Basis for Software Evaluation
2 Design of Software Experiments
3 Experimental Evaluation of Software Characteristics
4 Software Project Forecasting
5 Controlling Software Development Through the Life Cycle Model
6 Resource Models
7 High Level Language Metrics
8 Data Collection, Validation and Analysis
9 A Scientific Approach to Statistical Software
10 Performance Evaluation: A Software Metrics Success Story
11 Statistical Measures of Software Reliability
12 The Measurement of Software Quality and Complexity
13 Complexity of Large Systems
14 Software Maintenance Tools and Statistics
15 When is "Good" Enough? Evaluating and Selecting Software Metrics
Annotated Bibliography on Software Metrics

This book provides an extensive review of the status of software metrics as of 1981 or slightly before. Specifically, it contains a number of state-of-the-art evaluations, as well as recommendations for research initiatives in related areas of software metrics. For reference purposes, it also contains an extensive annotated bibliography of more than 350 related references, almost all of which were published in the last ten years.

Potier82

Potier, D., J. L. Albin, R. Ferreol, and A. Bilodeau. "Experiments with Computer Software Complexity and Reliability." *Proc. 6th Intl. Conf. on Software Engineering.* New York: IEEE, Sept. 1982, 94-103.

Abstract: *Experiments with quantitative assessment and prediction of software reliability are presented. The experiments are based on the analysis of the error and the complexity characteristics of a large set of programs. The first part of the study concerns the data collection process and the analysis of the error data and complexity measures. The relationships between the complexity profile and the error data of the procedures of the programs are then investigated with the help of discriminant statistical analysis technique. The results of these analyses show that an estimation can be derived from the analysis of its complexity profile.*

The software used in this study consisted of a family of compilers, all written in the LTR language. The compiler consisted of a kernel, implemented by seven compilation units, and a code generator, implemented by four compilation units.

These programs were developed by a number of programmers over an extended period of time, from 1972 through 1977 and beyond. Both textual (Halstead) complexity metrics and structural (McCabe, reachability, paths, etc.) complexity metrics were investigated. An error data file containing data on over one thousand errors was created, although long after the work was done, in some cases. Observations: All complexity measures, except the normalized cyclomatic and cocyclomatic numbers, discriminated between procedures with no errors and programs with errors. Thus, although the cyclomatic number discriminates in the same manner, the authors conclude that this is only because of its high correlation with the size and volume metrics. In addition, the discriminating effect appeared to be maximal with regard to errors created during the design specification or coding stages. In ranking the measures as to discriminating effects, the vocabulary, *n*, appears at the top level of the decision tree.

Putnam78

Putnam, L. H. "A General Empirical Solution to the Macro Software Sizing and Estimating Problem." *IEEE Trans. Software Eng. SE-4*, 4 (July 1978), 345-361.

Abstract: *Application software development has been an area of organizational effort that has not been amenable to the normal managerial and cost controls. Instances of actual costs of several times the initial budgeted cost, and a time to initial operational capability sometimes twice as long as planned are more often the case than not. A macromethodology to support management needs has now been developed that will produce accurate estimates of manpower, costs, and times to reach critical milestones of software projects. There are four parameters in the basic system and these are in terms managers are comfortable working with—effort, development time, elapsed time, and a state-of-technology parameter. The system provides managers sufficient information to assess the financial risk and investment value of a new software development project before it is undertaken and provides techniques to update estimates from the actual data stream once the project is underway. Using the technique developed in the paper, adequate analysis for decisions can be made in an hour or two using only a few quick reference tables and a scientific pocket calculator.*

The author studied data on large systems developed by the U. S. Army Computer Systems Command, which develops application software in the logistic, personnel, financial, force accounting, and facilities engineering areas. Systems studied ranged in size from 30 man-years of development and maintenance effort to over 1,000 man-years. The Norden/Rayleigh model was used to derive an estimating equation relating the size (LOC) of the project to the product of a state of technology factor, the cube root of the applied effort, and the development time to the 4/3 power. This was found to work fairly well in the environment for which the data were available. However, the author states that the estimators developed here probably cannot be used by other software houses, "at least not without great care and considerable danger," because of the difference in standards and procedures. The author later developed this basic model into the proprietary product SLIM (Software LIfe-cycle Methodology) [Conte86].

Putnam80

Putnam, L. H. *Tutorial, Software Cost Estimating and Life-Cycle Control: Getting the Software Numbers.* New York: IEEE, 1980.

