DOT/FAA/CT-91/1

FAA Technical Center
Atlantic City International Airport
N.J. 08405

Software Quality Metrics

August 1991

Final Report

This document is available to the U.S. public through the National Technical Information Service, Springfield, Virginia 22161

U.S. Department of Transportation
Federal Aviation Administration

ACKNOWLEDGEMENT

The Federal Aviation Administration Technical Center and Computer Resource Management, Incorporated, wish to acknowledge Honeywell, Incorporated, Sperry Commercial Flight Systems Group of Glendale, Arizona, for their assistance in writing this document. Honeywell provided the FAA with flight control software code they developed. The code was used in evaluating and comparing different complexity metrics methodologies. In addition, Honeywell provided continuing technical support for their software during the research. All of this assistance was given at no cost to the Government.

TABLE OF CONTENTS

TABLE OF CONTENTS
(Continued)

LIST OF ILLUSTRATIONS

LIST OF TABLES

1. INTRODUCTION

1.1 Software Quality Metrics

Technological advances have led to smaller, lighter, and more reliable computers that are used increasingly in flight applications. Federal Aviation Administration (FAA) Certification Engineers (CEs) are faced with certification of aircraft that depend on this digital technology and its associated software.

When digital technology is to be used to perform some function aboard aircraft, the designer documents the technology and the applicant presents a package to the CE. Typically, the package might include design and test specifications, test plans, and test results for the system. This package assures the CE that the designer has properly developed and validated the system. Software Quality Metrics (SQM) may be used during software development and testing.

SQM technology attempts to quantify various quality-oriented factors, such as reliability and maintainability. The software developer determines the quality factors that are important to the application. Software metrics that correlate to these factors are used on the code to determine to what extent these factors have been reached. Based on the results, the developer determines whether the software meets the requirements set for it, and how well the software will perform.

If SQM results are submitted to support a system to be certified, the CE should understand SQM and their results and implications. This technical report documents the results of a study conducted to analyze SQM as they apply to the code contained in avionic equipment and systems. CEs can read this report for an in-depth understanding of how SQM are applied and interpreted.

1.2 Scope of the Technical Report

The technical report only includes metrics that are based on code and that apply to the types of code used in avionic equipment. Metrics that are well substantiated by empirical evidence receive an in-depth discussion. Metrics that are not well substantiated, but might be applied in the future, are also discussed.

Several metrics were applied to sample flight-control code. Applying metrics provided greater insight into the nature of evaluating software quality. It was essential to learn how to identify correctly what was being measured and to understand how to interpret the results. The test case results are used to tie in theory with practice.

History metrics and metrics that cover all phases of software development are not within the scope of this report. Both categories, however, do warrant further investigation. In cases where an SQM falls in either of these categories, the analysis is limited to that portion of the SQM that addresses the code phase.

Because SQM rely on statistics to indicate which software metrics correlate with a quality factor, this report does not cover all possible combinations leading to good correlations. The most accepted SQM are discussed. Acceptance is based on what SQM experts have concluded from their correlational studies. Future combinations are addressed since the software metrics and quality factors that are likely to compose new SQM are discussed independently.

1.3 How the Technical Report is Organized

Chapter 2 discusses the background of SQM, defines them, and defines the model for analyzing them.

Chapter 3 describes those software metrics which directly measure code attributes. Each metric is discussed in terms of why it is well-behaved and repeatable. The chapter covers the initial conditions, qualifying assumptions, context, and experimental results.

Chapter 4 describes quality factors which are important in achieving quality software. Where definitions differ for a factor, they are discussed in light of the context in which the factor is used.

Chapter 5 discusses SQM, the correlations of software metrics with quality factors. This chapter analyzes the three most substantiated SQM families in detail: Halstead's, McCabe's, and TRW's. Detailed, step-by-step instructions on applying and interpreting metrics are presented in chapter 17 of the Digital Systems Validation Handbook - Volume II (Elwell and VanSuetendael 1989).

Chapter 6 presents the conclusions.

Appendices A and B contain data sheets that summarize the basic data a CE needs to understand, analyze, and evaluate each software metric and quality factor.

1.4 How to Use this Technical Report

Read chapters 1 and 2 to understand the intent of the study and the report, and to understand the definitions and the SQM model, respectively. For those readers who have an interest in understanding the theory behind existing SQM, chapters 3 and 4 present the details of software metrics and quality factors, respectively. These two chapters can be read independently, but should be read before chapters 5 and 6. While chapter 5 specifically addresses three particular pairings of software metrics to quality factors, future SQM may correlate the software metrics of chapter 3 to new quality factors. Conversely, the quality factors of chapter 4 may be correlated with new software metrics. The report is organized so that new SQM can still be analyzed.

2. INTRODUCTION TO METRICS

2.1 Background of SQM

Computers play a primary role in industry and government. Because the hardware of modern systems relies heavily on the supporting software, software can critically affect lives. Many applications (e.g., flight critical systems, mission-critical systems, and nuclear power plants) require correct and reliable software. The increased importance of software also places more requirements on it. Thus, it is necessary to have precise, predictable, and repeatable control over the software development process and product. Metrics evolved out of this need to measure software quality.

In the early 70s, Halstead maintained that an attribute of software, such as complexity, is directly related to physical entities (operators and operands) of the code, and thus could be measured (Halstead 1972 and 1977). Halstead's metrics are fairly easy to apply. They are based on mathematical formulas and are backed by over a decade of refinement and follow-on empirical studies.

At approximately the same time, researchers at TRW became interested in measuring software qualities. The TRW team was responding to earlier research on the subject by Rubey and Hartwick (Boehm et al. 1978). This approach considered quality factors to be based on elementary code features that are assets to software quality. This work was adopted by the Rome Air Development Center (RADC) of the Air Force Systems Command. In 1977, RADC determined that software metrics were viable tools. Software acquisition managers could use them to determine accurately whether requirements were satisfied for new, delivered systems (McCall, Richards, and Walters 1977).

In the mid-70s, McCabe devised the Cyclomatic Complexity Metric, a measure based on the number of decisions in a program. McCabe's metric has been used extensively in private industry (Ward 1989).

SQM based on these three software metric families are widely accepted. Recent research focuses on refining some of the initial assumptions and on creating new SQM.

In developing new metrics, theory often precedes empirical evidence. Today, numerous companies are marketing SQM. Some do not back their measurement claims with much independent research or empirical substantiation. Other companies, however, have empirically validated their metrics. Many companies use their metric systems to address all phases of the software life cycle.

Metric researchers are split broadly into two camps: those who claim software can be measured and those who state that the nature of software does not lend itself to metrical analysis. The latter group includes researchers who have been disillusioned with the field, or who have mathematically "proved" that software cannot be measured. To further split the former group, some are busy refining existing metrics, others disproving parts of them. In any case, the majority of researchers are concerned about software quality and the need to quantify it.

Metric research is aimed at reducing software cost, keeping development on schedule, and producing software that functions as intended. Additionally, metrics are being developed to evaluate whether software accomplishes its functions reliably and safely. In the present context, it is hoped that metrics can be used to aid in the certification of avionic equipment and systems.

2.2 Definition of SQM

Before SQM can be understood, the terms need to be defined. A computer-based system consists of a machine and its associated programs and documentation. Hardware consists of the physical components of a computer. Software consists of computer programs and the documentation associated with the programs.

Code is a subset of software, and exists for the sole purpose of being loaded into the machine to control it. Code that can be read by people is source code. The translation of the source code that is loaded into the machine is called object code. This report addresses metrics that measure aspects of source code. In this report, any reference to code is to be understood as a reference to source code.

As used in the term, "SQM", quality modifies software. SQM measure whether software is quality software. This measure is based on quality factors. A quality factor is any software attribute that contributes either directly or indirectly, positively or negatively, toward the objectives for the system where the software resides. A particular quality is defined, or singled out, when it is determined that it significantly affects those objectives. Two examples are software reliability and complexity.

Metrics is another word for measures. The most basic measure is a repeatable, monotonic relationship of numbers to a magnitude of a property to be quantified. As soon as such a relationship is defined, the one set is a measure of the other. Repeatable means anyone will get the same results at any time. Since the relationship is a monotonic function, a certain change in the measure always represents a certain change in the property being measured, where either change is simply an increase or decrease in magnitude. (Increasingly negative numbers represent a decreasing magnitude.) Although such a function is consistently nonincreasing or nondecreasing, it is not proportional. In addition, it is desirable that the montonic measure have a one-to-one correspondence to the magnitude of the measured property so that it is not possible for differing property magnitudes to give the same measure.

A more desirable measure is one where the magnitude of the property is reflected in the measure by a smooth, mathematical relationship. This measure is considered well-behaved. The most desirable and common well-behaved measure is a linear one, since any change in the magnitude of a property is proportionally represented by the measure.

The magnitude of properties of objects can usually be measured, since a property magnitude can almost always be expressed by a repeatable, monotonic relationship. Further, these measures are usually well-behaved, and are often linear. With respect to software, the objects are the printed symbols that represent ideas. Well-behaved measures exist for these objects. See Curtis (1980) for more details on measurement theory.

If a repeatable, monotonic relationship could be established that relates measures of objects (a set of numbers) to subjective qualities, the result would be, by definition, a quality measure. SQM is the name given to a measure that relates measures of the software objects (the symbols) to the quality factors.

All practical SQM establish rules for relating the software symbols to the quality factors. These rules are usually expressed as well-behaved mathematical relationships. However, few of the SQM produce a completely repeatable and monotonic measure of the software quality. Nearly all SQM in use exhibit variability and lack of monotonicity in their results.

Nevertheless, practical, reliable SQM do exist. Their validity is based on the statistical inference that can be applied to systems under development, drawn from the analysis of systems in operation. Experiments must be performed to analyze how well the values produced by the SQM correlate to the quality it claimed to measure. The strength of the relationship is expressed in terms of statistical correlation or regression. Thus, each SQM has a population or a sample space for which the measurement is valid, as well as a level of confidence that can be attributed to it.

When the relationship between two randomly varying quantities is measured in terms of correlation, the correlational coefficient, ñ, indicates how close the relationship, if any, comes to being linear. It does not indicate the nature of the coefficients of the linear relationship. A perfectly linear positive relationship is indicated by ñ=1. A perfectly linear negative relationship is indicated by ñ=-1. A value of ñ=0 indicates that the two quantities are not linearly correlated. Two quantities that are not correlated are not necessarily unrelated. They may follow some nonlinear relationship (Hays and Winkler 1970). Therefore, the correlational coefficient can establish the presence of a relationship, but not the absence.

A particular SQM may claim that a software quality is linearly related to a software metric. Correlational experiments may verify this with a value of ñ=0.9. However, the proposed linear SQM relationship is still not necessarily valid. In order to be useful as a relative measure, the slope of the linear relationship must be confirmed. In order to be useful as an absolute measure, the zero intercept of the linear relationship must also be determined. Thus, "if the correlation significantly differs from zero and if the estimates are generally close to their actual counterparts, then one could convincingly argue the metric is valid" (Dunsmore 1984).

When the relationship between two quantities is measured by regression analysis, the relationship is addressed more directly. A slope, an intercept, and a coefficient of determination, r^2, are calculated. This coefficient is a measure of the amount of variation in the dependent variable (quality factor) that may be explained by the variation in the independent variable (software metric). "A high coefficient of determination leaves less variability to be accounted for by other factors" (Basili and Hutchens 1983).

In practice, the word "metric" is used loosely. It must be discerned from the context whether the word is short for software metric or SQM. This is not always clear, because many practitioners assume that the software qualities are directly linked with the software objects. They use the word metric as though the measure of the software object were the same as measuring the software quality. In this technical report, the measure of the software objects will be referred to as software metrics, and the measure of the quality factors, as SQM.

2.3 The SQM Model

There are two categories of elements measured by an SQM. The software object category contains directly measurable items: lines of code, conditional statements in a program or module, and number of unique operators and operands. The software quality category contains qualities software should possess to some degree: reliability, maintainability, simplicity, and ease of use. The two categories contain components for creating a product. The product is an SQM. Either category contains only the raw elements, which by themselves, signify little. When measures of the elements in the first category are combined (or "mapped") with elements from the other, an SQM is formed.

Thus, a particular SQM consists of three components: software metrics, quality factors, and the mapping between the two. This relationship allows scientists, programmers, and engineers to measure the quality of the software being evaluated.

Practitioners emphasize the importance of agreeing on selecting qualities appropriate to the application. Selecting appropriate quality factors is subjective. This report discusses successful pairings or mappings to illustrate the strength of metrical analysis. This model is flexible because new SQM are produced simply by combining new or existing software metrics with new or existing quality factors. The decision on whether a software measure successfully "correlates" with a quality factor is subjective and depends on the needs of the application, the company sponsoring the research, and other concerns.

2.4 Limits of Measurements

SQM can be applied to any readable set of symbols. However, the results of a particular SQM will be meaningful only when the SQM is applied to the particular subset of symbols for which it has been prequalified. Thus, a metric can be thought of as a function which has a certain domain over which it is defined. Outside that domain, the results are unpredictable, and therefore the metric should not be applied.

Within the domain, the measure of a software quality is only as good as the proposed correlation of the software quality to the software object measured, as determined by experimentation. The domain is typically no larger than the sample space of the experiments, although, if a sufficient number of experiments are performed, the sample space can reliably project the properties of the population. Within the domain, when a highly-correlated SQM is applied, it typically does not measure absolute levels of the software quality.

An SQM is generally a ratio that is expressed as a unitless number. For example, a high correlation is found between software maintenance time and the number of decision statements in a program. If Program A has twice as many decision statements as Program B, the time required to maintain A probably will be double that of B (assuming previous experimental studies show a linear relationship between the two attributes). This type of result cannot specify a unit, such as particular number of hours of maintenance. It is a relative measure and only measures trends. It does not assess software quality, but improvements in quality. The improvement suggested by a doubling of an SQM may actually represent an inconsequential improvement in absolute terms. The measurement could simply indicate the software has improved from unacceptable to barely passable.

If, however, the relationship has been calibrated by making comparisons to software whose qualities are established by after-the-fact experience, the SQM does provide an absolute measure of software quality and will have units attached to it. Such a metric can be used to assess levels of software quality.

Software metrics, in and of themselves, predict nothing. They simply measure the state of the software at the time of measurement. For example, a particular code metric may indicate that at the time of measurement, a module contained a specific number of decisions.

On the other hand, when a software metric is correlated with a quality factor, the SQM inherently predict some future state of the software. The quality factors are only meaningful when the code is applied in a context. Code, as printed on paper, exhibits no software quality. But in the context of a programmer having to read the code in order to change it, the presence of the quality factor, maintainability, becomes evident. Thus, when an SQM measures a software quality, it predicts that when the code is viewed in that context, the code will exhibit that quality.

SQM indirectly address testing adequacy. Testing adequacy assumes that the level of software quality desired is produced by testing for failures and fixing them until the level is achieved. Clearly, SQM are helpful here. They provide a quantitative assessment of the quality level of the software produced. This assessment can be compared to the level specified. Testing is objectively adequate when the specification is fulfilled. Without SQM, determining testing adequacy is a qualitative decision.

Specifying test coverage requirements is one way of assuring testing adequacy. Radio Technical Commission for Aeronautics (RTCA) Standard DO-178A recommends certain types of testing for software in each of the criticality categories. If all the test cases required by each type are performed, testing adequacy is assured. While SQM do not address test coverage, some software metrics address test coverage by implying the number of test cases necessary. Adequate coverage would require that the count produced by these metrics be met. This would not be sufficient to establish test coverage, however, since the software metric does not address particular test cases. Other software tools are specifically designed to generate test cases.

2.5 Metrics and Standards

SQM can be used to satisfy the requirements of some standards that apply to avionics. Title 14 of the U.S. Code of Federal Regulations (CFR) contains federal laws that apply to aeronautics and space. Chapter 1, Subchapter C, Federal Aviation Regulation (FAR) Part 25, specifies the airworthiness standards for transport category airplanes. In particular, Section 25.1309 specifies that airplane systems and associated components must be designed in such a way that the probability of their failure is not excessive. Compliance with this design requirement must be demonstrated by analysis. SQM can be used in this analysis. This airworthiness standard currently contains no Special Conditions to which SQM technology would apply.

The requirement of Section 25.1309 is stated in qualitative terms. The FAA publishes formal guidelines for meeting the requirements of the CFR. One such guideline, Advisory Circular (AC) 25.1309-1A, gives specific design and analysis procedures and techniques which can be used to meet the requirements of Section 25.1309 of the CFR. Of particular interest for SQM is that this AC assigns quantitative thresholds to the probability of failure that is acceptable. This may make SQM even more useful for demonstrating compliance with Section 25.1309. The FAA has also adopted the Society of Automotive Engineers' (SAE) Aerospace Recommended Practice (ARP) 1834 as an informal guideline for the fault and failure analysis of digital systems and equipment, as introduced in AC 25.1309-1A. Some day SQM may be incorporated into these guidelines.

The certification process consists of determining whether avionic equipment and systems comply with Section 25.1309. To support this determination, the FAA has adopted the RTCA Standard, RTCA/DO-178A, "Software Considerations in Airborne Systems and Equipment Certification" (1985), as another informal guideline. This standard recommends several software design and analysis procedures that lend themselves to SQM application. Another informal guideline for certification is the Digital Systems Validation Handbook (DOT/FAA/CT-88/10), which now contains a study of SQM as used for certification purposes.

The software development standard, Defense System Software Development (DOD-STD-2167a), incorporates the use of SQM in the software development process. The requirements for reliability, maintainability, availability, and other quality factors are to be specified in the System/Segment Specification (SSS), as required by paragraphs 10.1.5.2.5ff of the associated Data Item Description, DID-CMAN-80008A. Furthermore, the software quality standard, Defense System Software Quality Program (DOD-STD-2168), recognizes that software development tools will be used in the software quality program. The tools are to be identified in the Software Quality Program Plan (SQPP) according to the specification of paragraph 10.5.6.2 of the associated Data Item Description, DID-QCIC-80572.

SQM have become so widely used that they are now being incorporated in national and international standards. Standardization of their application and interpretation will make them more useful to the software engineering community. The Institute of Electrical and Electronics Engineers (IEEE) has published standards on several commercial software metrics in IEEE Standard 982. This standard defines and explains the use of measures which can be used to produce reliable software. Furthermore, the IEEE Computer Society has produced a draft standard on an SQM methodology for the International Standards Organization (ISO). The ISO plans to release this standard in 1991.

3. SOFTWARE METRICS

3.1 General

Chapter 3 discusses software metrics. A representative sample is presented to give the reader information on metrics. These metrics apply to avionic code and have been used successfully in industry. The following paragraphs explain the criteria used to select the metrics.

1. The software metric must apply to code. Some metrics are included that might apply to code, although the application may not be apparent, at first glance.

2. The metric must apply to avionic equipment code. Programs written in Assembly, FORTRAN, Pascal, C, and Ada were considered to represent avionic code. Hence, any of these metric applications could be generalized to apply to avionic code. The majority of metrics are selected based on their universal application to code.

3. The metric must be related to a known quality factor. In some cases, metrics are not correlated with quality factors, but can still help programmers develop software. For example, a performance metric shows the number of seconds a program takes to execute. The measurement determines to what degree the performance deviates from an accepted value, but does not measure the quality of the program. Although these are software metrics, they are not SQM. Because these metrics may be used in subsequent SQM, they are also addressed.

4. The metric must be substantiated by independent experimental evidence. Many metrics have been derived but not tested rigorously. Metrics that have not been convincingly verified through experiment are addressed in sections 3.5 through 3.7.

Appendix A contains data sheets on each selected software metric. The CE may refer to these sheets to assess specific metrics.

Sections 3.3 and 3.4 discuss metrics that passed the previously discussed criteria. Sections 3.5 through 3.7 discuss some software metrics that have not passed criterion number 4. These metrics, based on a common theoretical foundation, are grouped as a family of metrics. This family is given the name of the person or organization that originated it. Each metric within the families is discussed and analyzed. The advantages and disadvantages of using a particular metric are also presented within the analysis. The last section within each family provides a general analysis on several or all metrics within the group.

3.2 Halstead's Software Metrics

Halstead coined the term "Software Science" for his approach to software metrics. He reasoned that algorithms have measurable characteristics because they obey physical laws. Software science, therefore, deals with properties of algorithms whether they are measured directly, indirectly, statically, or dynamically. The science also considers relationships that remain invariant under translation from one language to another (Halstead 1977).

Halstead's metrics are based on two code features: operators and operands. Operators are symbols which affect the value or ordering of operands. Operators may consist of contiguous symbols (such as the letters which form a word), or symbols which are physically separate but operate as a unit. For example, although left and right parentheses are separate, they form a single grouping operator. Operands are the variables or constants on which the operators act.

While these definitions supply necessary conditions for identifying operators or operands, they are not sufficient. Before applying Halstead's metrics, the operators and operands must be clearly and precisely defined so that the count of each is repeatable. These counts will be used in the formulas that determine a program's length, the volume of a program, its complexity, and other attributes.

The following counts compose the foundation of Halstead's metrics:

- The number of unique operators, η_1.

- The total number of operator occurrences, N_1.

- The number of unique operands, η_2.

- The total number of operand occurrences, N_2.

Halstead's family of equations have been developed for programs that implement algorithms. Algorithms implemented in object code are composed solely of operators and operands. Halstead assumes that a higher computer-oriented language can also be viewed as consisting solely of operators and operands, since the resulting object code contains these two fundamental units. Hence, he believes the metrics can be applied to source programs.

The following sections describe the quantities used in Halstead's metric system. All estimated quantities have a circumflex (^) over them to distinguish them from actual quantities. A critique of Halstead's theories is presented after the analyses.

3.1 Vocabulary

The Vocabulary of a program is as follows:

$$\eta = \eta_1 + \eta_2 \text{ words}$$

where η_1 is the number of unique operators and η_2 is the number of unique operands. For the purpose of discussing the sum of these quantities, the term "word" is used as the Vocabulary unit. A word is a sequence or string of characters set off by the delimiters defined for a particular context. In this sense, both operators and operands constitute words. By the equation, η is the total number of distinct operators and operands.

Consider the following FORTRAN statements:

```
A = B + C
D = A * A
```

The operators include the equal sign (=), the plus sign (+), and the multiplication sign (*). Additionally, the carriage return/line feed (CR/LF) is considered another operator because it separates the two statements. The number of unique operators, η_1, is four. "A", "B", "C", and "D" are unique operands, so η_2 is four. The total Vocabulary, η, for this example program is eight words.

The above example is easily calculated because the definitions for the operators and operands are straightforward. As discussed in section 3.2, the definitions of operators and operands need to be precise. Different languages have different syntactical rules which may require that additional operators be defined.

Even in the simple example presented, the CR/LF is considered to be an operator because it separates statements, hence it is performing an action on operands. In some languages this separator is explicit, like Pascal where the separator is a semicolon. Although it is a subjective decision whether to count the statement separator, once the decision is made, that convention should be followed throughout the counting procedure. If the separator is counted in Pascal, and a programmer wanted to compare attributes of that program with one written in FORTRAN, the CR should be counted in FORTRAN as well to make the counts consistent, hence more meaningful. Otherwise, anything learned about the metric in one context cannot be applied unequivocally to another.

While Halstead published guidelines to help define operators and operands, the guidelines do not cover every possibility. Halstead defines parentheses, for example, as a grouping operator. The pair, together, counts as one operator. Similarly, the beginning and end statements of a procedure count as one because the pair defines the area to be executed.

Consider the FORTRAN code shown in figure 3.2-1. According to Halstead, comments are not an essential part of the program, and are not counted. Declaration statements also are to be treated as comments. In the example, the two comments and the declaration statement, "INTEGER PRESS", are not included in the count. Although Halstead does not count such non-executable statements, an implementor may wish to consider counting them, since one could argue that they add to the volume of a program.

```
        SUBROUTINE ATMOS
C       PRINTS WHETHER CUBIC CENTIMETER VALUE
C       OF PRESSURE IS <, >, OR = TO 1 ATMOSPHERE
        INTEGER PRESS
        READ PRESS
        INPUT *, 'INPUT PRESSURE AS AN INTEGER:', PRESS
        IF (PRESS .LT. 760) THEN
                PRINT *, PRESS, 'IS LESS THAN 1 ATMOSPHERE'
                PRINT *, 'UNDER STANDARD CONDITIONS.'
        ELSE IF (PRESS .EQ. 760) THEN
                PRINT *, 'THIS IS 1 ATMOSPHERE'
                PRINT *, 'UNDER STANDARD CONDITIONS.'
        ELSE
                PRINT *, PRESS, 'IS GREATER THAN 1 ATMOSPHERE '
                PRINT *, 'UNDER STANDARD CONDITIONS.'
        END IF
        END
```

FIGURE 3.2-1. SAMPLE FORTRAN CODE

The "IF...THEN...ELSE IF...ELSE" is a control structure, where the executed branch depends on the various conditions. While Halstead emphasizes that all control structures are classified as operators, he does not delineate how they are to be counted. Questions on what parts of the "IF" branching structure ("IF", "THEN", "ELSE IF", "ELSE", "END IF") should be counted separately are left up to the implementor. If each part of this structure is counted as one operator, the operator count of this statement would be four (IF/THEN, ELSE IF/THEN, ELSE, and END IF). However, if the branching structure is viewed as one unit, the count would be one. In the latter case, the "IF" branching structure is a single, compound operator.

Halstead reasoned that the keyword, "CALL", is not a variable or constant, so it is not an operand. He classified the keyword combination, "CALL PRESS", as one operator because the two words together signify some action is taking place. From the above discussion, however, "CALL" could be considered an operator; and "PRESS" an operand because it specifies the location receiving the transfer of control.

When an implementor analyzes statements to establish a counting strategy, the operators and operands should generally alternate; i.e., the line should contain an operator, an operand, an operator, etc. The depth to which a line is analyzed is left up to the implementor. This depth, in turn, will determine which tokens should be operators and which tokens operands. In this example, the "PRINT" statements compose slightly less than 50 percent of the program if nonexecutable statements are not counted. Hence, the way in which a "PRINT" statement is interpreted can affect the counts greatly. Examine one of the print statements from the example:

```
PRINT *, PRESS, 'IS LESS THAN 1 ATMOSPHERE'
```

"PRINT" is an operator because it performs an action on the following character string. The asterisk instructs the computer to print each item in the "PRINT" list according to a default print format. The asterisk is the first uncertain token. Since it follows the operator "PRINT", it should be considered an operand. Since the next item is a comma (another operator) this analysis is reasonable. Following this logic, "PRESS" is the next operand, followed by a comma. To continue this alternating sequence, the apostrophes and the enclosed character string would be considered one operand. The number of unique operators, η_1, would be two, while the number of unique operands, η_2, would be three. The Vocabulary would be five words.