Table of Contents

The Work Breakdown Structure in Software Project Management (Tausworthe)
Useful Tools for Project Management (Norden)
Estimating Resources for Large Programming Systems (Aron)
A Method of Programming Measurement and Estimation (Walston & Felix)
Management of Software Development (Daly)
An Analysis of the Resources Used in the Safeguard System Software Development (Stephenson)
The Cost of Developing Large-Scale Software (Wolverton)
The Mythical Man-Month (Brooks)
Estimating Software Costs, Parts I, II & III (Putnam & Fitzsimmons)
SLIM (Putnam)
Bibliography

This is an excellent tutorial on software costing and estimating techniques, as they had developed to the late 1970's. It contains contributions from most of the major figures in this area up to this time, as can be seen from the table of contents, including Putnam, Walston and Felix, and Wolverton. Although this material is now quite dated, it is a good source of information on the work done prior to 1980.

Rodriguez86

Rodriguez, V. and W.-T. Tsai. "Software Metrics Interpretation Through Experimentation." *Proc. COMPSAC 86.* Washington, D. C.: IEEE Computer Society Press, Oct. 1986, 368-374.

Abstract: *This paper poses several conjectures derived from the current view of Software Metrics and then analyzes these conjectures through the use of source code metrics applied to two medium size software systems. The analysis attempts to determine the robustness of several metrics, the information conveyed through them and how that information could be used for software management purposes. One important use observed is the discriminant power of the metrics when software components are grouped together into sets of common characteristics to statistically distinguish between the groups.*

The authors attempt to determine the robustness of metrics such as McCabe's $v(G)$, LOC, etc., for a total of 15 code metrics plus Kafura and Henry's information flow metrics. Results are not easily summarized.

Rombach87

Rombach, H. D. "A Controlled Experiment on the Impact of Software Structure on Maintainability." *IEEE Trans. Software Eng. SE-13,* 3 (March 1987), 344-354.

Abstract: *This paper describes a study on the impact of software structure on maintainability aspects such as comprehensibility, locality, modifiability, and reusability in a distributed system environment. The study was part of a project at the University of Kaiserslautern, West Germany, to design and implement LADY, a* LA*nguage for* D*istributed s*Y*stems. The study addressed the impact of software structure from two perspectives. The language designer's perspective was to evaluate the general impact of the set of structural concepts chosen for LADY on the maintainability of software systems implemented in LADY. The language user's perspective was to derive structural criteria (metrics), measurable from LADY systems, that allow the explanation or prediction of the software maintenance behavior. A controlled maintenance experiment was conducted involving twelve medium-size distributed software systems; six of these systems were implemented in LADY, the other six systems in an extended version of sequential Pascal. The benefits of the structural LADY concepts were judged based on a comparison of the average maintenance behavior of the LADY systems and the Pascal systems; the maintenance metrics were derived by analyzing the interdependence between structure and maintenance behavior of each individual LADY system.*

The author reports results of a controlled experiment investigating the effect of software structure on the maintainability of software in a distributed system. The experiments were run on 12 systems of 1.5 to 15 KLOC. The software systems were developed in C-TIP (an extended Pascal) and LADY (a LAnguage for Distributed sYstems). Results indicated that complexity measures based on information flows are useful predictors of maintainability, including comprehensibility, locality and modifiability. Results with regard to reusability were inconclusive.

Rubin83

Rubin, H. A. "Macro-Estimation of Software Development Parameters: The ESTIMACS System." *Proc. SOFTFAIR: A Conference on Software Development Tools, Techniques, and Alternatives.* New York: IEEE, July 1983, 109-118.

Abstract: *System developers are continually faced with the problem of being asked to provide reliable estimates early in the software development process, often before any of the requirements are known. The ESTIMACS models offer a solution to his problem by relating gross business specifications to the estimate dimensions of effort hours, staff, cost, hardware, risk and portfolio effects. In addition, their implementation structure takes the user through a programmed learning experience in understanding the estimates produced.*

The author describes his recently developed ES-

TIMACS system for use in estimating, planning, and controlling the software development life cycle. The model includes estimators for development effort, staffing requirements, costs, hardware requirements, risk assessment, and total resource demands.

Rubin87

Rubin, H. A. "A Comparison of Software Cost Estimation Tools." *System Development 7*, 5 (May 1987), 1-3.