Another analysis of the above statement would have the "PRINT *," combination equal one operator. "PRESS" would be an operand, the comma an operator, and the apostrophes and character string together would equal one operand. In this count, η_1 would equal two. (The second comma is counted as a unique operator because the first comma was included in the "PRINT *," string.) The operand, η_2 would equal two. The Vocabulary would be four words.

If a single high-level language "PRINT" statement can be interpreted in more than one way, the effect on an entire program must be considered. The interpretation can significantly affect both operator and operand counts. Furthermore, if several programs are to be measured, a consistent counting strategy must be developed for the entire programming language and must be used consistently.

The level of scanning affects the operand and operator counts. The level is determined by the resolution, or token size. Each token must be determined to be an operator or operand. Consider the Assembly code shown in figure 3.2-2. In this example, the instructions can yield at least two possible ways of interpreting operators and operands, as shown in table 3.2-1.

```
          LD    A,0FH
          LD    A,DONUM
          LD    HL,LSTAT+78H
ZEROLP    LD    (HL),0
          INC   HL
          DEC   A
          JP    NZ,ZEROLP
          CALL  DOSCAN
```

FIGURE 3.2-2. SAMPLE ASSEMBLY CODE

In the higher level view, a token is set off by the following delimiters: space, CR/LF, or comma. For example, in the instruction "LD HL,LSTAT+78H", "LD" is the operator, "HL" is an operand, and "LSTAT+78H" is an operand. In the instruction "JP NZ,ZEROLP", "JP" is an operator, and "ZEROLP" and "NZ" are operands. Similarly, "CALL DOSCAN" is interpreted simply: "CALL" acts on the operand, "DOSCAN".

TABLE 3.2-1. TWO INTERPRETATIONS OF ASSEMBLY INSTRUCTIONS

Assembly Instruction	Higher Level		Lower Level	
	Operator	Operand	Operator	Operand
LD A,0FH	LD LD	A 0FH	A H	0F
LD (SPSO),A	LD	(SPSO) A	LD ()	SPSO A
LD A,DONUM	LD	A DONUM	LD	A DONUM
LD HL,LSTAT+78H	LD	HL LSTAT + 78	LD +	HL LSTAT 78H
ZEROLP LD (HL),0	LD	(HL) 0	LD ()	HL 0
INC HL	INC	HL	INC	HL
DEC A	DEC	A	DEC	A
JP NZ,ZEROLP	JP	NZ ZEROLP	JP	NZ ZEROLP
CALL DOSCAN	CALL	DOSCAN	CALL	DOSCAN

In a lower level view, the parentheses and the plus sign are recognized as operators and therefore are also counted as delimiters. Hence, the instruction "LD HL,LSTAT + 78H" is interpreted to contain the operators "LD" and "+". In this finer resolution, the operators and operands still alternate, as Halstead recommended; e.g., "LD" is an operator, "HL" an operand, the comma an operator, "LSTAT" an operand, "+" an operator, and "78H" an operand. At the lower level, the "(HL)" in one instruction is split into an operator, "()", and an operand, "HL".

Some thought must be given to interpreting control constructs, such as the jump and call instructions. For example, should "CALL DOSCAN" be considered one operator ("CALL") and one operand ("DOSCAN")? Or, should "CALL" be one operator, the space an operand, and "DOSCAN" another operator, since it directs control flow to code contained beneath its label?

The FORTRAN and Z80 Assembly code examples are presented to give the reader some areas to consider when counting rules are established. The examples point out the importance of agreeing on rules that will cover all the unique dilemmas that the constructs of a particular language may present to the practitioner.

The counting method should be established by a programmer who knows the language. Without this knowledge, it is more difficult to establish a counting strategy because a token must be declared an operator or operand based on the context in which it is found.

Difficulties in declaring operators or operands could occur if the same symbol is used for more than one function. In FORTRAN, a set of parentheses may delimit expressions, arguments, or subscripts. If the counting rules do not take into consideration this possibility, the number of unique operators will be undercounted. The same undercounting can occur with operands.

The Vocabulary of multiple modules taken together to get one Vocabulary measure is not the sum of the individual modules. Hence, Vocabulary is not additive.

3.2.2 Implementation Length

The Implementation Length of a program is as follows:

$$N = N_1 + N_2 \text{ words}$$

where N_1 is the total number of operators and N_2 is the total number of operands. This equation determines the total length of the program because all operators and operators, whether they are unique or not, are counted.

Consider the following FORTRAN code:

```
A = B + C
D = A * A
```

The total number of operators, N_1, is five (two equal signs, one plus sign, one multiplication sign, and one CR/LF); N_2 is six (three "A"s, one "B", one "C", and one "D"); and the Implementation Length, N, is 11 words.

The best measures are linear because they are easiest to interpret: a change in one quantity produces a proportional change in another. For practicality, most equations that are not linear are approximated using linear regression techniques.

3.2.3 Estimated Length

While the previous equation directly calculates the Implementation Length, the length of an implementation can be estimated. Halstead equated the number of possible combinations of operators and operands with the number of elements in the power set, or 2^N, of a set with N elements. Hence,

$$2^{N} = \eta_1^{\eta_1} \times \eta_2^{\eta_2}$$

The length equation is as follows:

$$N = \log_2\left(\eta_1^{\eta_1} \times \eta_2^{\eta_2}\right)$$

or

$$N = \log_2 \eta_1^{\eta_1} \times \log_2 \eta_2^{\eta_2}$$

yielding the following Estimated Length equation:

$$\hat{N} = \eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2 \quad \text{words}$$

where the Estimated Length produces an estimate of the total number of operators and operands.

Since the Estimated Length is an approximation of the Implementation Length, it has an error associated with it. The coefficient of correlation shows that a strong relationship exists between the Implementation Length and the Estimated Length. In fact, Elshoff (1976) used the Estimated Length equation to calculate the length of 154 PL/1 programs. He then compared the Estimated Length with the Implementation Length and found correlations higher than 0.97 between the two measures, giving strong support to the assumption that this equation can be used to estimate the actual length. This measure serves as a quick check for pure or well-structured programs. If the correlations are significantly lower than 0.97, the module estimated may contain impurities and should be analyzed and possibly recoded.

This estimate assumes that recurring patterns of operators and operands are placed in subroutines, rather than existing several places within the program. If code is repeated in the program, the Estimated Length will underestimate the Implementation Length.

The Estimated Length equation also assumes operators and operands alternate without variation. If the code does not follow this convention, the estimate will again underestimate the Implementation Length.

The Estimated Length equation can overestimate the actual length in the following situation. The Implementation Length indicates the total length of a program, and depends on the operator and operand counts. If the number of unique operators and operands increases, the length estimator () increases even though the Implementation Length does not. It will significantly overestimate the length if each unique operator or operand is used only once or twice in the body of the program (Shen, Conte, and Dunsmore 1983).

Because the Estimated Length equation is not linear, adding the Estimated Lengths of sub-modules would not be expected to give an accurate Estimated Length for the module they constitute. Nevertheless, an experiment by Ingojo indicated that the relationship holds fairly well (Halstead 1977).

The application of this metric to several programs has shown that is an acceptable estimator of N. Shen, Conte, and Dunsmore (1983) show, however, that the estimate does tend to overestimate the length for small programs and to underestimate the length for large ones.

This measure provides a reasonably monotonic, repeatable relationship, assuming a counting method is used consistently.

3.2.4 Volume

Halstead maintains that the size of an algorithm is an important aspect of software. He devised an equation based on the Vocabulary of the program and the Implementation Length. Implicit in this equation is the assertion, represented by the factor, $\log_2 \eta$, that people use a binary search to locate information.

When a given algorithm is translated from one language to another, its size changes. A measure of size should reflect this change. The Program Volume is as follows:

$$V = N\log_2 \eta \text{ bits}$$

where V is the Volume, N is the number of words in the program (or $N_1 + N_2$), and $\log_2 \eta$ is the minimum number of bits required to represent all unique words in the program.

Halstead also defines Volume in terms of effort:

$$V = EL$$

where E is the total number of mental discriminations required to generate a given program and L is the Program Level. In this definition, Volume is defined as the count of the number of mental discriminations required to generate a program. This definition assumes that programmers use a binary search to select the Vocabulary and that this search technique applies to programming activities. Since E is the total number of mental discriminations and L is unitless, Volume units can also be "discriminations".

The Volume equation uses combinatory logic as its basis. The general form for calculating the minimal number of digits required to represent each element of a set in a particular number base is as follows:

$$\log_b \#$$

where "b" represents the number base one selects, and "#" is the number of elements in the set.

Halstead chose base 2 because, in current computers, the words are stored in binary digits, or bits. $\text{Log}_2 \eta$ is the minimum number of bits that could uniquely represent each word in a Vocabulary of η words. This combinatory relationship is analogous to selecting the field size for printing numbers of a certain size. The specification I3 in FORTRAN, for example, simply allocates the number of digits required to print quantities with a maximum value. Thus, three digits are required to represent every number from 0 to 999 in base 10. If the number is greater than 999, an error would result. Similarly, the $\log_2 \eta$ gives the number of bits per word. For example, if $\eta = 8$, then $\log_2 8 = 3$ bits/word. Thus, the words can be represented by a minimum of three bits, as shown in table 3.2-2.

TABLE 3.2-2. REPRESENTATION OF ASSEMBLY WORDS

Words	Bits
CALL	000
JUMP	001
LD	010
CP	011
A	100
LOOP	101
PRINT	110
INPUT	111

Halstead then multiplied the number of bits per word, $\log_2 \eta$, by the total number of words, N, to give the total number of bits, V.

The Volume for a program can be directly calculated from the operator and operand counts, as follows:

$$V = (N_1 + N_2)\log_2(\eta_1 + \eta_2) \text{ bits}$$

This measure, like the Estimated Length, is monotonic and repeatable, assuming a counting method is used consistently.

3.2.5 Potential Volume

As previously noted, if a program is translated from one language to another, its volume will change. In particular, high-level languages tend to express an algorithm more efficiently than low-level ones. For example, the FORTRAN loop shown in figure 3.2-3, with a Volume of $8\log_2 7 \approx 22$, could be translated into the Assembly code also shown in figure 3.2-3, with a Volume of $20\log_2 13 \approx 74$ bits. Conversely, if an algorithm is implemented in a more powerful language, the Volume would be reduced. Halstead derived the Potential Volume equation to express the size of the most succinct implementation of an algorithm.

The Potential Volume of an algorithm is the shortest, ideal form it could assume. It is derived from the Volume equation:

$$V^* = (N_1^* + N_2^*)\log_2(\eta_1^* + \eta_2^*) \text{ bits}$$

where V^* is the Potential Volume, N_1^* is the minimum number of operator occurrences, N_2^* is the minimum number of operand occurrences, η_1^* is the minimum number of unique operators, and η_2^* is the minimum number of unique operands.

FORTRAN	ASSEMBLY
DO I = 1,10,2	LD A,1
.	LD B,10
.	LOOP: NOP
.	ADD A,2
	CP B
NEXT I	JP LT,LOOP

FIGURE 3.2-3. EXAMPLE OF LANGUAGE EFFECT ON POTENTIAL VOLUME

Theoretically, when an algorithm is implemented in its shortest form, neither operators nor operands are repeated. Thus, the unique number of operators equals the total number:

$$N_1^* = \eta_1^*$$

and the number of unique operands equals the total number:

$$N_2^* = \eta_2^*$$

By substituting these equalities into the Potential Volume equation, the Potential Volume could be rewritten as follows:

$$V^* = (\eta_1^* + \eta_2^*) \log_2 (\eta_1^* + \eta_2^*) \text{ bits}$$

Halstead further stated that the minimum possible number of operators, η_1^*, is two. It must consist of one distinct operator for the name of the function or procedure and another that serves as an assignment or grouping symbol:

$$\eta_1^* = 2$$

An example of this most succinct implementation is a function call that is composed of the function name, an argument, and its parentheses, FUNCTION(X). "FUNCTION" is one operator, the parenthesis pair is another, and "X" is the operand.

Through substitution, the previous equation becomes the following:

$$V^* = (2 + \eta_2^*) \log_2 (2 + \eta_2^*) \text{ bits}$$

here η_2^* represents the number of different input and output parameters. Thus, to calculate the Potential Volume, one only needs to evaluate the number of conceptually unique operands the application will use.

This analysis assumes a definition has been established for a module. A module can be defined trivially, as a single statement; or it can be defined as the next smallest independent function, a subroutine, a program, a subsystem, or an entire system of programs. In structured programming languages, such as Pascal, the main program calls subroutines. The subroutines reduce the amount of code in the main program. Halstead's definition of the ideal algorithm, however, could mean that the source code needed to solve the problem resides in an external routine, thus reducing the word counts in the calling module.

The Potential Volume takes into account the ratio between operators and operands. Ideally, if the algorithm is reduced to its simplest form, the total number of operators and operands, N, would equal three (two operators and one operand). The number of unique operators and operands, η, also would be three.

$$\frac{3\ (N)}{3\ (\eta)}$$

This ratio reduces to 1/1.

In real programs, the total number of operators and operands is always more than the unique number. Some are repeated throughout the program, depending on their function. This divergence from the 1/1 ratio is the amount the application differs from the minimal V. Halstead created the Potential Volume equation so users could develop more efficient algorithms from previously-existing ones.