Abstract: *There are only a handful of software cost estimation tools that are in general use today. For the 8th International Conference on Software Engineering held in August 1985, the authors, or representatives, of the most "popular" tools were presented with a common problem to analyze as basis for comparison. In this context, each was asked to address his analysis approach, input parameters used, parameters not used, and results generated. This article contains the statement of the problem, a summary of the results provided by each participant, and a discussion of the implication of the results for those embarking on estimation programs within their own IS organizations.*

The author reports on a comparison of models JS-2, SLIM, GECOMO, ESTIMACS (by author), PCOC, and SPQR/10, all applied to the same cost estimation problem. Details of the results are not given, but Rubin states that the results "varied in a range of almost 8 to 1."

Ruston79

Ruston, H. (Workshop Chair). *Workshop on Quantitative Software Models for Reliability, Complexity and Cost: An Assessment of the State of the Art.* New York: IEEE, 1979.

This proceedings of a workshop on models of the software process involving reliability, complexity, and cost factors, is a good collection of work done up to this time (late 1970s). Although now somewhat out-of-date, it still serves as a good reference for work done in this area prior to 1980.

Shen83

Shen, V. Y., S. D. Conte, and H. E. Dunsmore. "Software Science Revisited: A Critical Analysis of the Theory and Its Empirical Support." *IEEE Trans. Software Eng. SE-9*, 2 (March 1983), 155-165.

Abstract: *The theory of software science was developed by the late M. H. Halstead of Purdue University during the early 1970's. It was first presented in unified form in the monograph* Elements of Software Science *published by Elsevier North-Holland in 1977. Since it claimed to apply scientific methods to the very complex and important problem of software production, and since experimental evidence supplied by Halstead and others seemed to support the theory, it drew widespread attention from the computer science community.*

Some researchers have raised serious questions about the underlying theory of software science. At the same time, experimental evidence supporting some of the metrics continue to be presented. This paper is a critique of the theory as presented by Halstead and a review of experimental results concerning software science metrics published since 1977.

This paper is a critical review of Halstead's software science and its empirical support. Among other things, shortcomings in the derivations of *N*, *V** and *T* are noted.

Shen85

Shen, V. Y., T. J. Yu, S. M. Thebaut, and L. R. Paulsen. "Identifying Error-Prone Software—An Empirical Study." *IEEE Trans. Software Eng. SE-11*, 4 (April 1985), 317-324.

Abstract: *A major portion of the effort expended in developing commercial software today is associated with program testing. Schedule and/or resource constraints frequently require that testing be conducted so as to uncover the greatest number of errors possible in the time allowed. In this paper we describe a study undertaken to assess the potential usefulness of various product- and process-related measures in identifying error-prone software. Our goal was to establish an empirical basis for the efficient utilization of limited testing resources using objective, measurable criteria. Through a detailed analysis of three software products and their error discovery histories, we have found simple metrics related to the amount of data and the structural complexity of programs to be of value for this purpose.*

This study involved five products developed and released since 1980, in three different languages (assembler, Pascal and PL/S). The authors report that the best predictors of defect rates at the end of program design and program coding phases were Halstead's n_1 and n_2, and the total number of decisions, DE. At the end of the software testing period, the best defect indicating metrics were found to be Halstead's n_2 and the actual number of program trouble memoranda (PTMs). The authors also state that "Our study of error density shows that this measure is, in general, a poor size-normalized index of program quality. Its use in comparing the quality of either programs or programmers without regard to related factors such as complexity and size is ill-advised."

Shepperd88

Shepperd, M. "A Critique of Cyclomatic Complexity as a Software Metric." *Software Engineering J. 3*, 2 (March 1988), 30-36.

Abstract: *McCabe's cyclomatic complexity metric is widely cited as a useful predictor of various software attributes such as reliability and development effort. This critique demonstrates that it is based upon poor theoretical foundations and an inadequate model of software development. The argument that the metric provides the developer with a useful engineering approximation is not borne out by the empirical evidence. Furthermore, it would appear that for a large class of software it is no more than a proxy for, and in many cases is outperformed by, lines of code.*

The author's criticisms, very briefly summarized, include the following. Theoretical: 1) simplistic approach to decision counting, 2) independence of generally accepted program structuring techniques, and 3) arbitrary impact of program modularization. Empirical: Studies to date do not establish validity of $v(G)$ as a reliable measure of any observed software properties. Responses can be summarized as follows. Theoretical: $v(G)$ is a simple, objective measure of one aspect of a software product. It is probably unrealistic to expect it to correlate well with any simply observable, gross characteristic of software, since such characteristics are determined by a large number of factors, many of which are unknowable, unmeasurable, or uncontrollable. Empirical: For similar reasons, empirical studies have failed to yield definitive results regarding the validity of $v(G)$ as a software metric, as pointed out by the author. A fundamental problem, noted in the article, is the lack of any explicit underlying model, without which attempts at empirical validation are meaningless.