3.2.6 Program Level

The Program Level measures a person's ability to understand a program. It is based on Volume, is a unitless ratio, and measures the level of sophistication at which a program is written. The Program Level is derived from the previous definitions of Program Volume, V, and Potential Program Volume, V^*:

$$L = V^*/V$$

where only the most succinct expression for an algorithm can have a level of one. Hence, programs with greater volumes have lower Program Level values, so that L is always less than or equal to one.

Rearranging this equation shows the inverse relationship between Program Level and Volume:

$$L/V^* = 1/V$$

By omitting the implementation-independent term, V^*, the relationship becomes the following:

$$L \propto 1/V$$

The lowest Program Level possible is when L = 1, where the program would consist of only one statement. This statement would consist of a call to a procedure. The procedure's statements exist outside the program. Hence, the program would be as terse as possible. Although L = 1 reduces the program to its simplest form, it may not be the best language to use to implement a program.

The Program Level equation can be used by program planners and project managers to estimate the programming effort of a project. It can be used on each program to be written. The sum of the programs would give the estimated effort. To use this measure in a proper context, one gauges how easily the potential language will be understood against how easily the language is implemented.

If a programming group consists of experienced programmers who easily understand the algorithms at a high level without further explanation, a higher level language might be used. If, however, the group consists of mostly new programmers who need to understand the program's algorithms in smaller chunks, a lower level language might be chosen. Halstead implies that lower level languages are "wordier" because they explicitly define more steps. Higher level languages, in contrast, use constructs that contain many "hidden" steps. Thus, higher languages' source code contains "fewer words" or instructions to perform the same functions than those of lower level source. Using a lower level language forces the programmer to "spell out" or further explain the same constructs.

Although Halstead used this analogy to relate Program Level to language, one could argue the converse is true: experienced programmers are the ones who relish the complicated details a low level language like Assembly provides. Inexperienced programmers may be bogged down by the confusing detail of lower level languages.

3.2.7 Estimated Program Level

If the Potential Volume for an application is not known, it is possible to calculate the Program Level from the operator and operand counts as follows:

$$\hat{L} = \left(\eta_1^* / \eta 1\right) \times \left(\eta_2 / N_2\right)$$

where η_1^* is the minimum number of unique operators, and η_1, η_2, and N_2 are as previously defined.

3.2.8 Program Difficulty

The inverse of the Program Level, L, is the Difficulty:

$$D = 1/L$$

As the program's Volume increases, the difficulty also increases. Operands and operators that are used repeatedly will tend to increase the Volume and the Program Difficulty. For example, a Z80 Assembly program written to calculate square roots will have a much larger Volume than a FORTRAN program written to calculate the same. The Assembly program would have a higher difficulty rating than the FORTRAN program.

3.2.9 Intelligence Content

The Intelligence Content is the result of multiplying the Estimated Program Level and the Volume:

$$I = \hat{L} \times V \text{ bits}$$

The Intelligence Content is independent of the language in which an algorithm is expressed (Halstead 1977). Halstead claims this property is invariant because it is constant within ±10 percent in studies conducted using different languages. As the Language Level decreases, the Volume increases.

To use directly measurable quantities to implement the Intelligence Content, the equation should be expanded as follows.

1. Because $V = N\log_2\eta$, substitute this quantity into the original equation to obtain the following:

   $$I = \hat{L} \times N \log_2 \eta \text{ bits}$$

2. Because $\hat{L} = (\eta_1^*/\eta_1) \times (\eta_2/N_2)$, substitute this quantity into the result of step 1:

   $$I = [(\eta_1^*/ \eta_1) \text{ x } (\eta_2/N_2)] \text{ x } N \log_2 \eta \text{ bits}$$

3. Because $N = N_1 + N_2$, and $\eta = \eta_1 + \eta_2$, substitute these quantities into the result of step 2:

   $$I = [(\eta_1^*/ \eta_1) \text{ x } (\eta_2/N_2)] \text{ x } [(N_1 + N_2) \log_2 (\eta_1 + \eta_2)] \text{ bits}$$

From this last equation, simply substitute the value obtained for operator and operand counts.

3.2.10 Program Purity

When Halstead created his Length and Intelligence Content equations and attempted to prove their invariance, the tests initially were conducted in laboratory settings. The programs used in the tests were written by expert programmers who were fluent in the language and who followed good programming techniques. Hence, the tests did not entirely reflect real-world situations. The correlations were lower and the relationships were weaker when the metrics were used on programs written by novices.

To account for the weakened metric relationships, he classified six areas where novice programs revealed "impurities". These areas were categorized into six classes of impurities. When the impurities are removed, the estimate of the Potential Volume is improved. The Potential Volume equation is as follows:

$$V^* = V \times \hat{L}$$

Consider the following code:

```
SUBR(RADIUS WIDTH CAREA)
```

where η_1 is 2 and η_2 is 3. N_1 is then 2 and N_2 is 3. Using these counts in the Potential Volume equation yields a V^* of 11.6. This is the standard by which each impurity discovered in the next section is measured. The amount of deviation from 11.6 is calculated. Once a programmer identifies impurities by recognizing the types (as discussed in the following section), the algorithm must be recoded to remove the impurity, showing an improvement in the Potential Volume.

3.2.10.1 Impurity 1 - Complementary Operations

Complementary operations occur when two complementary operators are successively applied to the same operand. Consider the following code:

```
RADTOT = RADIUS + WIDTH
CAREA  = RADTOT * RADTOT + RADTOT - RADTOT
```

In this example, the complementary operators are the plus and minus signs. The affected operand is "RADTOT". To calculate the estimated Potential Volume with the impurity present, first count unique operators and operands and total numbers of each. The number of unique operators, η_1, is five (the plus, minus, multiplication, and equal signs, and a statement separator). The total number of operators, N_1, is seven (two plus signs, one minus sign, one multiplication sign, two equal signs, and one statement separator). The number of unique operands, η_2, is four ("RADTOT", "RADIUS", "WIDTH", and "CAREA"). The total number of operands, N_2, is eight (five "RADTOT"s, one "RADIUS", one "WIDTH", and one "CAREA"). Since the Potential Volume is the product of the Program Volume and Program Level, first calculate these quantities separately.

Substitute the counts into the Program Volume equation:

$$\begin{aligned} V &= (N_1 + N_2) \log_2 (\eta_1 + \eta_2) \\ &= (7 + 8)\log_2(5 + 4) \\ &= 15\log_2 9 \\ &\approx 47.5 \text{ bits} \end{aligned}$$

Substitute the counts into the Estimated Program Level equation:

$$\begin{aligned} \hat{L} &= [(2/\eta_1) \times (\eta_2 / N_2)] \\ &= [(2/5) \times (4/8)] \\ &= [(0.4) \times (0.5)] \\ &= 0.2 \end{aligned}$$

Multiply the Program Level by the Program Volume:

$$\begin{aligned} V &= V^{*} \times \hat{L} \\ &= 47.5 \times 0.2 \\ &\approx 9.5 \text{ bits} \end{aligned}$$

This is not a very close estimate of the true Potential Volume. To purify the version of code, the complementary operations are removed, leaving the original pure form:

```
RADTOT = RADIUS + WIDTH
CAREA  = RADTOT * RADTOT
```

Calculate the new Potential Volume. The equations are combined for brevity:

$$\begin{aligned} V^{*} &= V \times \hat{L} \\ &= (N_1 + N_2)\log_2(\eta_1 + \eta_2) \times [(2/\eta_1) \times (\eta_2/N_2)] \\ &= (5+6)\log_2(4+4) \times [(2/4) \times (4/6)] \\ &\approx 11.0 \text{ bits} \end{aligned}$$

Note that 11.0 is closer to 11.6 than 9.5 is. The measurement should reflect the Potential Volume measurement set for the standard. In this case, the most succinct algorithm has a Potential Volume of 11.6.

Optimizing compilers for most programming languages will remove the complementary operations impurity automatically. However, if this impurity is not removed, it will affect the software metric correlations. Note the impurities do not always produce an implementation whose Potential Volume is excessive. A program is considered impure to the extent that it violates structured programming techniques. Halstead uses impurity specifically to mean anything that upsets the relationship among V, L, and I.

After specific modules that significantly deviate from the expected correlations have been isolated, these modules are analyzed for impurities. In the examples discussed, it was found that the "pure" version of the code had a volume of 11.6 bits. Hence, when the impurities were removed, the resulting code had a Potential Volume closer to the standard, 11.6 bit program. When modules are analyzed for impurities, they are judged against a presumed pure implementation of an algorithm.

3.2.10.2 Impurity 2 - Ambiguous Operands

This impurity occurs when the same operand represents different quantities within a program. It is typically harder to understand a program that uses operands in this way because the operands must be viewed within a context to understand the immediate meaning.

The following code demonstrates this impurity:

```
CAREA = RADIUS + WIDTH
CAREA = CAREA * CAREA
```

The first statement assigns a value to "CAREA". The second statement uses the result stored in "CAREA" to determine a new value, which also is stored in "CAREA".

The operator and operand counts yield the following:

$$\eta_1 = 4$$
$$\eta_2 = 3$$
$$N_1 = 5$$
$$N_2 = 6$$

Substituting these values into the Program Volume and Level equations gives the following:

$$\begin{aligned} V &= (N_1 + N_2) \log_2 (\eta_1 + \eta_2) \\ &= (5 + 6)\log_2(4 + 3) \\ &= 11 \log_2 7 \\ &\approx 30.9 \text{ bits} \end{aligned}$$

and

$$\begin{aligned} \hat{L} &= [(2/\eta_1) \times (\eta_2/N_2)] \\ &= [(2/4) \times (3/6)] \\ &= [(0.5) \times (0.5)] \\ &= 0.25 \end{aligned}$$

To determine the Potential Volume, multiply the Estimated Program Level by the Program Volume:

$$\begin{aligned} V^* &= V \times \hat{L} \\ &= 30.9 \times 0.25 \\ &\approx 7.7 \text{ bits} \end{aligned}$$

Clearly, this is not very close to 11.6. When the impurity is removed, the statements become:

```
RADTOT = RADIUS + WIDTH
CAREA  = RADTOT * RADTOT
```

giving a Potential Volume of 11.0 bits, as calculated in the previous section.

Some higher level languages provide a construct that allows operands to be stored in the same memory location, e.g., FORTRAN has the "EQUIVALENCE" statement. Using the "EQUIVALENCE" statement saves memory storage, yet also reduces the program's clarity. If the compiler does not have this capability, the estimated Potential Volume will overestimate the actual Potential Volume.

3.2.10.3 Impurity 3 - Synonymous Operands

Synonymous operands occur when two or more variables are used to name the same quantity, as shown in the following code:

```
RADTO1 = RADIUS + WIDTH
RADTO2 = RADIUS + WIDTH
CAREA  = RADTO1 * RADTO2
```

The expression "RADIUS + WIDTH" is assigned to two operands: "RADTO1" and "RADTO2". Substituting operator and operand counts into the Potential Volume equation yields the following:

$$\begin{aligned}V^* &= V \times \hat{L}\\ &= (N_1 + N_2)\log_2(\eta_1 + \eta_2) \times [(2/\eta_1) \times (\eta_2/N_2)]\\ &= (8 + 9)\log_2(4 + 5) \times [(2/4) \times (5/8)]\\ &\approx 15.0 \text{ bits}\end{aligned}$$

Clearly, this is not very close to 11.6.

3.2.10.4 Impurity 4 - Common Subexpressions

A common subexpression occurs when a combination of terms is used several places in a program, instead of assigning a new variable to the combination.

Consider the following code:

```
CAREA = (RADIUS + WIDTH) * (RADIUS + WIDTH)
```

The subexpression "(RADIUS + WIDTH)" is used twice in the expression. The Potential Volume for this statement is:

$$\begin{aligned}V^* &= V \times \hat{L}\\ &= (N_1 + N_2)\log_2(\eta_1 + \eta_2) \times [(2/\eta_1) \times (\eta_2/N_2)]\\ &= (6 + 5)\log_2(4 + 3) \times [(2/4) \times (3/5)]\\ &\approx 9.3 \text{ bits}\end{aligned}$$

Subexpression redundancy would be eliminated by placing the result of the subexpression into a new variable, multiplying the new variable by itself, and placing the result into "CAREA":

```
RADTOT = RADIUS + WIDTH
CAREA  = RADTOT * RADTOT
```

The revised version has a Potential Volume of 11.0 bits.

Some optimizing compilers eliminate common subexpressions. If the compiler does not handle this situation, the programmer must check for common subexpressions.

3.2.10.5 Impurity 5 - Unwarranted Assignment

This impurity occurs when a common subexpression has been assigned a unique name, but the unique name is never used again in the program. Thus, the assignment is unnecessary.

The following code contains this impurity:

```
RADTO1 = RADIUS + WIDTH
CAREA  = RADTO1**2
```

The result of the first statement is placed into "RADTO1". In the next statement, the result is placed in "CAREA". Assuming "RADTO1" is not used in the program again, this variable assignment is unnecessary.

The Potential Volume is as follows:

$$
\begin{aligned}
V^* &= V \times \hat{L} \\
&= (N_1 + N_2)\log_2(\eta_1 + \eta_2) \times [(2/\eta_1) \times (\eta_2/N_2)] \\
&= (5+6)\log_2(4+5) \times [(2/4) \times (5/6)] \\
&\approx 14.5 \text{ bits}
\end{aligned}
$$

Removing the impurity yields:

```
CAREA = (RADIUS + WIDTH)**2
```

where the Potential Volume is 12.0 bits.