Stetter84

Stetter, F. "A Measure of Program Complexity." *Computer Languages 9*, 3-4 (1984), 203-208.

Abstract: *The author proposes a measure of program complexity which takes into account both the relationship between statements and the relationships between statements and data objects (constants and variables). This measure, called program flow complexity, can be calculated from the source text of a program in an easy way.*

This paper is a review of McCabe's complexity measure and Myers's extension to it. The author proposes a "cyclomatic flow complexity" that is claimed to eliminate the shortcomings of the former metrics.

Symons88

Symons, Charles R. "Function Point Analysis: Difficulties and Improvements." *IEEE Trans. Software Eng. 14*, 1 (Jan. 1988), 2-11.

Abstract: *The method of Function Point Analysis was developed by Allan Albrecht to help measure the size of a computerized business information system. Such sizes are needed as a component of the measurement of productivity in system development and maintenance activities, and as a component of estimating the effort needed for such activities. Close examination of the method shows certain weaknesses, and the author proposes a partial alternative. The paper describes the principles of this "Mark II" approach, the results of some measurements of actual systems to calibrate the Mark II approach, and conclusions on the validity and applicability of function point analysis generally.*

Symons presents a critical review of Albrecht's FP methodology, pointing out several perceived shortcomings. He concludes that the method was developed in a particular environment and is unlikely to be valid for more general applications. He then proceeds to develop an alternative formulation, for what he calls "Mark II" Function Points. Whereas Albrecht's FP formula involves inputs, outputs, internal files, external files, and external inquiries, the author's new formula involves only inputs, outputs, and entities. In addition, the author introduces six new factors into the computation of the TCF (Technical Complexity Factor), thus raising the total number of such factors from 14 to 20. Although the author may have provided additional rationale for the new formulation, the net result seems to be a relatively minor modification of the original FP formulas. Furthermore, the new Mark II FP formulas suffer from the same type of counting difficulties and lack of universality for which the original formulas were criticized.

Tausworthe81

Tausworthe, R. C. *Deep Space Network Software Cost Estimation Model.* TR #81-7, Jet Propulsion Lab, Pasadena, Calif., 1981.

Abstract: *This report presents a parametric software cost estimation model prepared for JPL Deep Space Network (DSN) Data Systems implementation tasks. The resource estimation model modifies and combines a number of existing models, such as those of the General Research Corp., Doty Associates, IBM (Walston-Felix), Rome Air Development Center, University of Maryland, and Rayleigh-Norden-Putnam. The model calibrates the task magnitude and difficulty, development environment, and software technology effects through prompted responses to a set of approximately 50 questions. Parameters in the model are adjusted to fit JPL*

software life-cycle statistics. The estimation model output scales a standard DSN Work Breakdown Structure, which is then input to a PERT/CPM system, producting a detailed schedule and resource budget for the project being planned.

The above abstract is quoted from DACS Document #MBIB-1, "The DACS Measurement Annotated Bibliography, A Bibliography of Software Measurement Literature," May 1986.

Thebaut84

Thebaut, S. M. and V. Y. Shen. "An Analytic Resource Model For Large-Scale Software Development." *Information Processing and Management 20*, 1-2 (1984), 293-315.

Abstract: *Recent work conducted by members of the Purdue software metrics research group has focused on the complexity associated with coordinating the activities of persons involved in large-scale programming efforts. A resource model is presented which is designed to reflect the impact of this complexity on the economics of software development. The model is based on a formulation in which development effort is functionally related to measures of product size and manloading. The particular formulation used is meant to suggest a logical decomposition of development effort into components related to the independent programming activity of individuals and to the overhead associated with the required information flow within a programming team. The model is evaluated in light of acquired data reflecting a large number of commercially developed software products from two separate sources. Additional sources of data are actively being sought. Although strongly analytic in nature, the model's performance is, for the available data, at least as good in accounting for the observed variability in development effort as some highly publicized empirically based models for comparable complexity. It is argued, however, that the model's principal strength lies not in its data fitting ability, but rather in its straight forward and intuitively appealing representation of relationships involving manpower, time, and effort.*

The cooperative programming model (COPMO) is proposed in this article. In this model, the equation for total effort includes two terms, one corresponding to the effort expended in programming-related activities by individuals and the other corresponding to the effort expended in coordinating these activities among all programming team members. As noted above, attempts to validate the model against empirical data indicate that the model compares favorably with other models of comparable complexity, while possessing a more satisfying intuitive basis. NASA and Boehm's data sets were used to compare the model with the COCOMO and Putnam models.

Troy81

Troy, D. A. and S. H. Zweben. "Measuring the Quality of Structured Designs." *J. Syst. and Software 2*, 2 (June 1981), 113-120.

Abstract: *Investigates the possibility of providing some useful measures to aid in the evaluation of software designs. Such measurements should allow some degree of predictability in estimating the quality of a coded software product based upon its design and should allow identification and correction of deficient designs prior to the coding phase, thus providing lower software development costs. The study involves the identification of a set of hypothesized measures of design quality and the collection of these measures from a set of designs for a software system developed in industry. In addition, the number of modifications made to the coded software that resulted from these designs was collected. A data analysis was performed to identify relationships between the measures of design quality and the number of modifications made to the coded programs. The results indicated that module coupling was an important factor in determining the quality of the resulting product. The design metrics accounted for roughly 50-60 percent of the variability in the modification data, which supports the findings of previous studies. Finally, the weaknesses of the study are identified and proposed improvements are suggested.*

The authors attempt to correlate software design parameters, as taken from structure charts, with software quality, as measured by defect counts.

Walston77

Walston, C. E. and C. P. Felix. "A Method of Programming Measurement and Estimation." *IBM Systems J. 16*, 1 (1977), 54-73. Reprinted in [Putnam-80], 238-257.

Abstract: *Improvements in programming technology have paralleled improvements in computing system architecture and materials. Along with increasing knowledge of the system and program development processes, there has been some notable research into programming project measurement, estimation, and planning. Discussed is a method of programming project productivity estimation. Also presented are preliminary results of research into methods of measuring and estimating programming project duration, staff size and computer cost.*

This is a classic paper in the area of software project measurement and estimation; it is based on statistical analyses of historical software data. It discusses the software measurements program initiated in 1972 in the IBM Federal Systems Division as an attempt to assess the effects of structured programming on the software development process. At the time the paper was written, the software database

contained data on 60 completed projects that ranged from 4,000 to 467,000 LOC, and from 12 to 11,758 person-months of effort. The projects represented 28 high-level languages, and 66 computer systems, and were classified as small less-complex, medium less-complex, medium complex, and large complex systems. After obtaining a basic relationship between LOC and total effort, 68 variables were investigated for their effects on productivity. Of these, 29 were found to correlate with productivity changes; they were then used to compute a productivity index for a given project.

Wolverton74

Wolverton, R. W. "The Cost of Developing Large-Scale Software." *IEEE Trans. Computers C-23*, 6 (June 1974), 615-636. Reprinted in [Putnam80], 282-303.

Abstract: *The work of software cost forecasting falls into two parts. First we make what we call structural forecasts, and then we calculate the absolute dollar-volume forecasts. Structural forecasts describe the technology and function of a software project, but not its size. We allocate resources (costs) over the project's life cycle from the structural forecasts. Judgement, technical knowledge, and econometric research should combine in making the structural forecasts. A methodology based on a 25 x 7 structural forecast matrix that has been used by TRW with good results over the past few years is presented in this paper. With the structural forecast in hand, we go on to calculate the absolute dollar-volume forecasts. The general logic followed in "absolute" cost estimating can be based on either a mental process or an explicit algorithm. A cost estimating algorithm is presented and five traditional methods of software cost forecasting are described: top-down estimating, similarities and differences estimating, ratio estimating, standards estimating, bottom-up estimating. All forecasting methods suffer from the need for a valid cost data base for many estimating situations. Software information elements that experience has shown to be useful in establishing such a data base are given in the body of the paper. Major pricing pitfalls are identified. Two case studies are presented that illustrate the software cost forecasting methodology and historical results. Topics for further work and study are suggested.*

This is a classic paper, for Wolverton's model is one of the best-known cost estimation models developed in the early 1970s. The method is based upon using historical data from previous projects. Estimating the cost for a software module consists of three steps: first, estimating the type of software module; second, estimating the difficulty (complexity) based upon a six-point scale; and third, estimating the size (LOC) of the module. Once these three factors have been estimated, the cost of the module can be computed from historical cost data for similar projects. The cost of the software system is then simply the sum of the costs for all modules. Like most such models, it may work well in the environment for which it was developed but cannot be used in other environments without caution and, probably, recalibration to that environment.