3.2.10.6 Impurity 6 - Unfactored Expressions

Expressions that have been factored are easier to comprehend. Consider the following code:

```
NEW = RADIUS * RADIUS + 2 * RADIUS * WIDTH + WIDTH * WIDTH
```

Unfactored, the Potential Volume is as follows:

$$
\begin{aligned}
V^* &= V \times \hat{L} \\
&= (N_1 + N_2)\log_2(\eta_1 + \eta_2) \times [(2/\eta_1) \times (\eta_2/N_2)] \\
&= (7+8)\log_2(3+4) \times [(2/3) \times (4/8)] \\
&\approx 14.0 \text{ bits}
\end{aligned}
$$

Removing the impurity, yields:

```
NEW = (RADIUS + WIDTH)**2
```

which is 12.0 bits.

This impurity class rarely occurs because most programmers automatically create factored expressions in their programs. Also, while the other classes can be purified by sophisticated compilers, this impurity class must be detected by a program reader. The program reader must assess the degree of impurity to decide whether the module should be recoded.

3.2.10.7 General Analysis on Impurity Classes

Table 3.2-3 displays variants of an algorithm that calculates the area of a circle. To calculate the area of a circle, each listed algorithm's value is then multiplied by π. This statement has been omitted for simplicity's sake.

The first algorithm listed is the theoretical, pure version. It is the simplest form, a called subroutine. The second is the standard, referenced, pure algorithm. The four succeeding algorithms reduce to the standard reference once the impurity is removed. (The preceding sections explained how the numbers were obtained for each impure version.) In table 3.2-3, classes 5 and 6 were omitted because their revised algorithms did not reduce to the same reference.

The impure algorithms deviated from the standard reference in the negative or positive direction. Due to the increased Volume, the algorithm containing synonymous operands, Impurity #3, deviated most from the reference.

TABLE 3.2-3. ALGORITHM FORMS

Algorithm	Impurity Class	η_1	η_2	N_1	N_2	Volume V	Level L	LxV
AREA (RADIUS WIDTH CAREA)	^NONE	2	3	2	3	12	1.00	11.6
RADTOT = RADIUS+WIDTH CAREA = RADTOT*RADTOT	NONE	4	4	5	6	33	0.34	11.2
RADTOT = RADIUS+WIDTH CAREA = RADTOT*RADTOT+ RADTOT-RADTOT	1	5	4	7	8	48	0.20	9.5
CAREA = RADIUS+WIDTH CAREA = CAREA+CAREA	2	4	3	5	6	31	0.25	7.7
RADTO1 = RADIUS+WIDTH RADTO2 = RADIUS+WIDTH CAREA = RADTO1*RADTO2	3	4	5	8	9	54	0.28	15.0
CAREA = (RADIUS+WIDTH)* (RADIUS+WIDTH)	4	4	3	6	5	31	0.30	9.3
^ Theoretical Pure Version								

3.2.11 Programming Effort

Programming Effort attempts to measure the mental work required to create a program. It is defined as:

$$E = V/L \text{ discriminations}$$

where E is the effort, or the total number of elementary mental discriminations, V is the Volume, and L is the Program Level.

Consider the BASIC code that calculates the circumference of a circle, as shown in figure 3.2-4.

```
LET PI = 3.1416
INPUT R
LET C = 2*PI*R
PRINT C
```

FIGURE 3.2-4. "BASIC" CODE EXAMPLE

First, count the operators and operands:

$\eta_1 = 5$ ("INPUT", "LET" and "=" pair, "*", "PRINT", the line separator)
$\eta_2 = 5$ ("R", "PI", the constant 3.1416, "C", the constant 2)
$N_1 = 9$ ("INPUT", 3 line separators, 2 "LET" and "=" pairs, 2 "*"s, "PRINT")
$N_2 = 8$ (2 "R"s, 2 "PI"s, the constant 3.1416, 2 "C"s, the constant 2)

The "LET" and "=" constitute a combination operator; one does not occur without the other. Like a set of parentheses, the pair counts as a single operator.

Calculate the Program Volume:

$$\begin{aligned} V &= (N_1 + N_2) \log_2 (\eta_1 + \eta_2) \\ &= (9 + 8)\log_2(5 + 5) \\ &= 17\log_2 10 \\ &\approx 56.5 \text{ discriminations (where the alternate interpretation of volume is used.)} \end{aligned}$$

Substitute the counts into the Estimated Program Level equation:

$$\begin{aligned} \hat{L} &= [(2/\eta_1)\times(\eta_2/N_2)] \\ &= [(2/5)\times(5/8)] \\ &= [(0.40)\times(0.625)] \\ &= 0.25 \end{aligned}$$

Calculate the Programming Effort:

$$\begin{aligned} E &= V/\text{ discriminations} \\ &= 56.5 \text{ discriminations}/0.25 \\ &= 226 \text{ discriminations} \end{aligned}$$

One of the assumptions underlying E is that the programmer is fluent in the language in which the algorithm is to be written, already understands the algorithm, and is not distracted.

Programming Effort is based on the idea that a programmer uses a mental binary search of the Vocabulary of unique words in order to select each word to be used in the implementation. Further, the process of selecting each word requires an effort that is related to how well the programmer conceptually understands the algorithm to be written.

The Program Difficulty is the reciprocal of the Program Level. Hence, a program of large Volume requires a higher level of effort to write than one of a lower Volume, assuming both programs have the same Potential Volume.

3.2.12 Estimated Programming Time

Programming Time is an estimate of the amount of time it took a programmer to write a program. The equation is:

$$\hat{T} = E/S \text{ seconds}$$

where E is the Programming Effort and S is the Stroud Number. The Stroud Number is a value the implementor arbitrarily chooses from a range of $5 \leq S \leq 20$ per second. For concentrating programmers who are fluent in the language used and who are provided with nonprocedural problem statements, a value of 18 discriminations per second has been used successfully (Funami and Halstead 1976). The range is obtained from psychological studies that determined the total number of elementary mental discriminations a person could perform per second.

Consider the example given in section 3.2.11, Programming Effort. The Effort was 226 discriminations. Using this value and a Stroud Number of 15 (assuming slightly less than optimal programming working conditions) yield the following results:

$$\begin{aligned} \hat{T} &= E/S \\ &= 226 \text{ discrimination}/15 \text{ discriminations per second} \\ &\approx 15 \text{ seconds} \end{aligned}$$

It took the programmer approximately 15 seconds to write the program. Halstead also suggests that it should take a programmer the same amount of time to read and understand these implemented lines of code.

Without calculating the Program Volume and Level, the operator and operand counts can be directly substituted into an expanded form of the equation:

$$\hat{T} = \frac{\eta_1 N_2 \left(\eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2\right) \log_2 \eta}{2\eta_2 S} \text{ seconds}$$

where all parameters on the right are directly measurable, except for the Stroud Number.

Consider another example:

```
RADTOT = RADIUS + WIDTH
CAREA   = RADTOT * RADTOT
```

where $\eta_1 = 5$, $\eta_2 = 4$, $N_1 = 7$, and $N_2 = 8$.

Substitute the count values and a value of S of 15 into the following equation:

$$\hat{T} = \frac{\eta_1 N_2 (\eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2) \log_2 \eta}{2\eta_2 S} \text{ seconds}$$

$$= \frac{5 \times 8 (5 \log_2 5 + 4 \log_2 4) \log_2 9}{2 \times 4 \times 15} \text{ seconds}$$

$$\approx 20.7 \text{ seconds}$$

This equation implies that it took the programmer approximately 20.7 seconds to write the program.

The Programming Time equation assumes that the programmer's attention is entirely undivided, and that the range of values found for the Stroud Number applies to programming activity. If the programmer's time is not undivided, the result can be scaled to reflect the amount of time spent on the task. The Stroud Number can be fixed by evaluating S for the programmer. It also assumes that not only do people make a constant number of mental discriminations per second, but that they are capable of making a constant number of discriminations. It also assumes that programs do not contain any impurities.

While this relationship shows the estimated time it takes to understand a program, it can also be used as a guideline to predict the time required to implement future applications. First, use the following equation to predict the effort if the Potential Volume and Language Level are known:

$$E = V^{*3}/\lambda^2$$

Then use this result in the Programming Time equation.

3.2.13 Language Level

This equation calculates how efficiently a particular language represents algorithms:

$$\lambda = LV^*$$

where L is the Program Level and V^* is the Potential Volume. For programs written in a particular language, as the Potential Volume increases the Program Level decreases proportionally. For a particular language, λ remains constant.

To calculate the Language Level using direct quantities, the following relationship may be used:

$$\lambda = L^2V$$

where L is the Program Level and V is the Volume.

Consider the code shown earlier in figure 3.2-4. The Program Volume was 56.5 bits, and the Program Level was 0.25. The Language Level is as follows:

$$\begin{aligned}\lambda &= L^2V \\ &= (0.25)^2 \text{ x } 56.5 \\ &\approx 3.53\end{aligned}$$

This metric allows managers and project leaders to select an implementation language for an application more easily. For example, while FORTRAN could be used to develop a record keeping application, COBOL would be more efficient. COBOL would allow the application to be designed using less code; hence, COBOL is a more powerful language to use in this instance.

Shen, Conte, and Dunsmore (1983) analyzed a number of programs written in different languages. Language Level values were determined for several languages, as shown in table 3.2-4. The published values indicate a logical progression: Assembly has the lowest level, PL/1 the highest; Assembly has the most Volume, PL/1 the least.

TABLE 3.2-4. LANGUAGE LEVEL VALUES

Language	λ
PL/1	1.53
Algol	1.21
FORTRAN	1.14
CDC Assembly	0.88

3.2.14 Number of Bugs

The number of bugs in an implementation can be estimated by using the following equation:

$$\hat{B} = V/E_0 \text{ bugs}$$

where V is the Program Volume in number of discriminations, and $1/E_0$ is the average number of discriminations a person is likely to make for each bug introduced into the code.

E_0 is determined from a programmer's previous work. Halstead determined through experiments that 3,200 discriminations per bug is a typical value.

Using the volume obtained in the BASIC code example, and 3,200 discriminations per bug as a typical value, the number of bugs in the example would be as follows:

$$\begin{aligned}\hat{B} &= V/E_0 \text{ bugs}\\ &= 56.5 \text{ discriminations}/3{,}200 \text{ discriminations per bug}\\ &= 0.02 \text{ bugs}\end{aligned}$$

The number of bugs in this case should be less than one because the example contained only a few simple statements. Had the volume been greater, say 200,000 bits, the number of expected bugs would have been approximately 63. This equation gives meaningful results for real-life applications where the volume is much greater than the examples provided here.

3.2.15 General Analysis of Halstead's Software Metrics

Criticism of Halstead's software metrics falls into two categories: criticism of the software science theory, and criticism of the empirical results used to confirm the theories. In the former case, researchers look for consistent logic among the equations, analyze whether the assumptions used to derive them are valid, and determine whether the metrics measure the software attributes they purport to measure. In the latter, empirical results are analyzed and critiqued.

To determine whether the metrics measure these software attributes, experiments must be conducted. From early experiments, several criticisms pointed out weaknesses in the experimental evidence. Later experiments have provided better results.

3.2.15.1 The Theory Behind Halstead's Software Science

In general, the original theory addressed metrics that would analyze algorithms, not programs. Since algorithms only contain operators and operands, it was easy to form a counting strategy because classifying each was fairly straightforward.

In certain languages it is difficult to determine whether some tokens should be operators or operands (Lassez et al. 1981). The meaning may depend on how the token is used at execution time, or on information given in the declaration section.

Since operator and operand counts originally did not include declaration sections and other non-executable code, the counts could yield inaccurate results if they were used to measure effort or time. In a COBOL application, for example, the extensive declaration section would not be counted. Yet this section requires a significant amount of programmer effort since it can account for a major portion of a program.

To address this problem, the developer should establish the rules for defining the operators and operands of the implemented language. If a large application uses several languages, rules should be established for each language based on the constructs of that language. While most measures purportedly are language-independent, it is critical to define the counting strategy to accommodate the syntax peculiarities of a particular language. For a certification package, the CE should request a written list of the rules followed in the counting procedures. If automatic counting tools or analyzers are developed for this purpose, documentation on the tools should include a description of the counting methods used.

While mathematical derivations have been performed on all of Halstead's equations, researchers have questioned the assumptions underlying some of the variables. Researchers have also questioned the derivations. For example, Halstead bases many of the relationships on the assumption that programmers use the "binary search" method when they create algorithms or programs. He further contends the binary search is the most efficient to use on an ordered list (Halstead 1977). Halstead used this relationship in many of his fundamental equations, including the Program Length and Program Volume.

Critics question the validity of the use of a binary search mathematical component in some of Halstead's equations. Although a software program may use a binary search to select items, it is not clear that people, in general, use mental processes that can be likened to such a search. Nor is it clear that programmers, in particular, use binary searches when selecting their Vocabulary for writing a program. However, experimental evidence suggests that Program Length and Program Volume equations (which include this assumption) provide monotonic, repeatable relationships, providing the counting methods employed are used consistently.

The Stroud Number has been derived from psychological studies. It is the number of elementary discriminations a person can perform per second. The Programming Time equation is based on the assumption that the Stroud Number is valid. The value of the Stroud Number, however, does not change the basic relationships. Tailoring the value simply provides a closer approximation of actual programming time.

The explanations for some of the derivations are not given. In the Implementation Length equation, Halstead provides no reason for dividing a program of Length N into N/ η substrings. While this equation cannot be justified theoretically, its validity has been established through experiments.

3.2.15.2 Empirical Evidence

Although the theoretical foundation of some of Halstead's metrics is suspect, it is possible to determine whether the formulas approximate real values by empirically validating the metrics. The number of empirical studies performed over 15 years on his metrics substantiate their utility.

Researchers have criticized early experiments conducted by Halstead and others for the following reasons:

- The sample sizes were too small, so there was not enough data to infer much from the experiment. Also, the small sample constrained the use of parametric statistics.
- The programs were small. All except one were single modules containing fewer than 50 lines.
- The programmers were generally college students. Hence, the results may not apply to professional programmers.

While these first criticisms were well-founded because the work had been conducted in a university setting, further testing has validated most of Halstead's equations.

The Estimated Length equation is a good indicator of the Implementation Length. In analyzing 1,637 modules, the relative error between N and was less than six percent, although the error can be larger for separate modules. This measure is most accurate when programs range in size from 2,000 to 4,000 words (Shen, Conte, and Dunsmore 1983). For larger applications, the error of the length equation can be minimized by dividing a program into separate modules and then summing the individual module estimates. For programs that are much smaller than the minimum range listed, inlining subroutines may help to create larger modules that would minimize the error.

Ottenstein (1981) conducted a study based on 11 programs written by first year graduate students for a compiler class. She tested the correlation between the Estimated Length and the Implementation Length. The study used three counting methods. In the first method, operators and operands were computed for each routine and then were summed over the entire module. If an operator or operand occurred in two different routines, it was counted as unique in each. N_1 and N_2 were calculated by counting the total number of occurrences of operators and operands in the module.

The second method was similar to the first but did not consider routine boundaries. If a variable was global, it was counted only once. Operators were considered in the same way. The third method estimated the number of operators used in the program, leaving unknown only the number of distinct operators that would be used in the code. The results from this study indicate that is a good estimator of N, as shown in table 3.2-5.

In table 3.2-5, N1 represents the correlation between N and when the first counting method was used. N2 represents the correlation found when the second counting method was used; N3 represents the correlation using the third method.

The results from this study indicate a high correlation exists between Halstead's Estimated Length and Implementation Length. The average correlation coefficient was 0.968 (significant at the 0.001 level) between the Estimated Length and the Implementation Length.

TABLE 3.2-5. LENGTH CORRELATION RESULTS

N	$\hat{N}$
N1	0.976
N2	0.971
N3	0.951

Bulut and Halstead (1974) analyzed 429 FORTRAN programs from a library at the Purdue University Computing Center. The total sample had 242,990 occurrences of operators and operands, with individual program lengths ranging from seven to 10,661 occurrences. The coefficient of correlation between the observed and predicted lengths was 0.95.

Elshoff (1976) tested the length estimator on 154 programs. The correlation between actual and estimated lengths was 0.985 for structured programs and 0.976 for non-structured programs, further validating this metric.

Studies have been performed to test the number of bugs estimate, Akiyama's study (1971) involved a six module software project. The correlation coefficient between number of predicted bugs and actual bugs was 0.95 (significant at the 0.01 level). Another study on implementing an assembler in PL/1 (Ottenstein 1981), calculated a correlation coefficient between the predicted number of bugs and actual number of bugs of 0.828 (significant at the 0.02 level).

3.2.16 Conclusions

Although critics have raised serious questions about the validity of some of the assumptions underlying Halstead's metrics, the metrics have been substantiated by almost two decades of empirical studies. In spite of the theoretical questions on the derivations of some of the formulas and on the assumptions built into them, field studies have supported their validity.

3.3 McCabe's Cyclomatic Complexity Metric

McCabe developed a metric that allows developers to identify modules that are difficult to test or maintain. As a result, he developed a software metric that equates complexity to the number of decisions in a program. Developers can use this measure to determine which modules of a program are overly-complex and need to be re-coded. The metric can also be used to define the minimal number of test cases that are needed to adequately test the program's paths.

McCabe bases the metric on graph theory. Because the metric is discussed in graph theory terms, fundamental graph concepts are presented here.

The flow of a program can be represented by a program control graph. The elements of the graph are nodes (or vertices) and edges (or arcs). A node is a sequential block of code. Control flow may only enter the first statement of the block and may only branch from the last statement. The nodes represent the executable statements of a program and the edges correspond to the flow of control between nodes. The top node of the graph, from which other nodes stem, is the entry node. The bottom node defines the end of the block of code, as illustrated in figure 3.3-1.

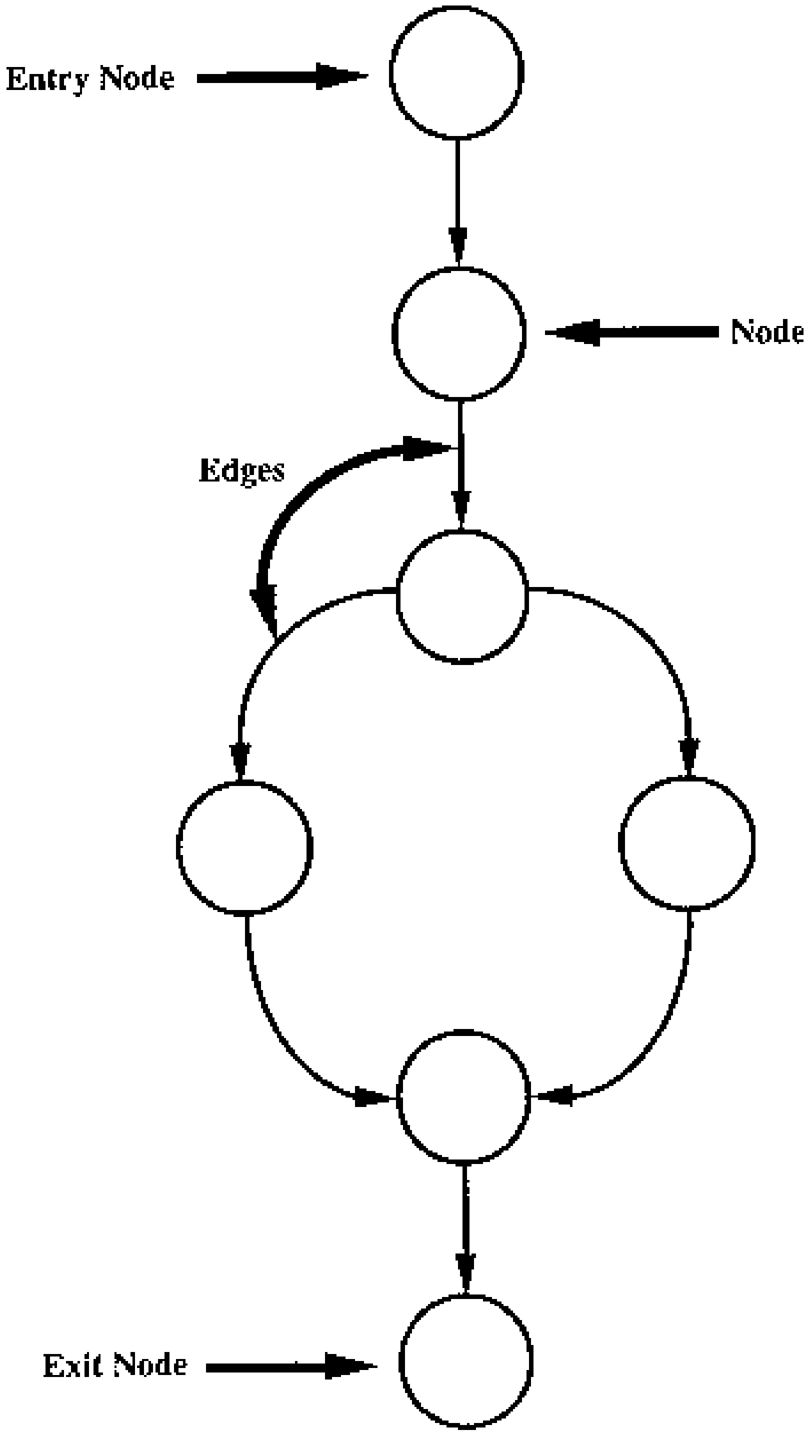


FIGURE 3.3-1. A PROGRAM CONTROL GRAPH AND ITS ELEMENTS

Consider the FORTRAN code (McCabe 1982) shown in figure 3.3-2. The column of numbers at the left represents a way of partitioning the code into nodes. Each node is assigned a number. Note that only the executable code qualifies. The "SUBROUTINE SEARCH" statement and the declarations are not placed in nodes. Figure 3.3-3 shows the graph representing this code.

The metric can be calculated either from evaluating the graphs of the program or from evaluating the program's statements. The graph analysis method is presented first to show the theory behind the metric. The second method is presented next to show how the Cyclomatic Complexity can be computed directly from code.

```
      SUBROUTINE SEARCH(STRING,PTR,BOOL,COUNT)
      INTEGER A,B,C,X
      INTEGER STRING(80), COUNT, PTR
      LOGICAL BOOL
      DATA A,B,C,X/"101","102","103","130"/
      COUNT=0
      BOOL=.FALSE.
  1   IF STR(PTR) .EQ. A
  2       THEN PTR = PTR+1
  3       WHILE STR(PTR) .EQ. B .0R. STR(PTR) .EQ. C DO
  4               COUNT=COUNT+1
  5               PTR=PTR+1
  6       END
 7,8      IF (STR(PTR) .EQ. X)) BOOL=.TRUE.
  9    ENDIF
 10    RETURN
```

FIGURE 3.3-2. A FORTRAN CODE EXAMPLE

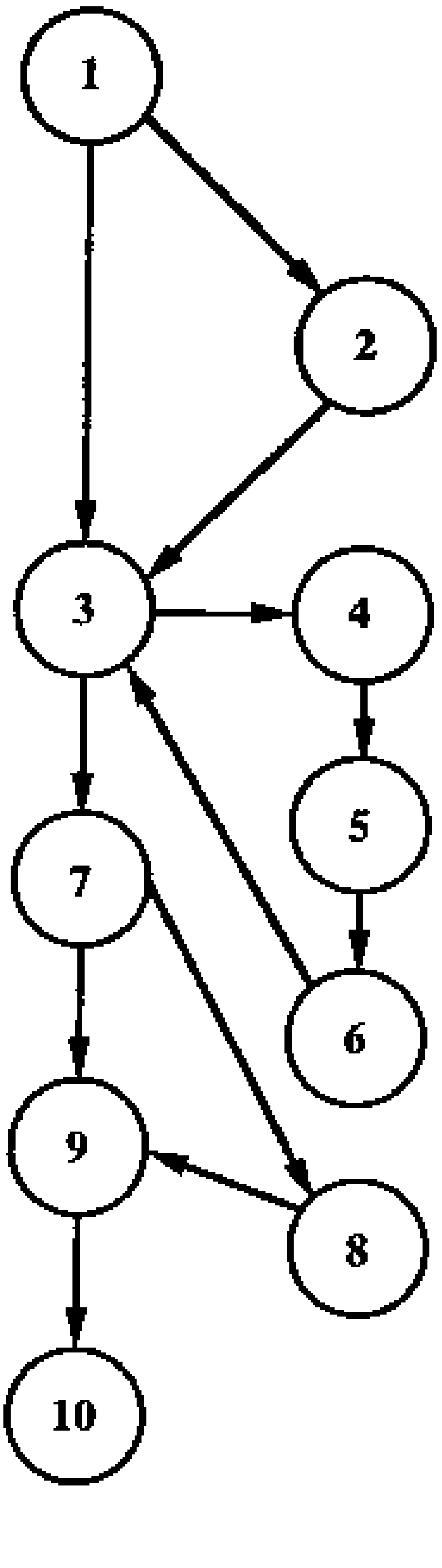


FIGURE 3.3-3. A PROGRAM CONTROL GRAPH OF FORTRAN CODE

3.3.1 Method 1 - Evaluating Program Control Graphs

3.3.1.1 Cyclomatic Complexity

The general formula for calculating the Cyclomatic Complexity is as follows:

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

where "v(G)" is the Cyclomatic Complexity, "e" is the number of edges, "n" is the number of nodes, and "p" is the number of modules. A module can always be represented by a connected graph. A connected graph is one in which any two nodes of the graph are connected to each other by a chain of nodes and edges. However, this formula is based on a theorem that applies to strongly-connected graphs.

In a strongly-connected graph, for any two nodes, A and B, there is a path from node A to node B and from node B to node A. If an edge were added to a program control graph, as shown in figure 3.3-4, it would be strongly-connected. Since program control graphs do not have this additional edge, a count of edges would always be one short. Thus, the formula above adds an assumed edge so that the formula can be used on program control graphs (which are not strongly-connected).

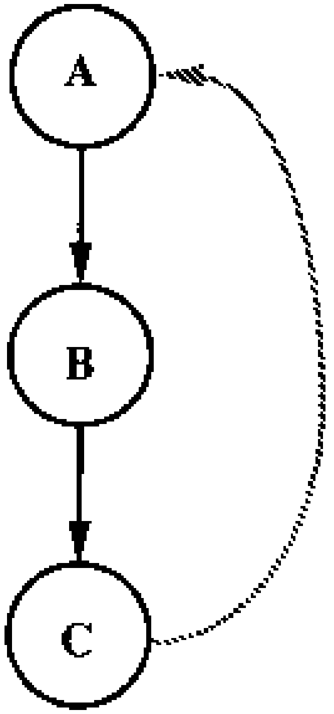


FIGURE 3.3-4. A STRONGLY-CONNECTED GRAPH

The graph in figure 3.3-4 has three nodes and two edges, not counting the dashed line. (Without the dashed line, the graph is a program control graph.) Since the graph is connected, it consists of one module. Hence, the Cyclomatic Complexity, v(G), is as follows:

$$\begin{aligned} v(G) &= e - n + 2p \\ &= 2 - 3 + 2 \\ &= 1 \end{aligned}$$

Each variable can be counted and summed for all modules of a program. The counts are then used in the formula to yield a Cyclomatic Complexity for the program. Or, if the entire formula is applied to each module, the v(G) number for each module is added to all the others to yield the program's v(G). The formula is additive: either method of applying the formula will yield the same Cyclomatic Complexity result.