Woodfield81

Woodfield, S. N., V. Y. Shen, and H. E. Dunsmore. "A Study of Several Metrics for Programming Effort." *J. Syst. and Software 2*, 2 (June 1981), 97-103.

Abstract: *As the cost of programming becomes a major component of the cost of computer systems, it becomes imperative that program development and maintenance be better managed. One measurement a manager could use is programming complexity. Such a measure can be very useful if the manager is confident that the higher the complexity measure is for a programming project, the more effort it takes to complete the project and perhaps to maintain it. Until recently most measures of complexity were based only on intuition and experience. In the past 3 years two objective metrics have been introduced, McCabe's cyclomatic number* v(G) *and Halstead's effort measure* E. *This paper reports an empirical study designed to compare these two metrics with a classic size measure, lines of code. A fourth metric based on a model of programming is introduced and shown to be better than the previously known metrics for some experimental data.*

Four software metrics—LOC, McCabe's $v(G)$, Halstead's E, and an author-modified E metric—are compared to observed program development times. The authors introduce the "Logical Module Hypothesis" as support for a modification of the E metric.

Woodward79

Woodward, M. R., M. A. Hennell, and D. Hedley. "A Measure of Control Flow Complexity in Program Text." *IEEE Trans. Software Eng. SE-5*, 1 (Jan. 1979), 45-50.

Abstract: *This paper discusses the need for measures of complexity and unstructuredness of programs. A simple language independent concept is put forward as a measure of control flow complexity in program text and is then developed for use as a measure of unstructuredness. The proposed metric is compared with other metrics, the most notable of which is the cyclomatic complexity measure of McCabe. Some experience with automatic tools for obtaining these metrics is reported.*

The concept of a "knot" as a measure of program complexity is introduced and compared with McCabe's $v(G)$.

Yau80

Yau, S. S. and J. S. Collofello. "Some Stability Measures for Software Maintenance." *IEEE Trans. Software Eng. SE-6*, 6 (Nov. 1980), 545-552.

Abstract: *Software maintenance is the dominant factor contributing to the high cost of software. In this paper, the software maintenance process and the important software quality attributes that affect the maintenance effort are discussed. One of the most important quality attributes of software maintainability is the stability of a program, which indicates the resistance to the potential ripple effect that the program would have when it is modified. Measures for estimating the stability of a program and the modules of which the program is composed are presented, and an algorithm for computing these stability measures is given. An algorithm for normalizing these measures is also given. Applications of these measures during the maintenance phase are discussed along with an example. An indirect validation of these stability measures is also given. Future research efforts involving applications of these measures during the design phase, program restructuring based on these measures, and the development of an overall maintainability measure are also discussed.*

Yau85

Yau, S. S. and J. S. Collofello. "Design Stability Measures For Software Maintenance." *IEEE Trans. Software Eng. SE-11*, 9 (Sept. 1985), 849-856.

Abstract: *The high cost of software during its life cycle can be attributed largely to software maintenance activities, and a major portion of these activities is to deal with the modifications of the software. In this paper, design stability measures which indicate the potential ripple effect characteristics due to modifications of the program at design level are presented. These measures can be generated at any point in the design phase of the software life cycle which enables early maintainability feedback to the software developers. The validation of these measures and future research efforts involving the development of a user-oriented maintainability measure, which incorporates the design stability measures as well as other design measures, are discussed.*

The approach taken is based upon the data abstraction and information hiding principles discussed by D. L. Parnas. Thus, the metrics defined assume a modular program structure and should be applicable to software designs employing modern programming practices. A design stability measure (DS) is computed for each module, and these values are then used to compute a program design stability measure (PDS) for the whole program. The design stability measures are based upon the assumptions buried in the module designs, and the potential ripple effects upon other modules if a module is modified.