To illustrate this equation's additive property, consider the graph that depicts three modules shown in figure 3.3-5.

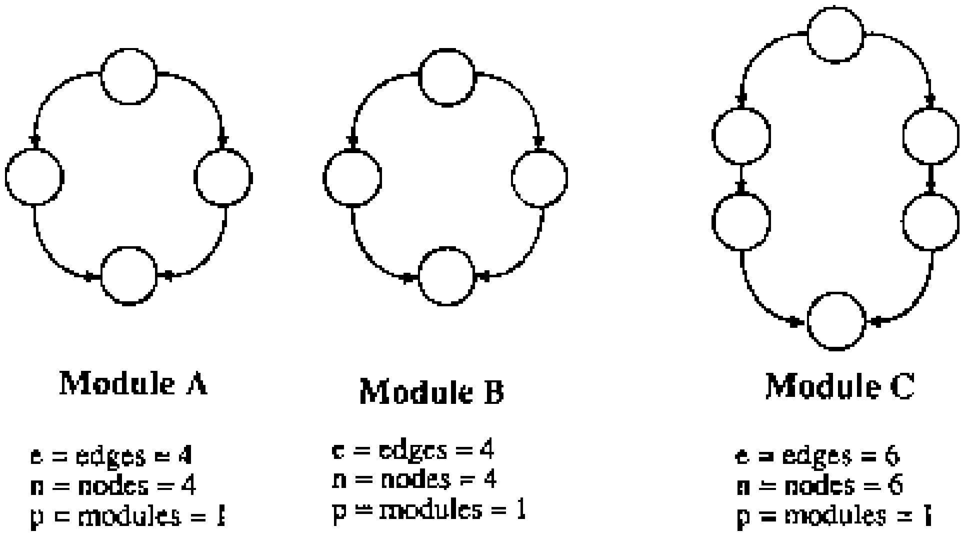


FIGURE 3.3-5. THREE MODULES

The Cyclomatic Complexity for all three modules can be calculated two ways. To calculate v(G) using total counts in the formula, sum the edges or nodes for all modules and then use the sums in the formula. Module A has four edges, module B has four, and module C has six. So, the sum of the edges, "e", is 14. Module A has four nodes, module B has four, and module C has six. So, the sum of nodes, "n", is 14. Module A is one module, module B is one, and module C is one. So, the sum of the modules, "p", is three. Using these totals in the formula for v(G) gives a Cyclomatic Complexity of six.

The alternative method involves using the formula on each module and adding the v(G) obtained for each module. Module A has four edges, four nodes, is one module, and has a Cyclomatic Complexity of two. Module B is identical to A, so its v(G) is also two. Module C has six edges, six nodes, is one module, and has a v(G) of two. Adding the three modules gives an overall v(G) of six, indicating this equation is additive. The Cyclomatic Complexity result is always a positive integer.

At the simplest level, where a graph has one module, the formula reduces to the following:

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

This reduced equation has been used extensively in industry and is often referred to as "McCabe's Complexity Metric", although his general equation actually includes the variable "p". Using the reduced form can lead to incorrect results if the developer is not careful. If the totals of each variable for several modules are used in the equation, the result will underestimate the real Cyclomatic Complexity because the number of modules variable, p, is not present in this shortened equation. For example, although the edges and nodes are fully accounted for in 100 modules, the number of modules mistakenly will be represented as one. To use this shortened form correctly, use the formula on each module, then add the Cyclomatic Complexity numbers of each. To use either formula "e - n + 2" or "e - n + 2p", the developer needs to evaluate the flow graphs that represent the code.

The control graph for any module is represented by determining what constitutes a node and by using some shape to depict it. By convention, circles are generally used. Lines are then drawn to show the edges between nodes.

McCabe defines three types of nodes: predicate, collecting, and function. A predicate node ends with a code statement that branches to multiple locations. A collecting node begins with a labeled code statement to which flow branches. A function node contains code statements that occur between a branch statement and a labeled statement, i.e., it contains code found between the predicate and collecting nodes.

If function nodes are omitted from the graph, v(G) will not be affected, as shown in figure 3.3-6. In this figure, a module is represented by one predicate node, eight function nodes, and one collecting node. Thus, e = 10, n = 10, and p = 1. The Cyclomatic Complexity is two.

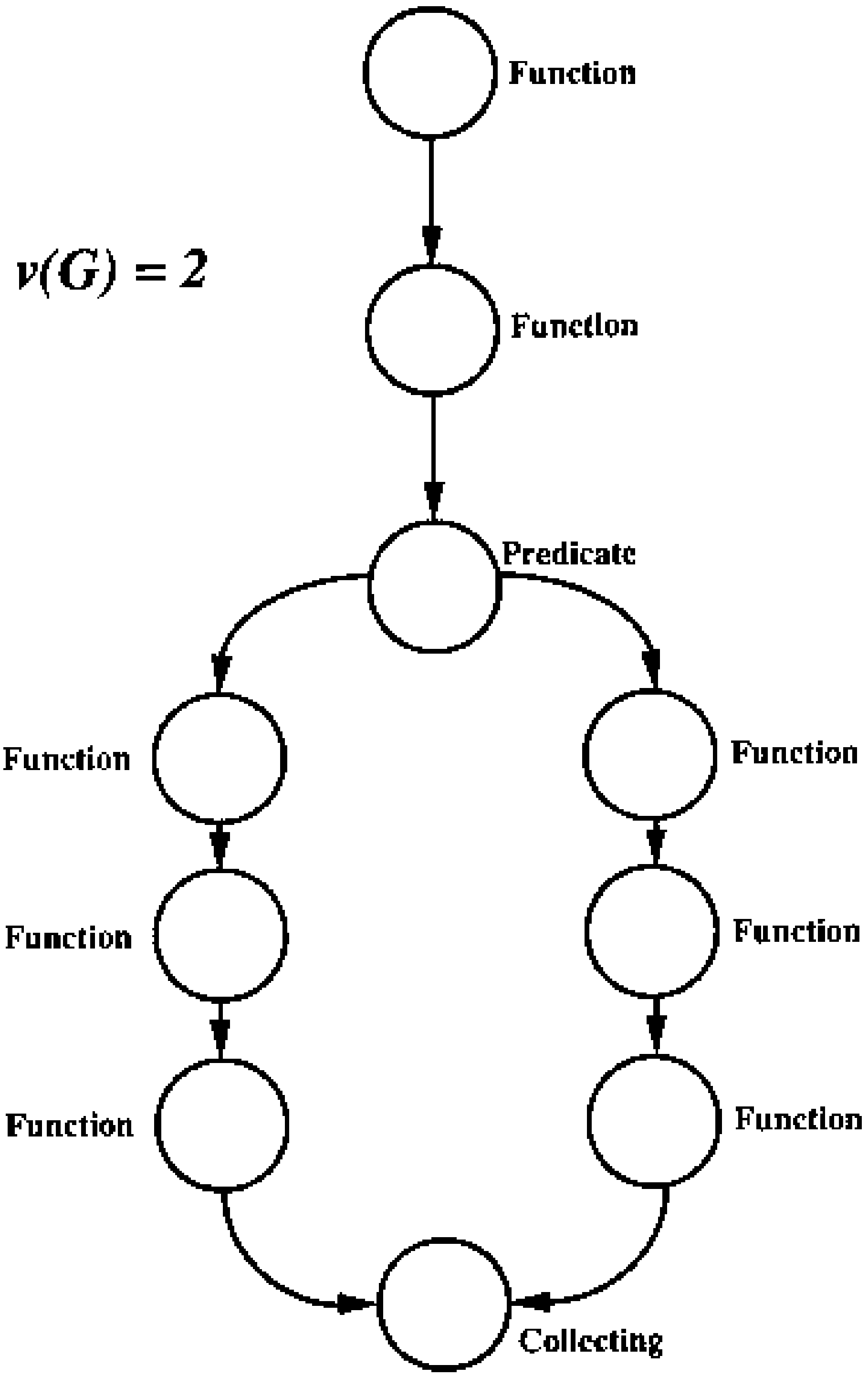


FIGURE 3.3-6. A MODULE REPRESENTED BY THREE TYPES OF NODES

Figure 3.3-7 shows a different graph of the same code. This graph does not include the function nodes. Because the predicate node is a branch condition, it has two edges leaving it.

In figure 3.3-7, e = 2, n = 2, and p = 1. Hence, v(G) is two. The Cyclomatic Complexity is unchanged. This occurs because each function node has exactly one edge associated with it. This is not so with the other types of nodes. The module is now represented as simply a branch leading to a collecting node. The developer decides whether or not to include function nodes in the graphs. Including function nodes will show more detail of the code contained within a module. Omitting function nodes, on the other hand, may emphasize the decision structure of the modules.

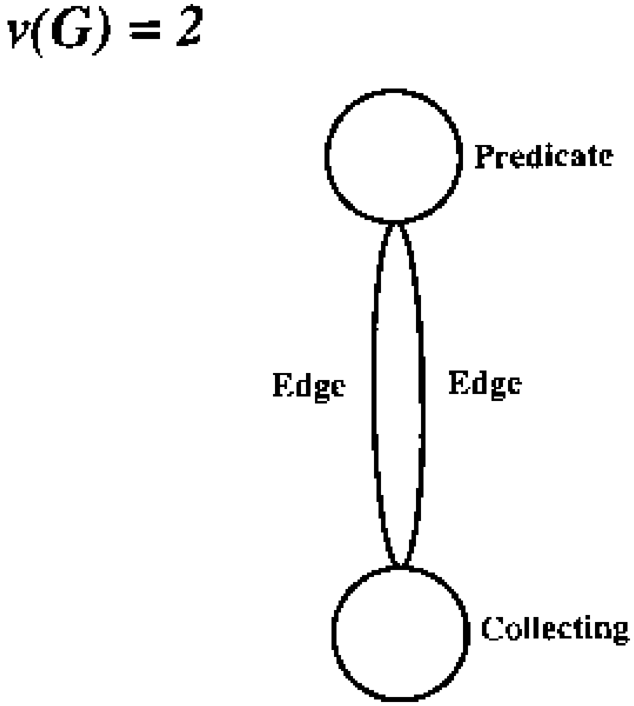


FIGURE 3.3-7. A MODULE REPRESENTED BY TWO TYPES OF NODES

Although function nodes can be omitted without changing the Cyclomatic Complexity of a module, edges, predicate nodes, and collecting nodes must all be counted. If a node is missed, the Cyclomatic Complexity Number will be higher than it should be, since nodes are subtracted from the number of edges. Conversely, if an edge is missed, v(G) will be lower than it should be. If the number of modules is not counted properly, the result will be either higher or lower, depending on whether modules are undercounted or overcounted.

The "e - n + 2p" measure can be applied to one module or to applications containing several modules. Each module's complexity can be determined. The program's overall complexity can then be calculated. This measure only applies to executable source code.

The Cyclomatic Complexity of a module also gives the maximum number of linearly independent paths through it. A path is a route from the entry node to the exit node. For a path to be linearly independent, it must not be a linear combination of any other paths in the graph. Consider the program control graph with entry node "a" and exit node "f", as shown in figure 3.3-8 (McCabe 1976). (The dashed line represents an imagined path allowing node "f" to branch back to the entry node, "a". This makes the program control graph a strongly-connected graph. This path is not counted.) From the Cyclomatic Complexity, the maximum number of linearly independent paths in this graph is five.

One set of five linearly independent paths of this graph could be represented as follows:

B1 = (abef), (abeabef), (abebef), (acf), (adcf)

where B1 is a basis set of linearly independent paths. This set forms a basis for generating all possible paths through the graph. For example, the path (abeabebebef) can be represented as a linear combination of the paths (abebef) and (abef), as follows:

(abeabebebef) = 2(abebef) - (abef)

The basis set helps the developer to generate test cases for achieving path coverage, since all possible paths can be generated from it. The basis set ensures that no test cases will be excluded due to an insufficient basis. On the other hand, there may exist a basis set that is smaller.

Knowing the maximum size of the basis set does not guarantee that the test cases generated are the ones needed to achieve path coverage. Nevertheless, since there exists the number of linearly independent paths specified by the Cyclomatic Complexity, this measure sets a lower limit on the number of paths that must be tested to achieve path coverage. A test set that contains fewer paths cannot achieve test coverage, even if the paths are properly generated.

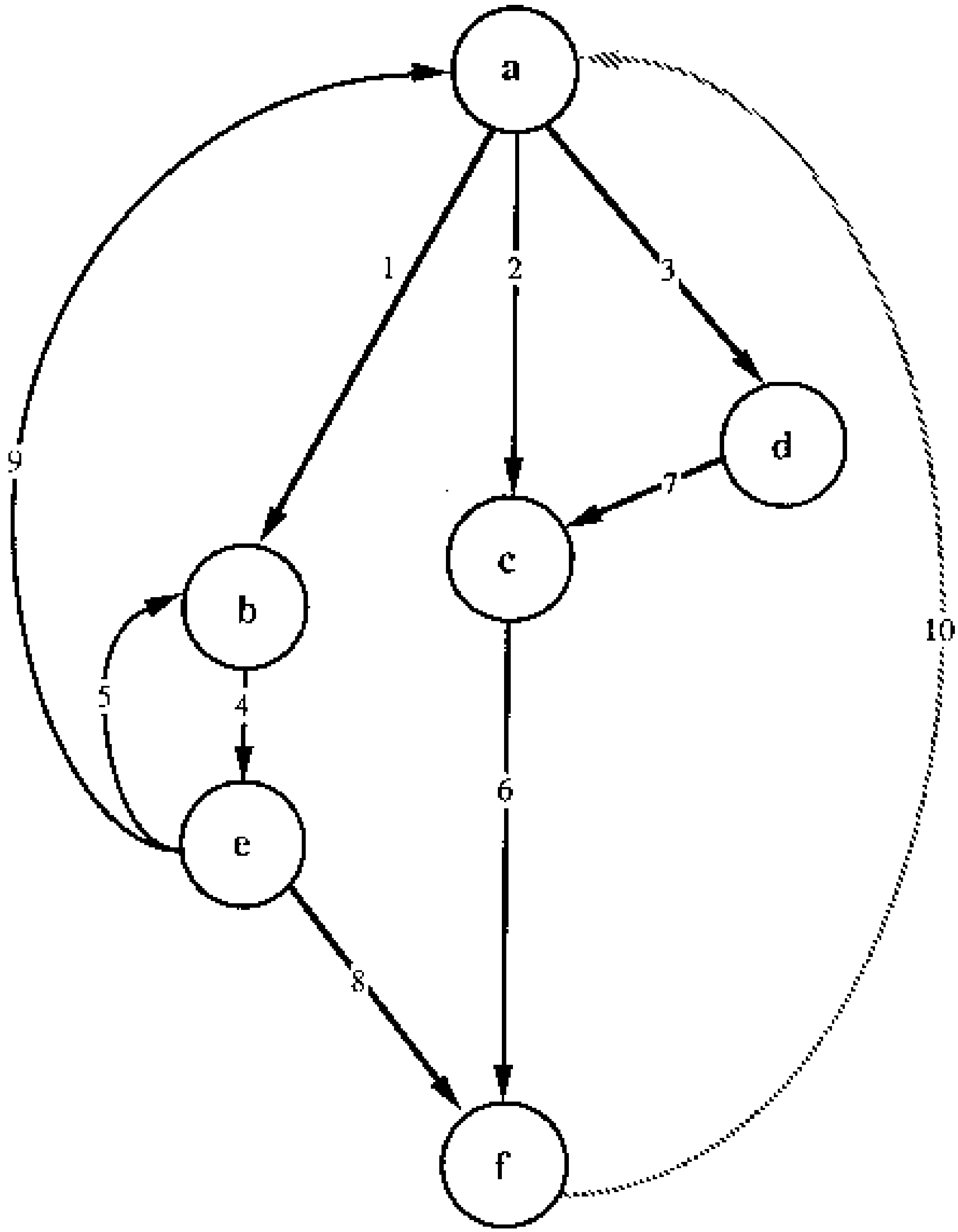


FIGURE 3.3-8. GRAPH WITH EDGE BACK

The Cyclomatic Complexity measure can be used on both structured and unstructured programs. Additionally, it can be automated, as evidenced by the tools that exist on the market.

3.3.1.2 Essential Complexity

The Essential Complexity measures the amount of unstructured code in a program. Modules containing unstructured code may be more difficult to understand and maintain. This measure allows developers to focus on modules of a program that are overly-complex because of their unstructured code. Unstructured programs contain one or more of the following:

- a branch out of a loop
- a branch into a loop
- a branch into a decision
- a branch out of a decision

Once the control graph of a module has been drawn, the graph can be reduced to obtain the Essential Complexity of that module.

The Essential Complexity is given by the Cyclomatic Complexity of the graph after all proper subgraphs have been removed. A proper subgraph consists of a single elementary control structure.

To reduce a graph, Mannino, Stoddard, and Sudduth (1990) recommend starting with the deepest nested structure in the program, and replacing each proper subgraph with a straight line. This can be done because all structured graphs can be reduced to a single function node. This is equivalent to moving the code statements into a subroutine. The Essential Complexity is obtained when the graph can no longer be reduced. Figure 3.3-9 illustrates how a graph is reduced. The function nodes produced are retained for clarity.

In figure 3.3-9, reading from left to right, the first graph has a v(G) of four; the second graph has a v(G) of three after code A is removed; the third graph has a v(G) of two after code B is removed; and the fourth graph has a v(G) of one, after code C is removed. By removing proper subgraphs, the developer can concentrate on unstructured areas in the program.

The Essential Complexity formula is as follows:

$$ev(G) = v(G) - m$$

where "ev(G)" is the Essential Complexity, " v(G)" is the Cyclomatic Complexity, and "m" is the number of proper subgraphs. For example, if a module has a Cyclomatic Complexity of 10, and three simple IF-THEN-ELSE structures could be removed, the Essential Complexity would be seven. By identifying the proper subgraphs, it is easier to focus on the unstructured code that produced the "ev" of seven.

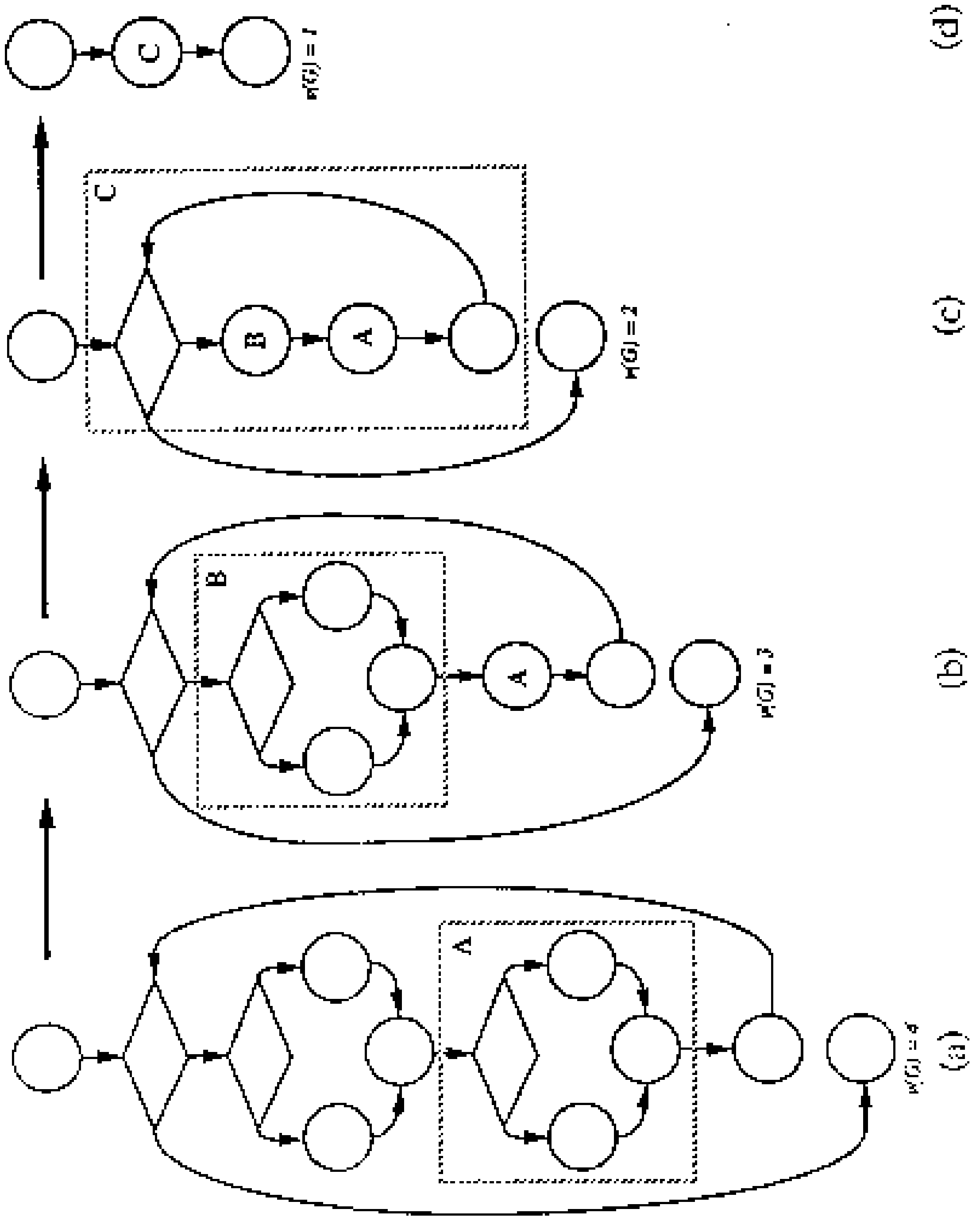


FIGURE 3.3-9. REDUCING A PROGRAM CONTROL GRAPH

In general, when the Essential Complexity equals the Cyclomatic Complexity, the graph does not contain any proper subgraphs. Hence, the Cyclomatic Complexity can only be reduced by recoding the program. One exception to this case is when structured code is already in the most succinct form: a single proper subgraph consisting of a function node. In this case, ev(G) = v(G).

3.3.2 Method 2 - Evaluating Program Statements

3.3.2.1 Cyclomatic Complexity

The Cyclomatic Complexity can also be computed by counting the branch conditions in a module:

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

where " π " is the number of simple branch conditions.

Consider the control graph of a simple program, shown in figure 3.3-10. The two branch conditions plus one gives a v(G) of three.

Or, the formula can be applied directly to code by counting (manually or with a software tool) the number of branch conditions in the program. Consider the Z80 Assembly code in figure 3.3-11.

Each jump (identified in bold) contains a simple condition:

$$\begin{aligned} v(G) &= \pi + 1 \\ &= 3 + 1 \\ &= 4 \end{aligned}$$

Counting the conditions was straightforward because the jumps each contain simple conditions. If the language contains a compound conditional, such as an "IF A AND B", the count would be two since the construct implies two simple conditions: data is tested against A and against B. More complex conditions should be analyzed similarly. The count depends on the number of simple conditions implied in the construct.

The formula " $\pi + 1$" is convenient because it allows developers to calculate the Cyclomatic Complexity of a program without having to use graph analysis. This Cyclomatic Complexity equation only applies to individual modules. The individual results must be summed to determine the Cyclomatic Complexity for an entire application. If a developer tries to sum all the conditions of all modules and then to use the equation, the calculation will underestimate the Cyclomatic Complexity.

For example, consider an application composed of 10 modules. For simplicity, assume each module has nine conditions. Then, each module has a Cyclomatic Complexity of 10: nine conditions plus one. Summing the Cyclomatic Complexity gives an overall application complexity of 100. This calculation is correct.

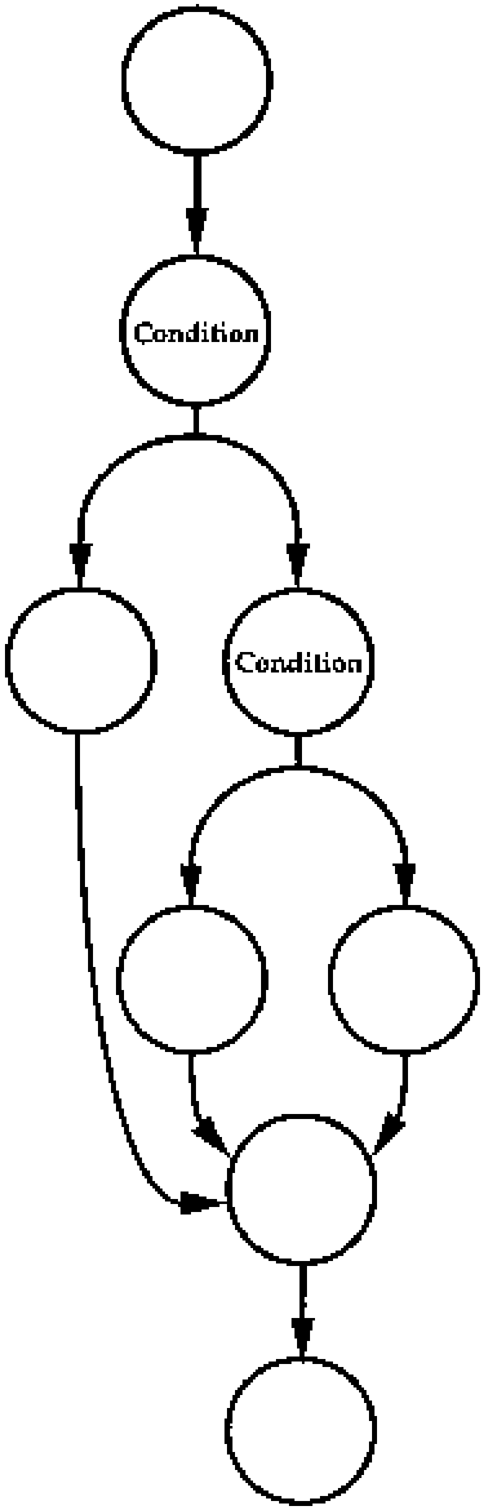


FIGURE 3.3-10. A CONTROL GRAPH OF A SIMPLE PROGRAM

```
;   *************************************
;   *                                   *
;   *    SUBROUTINE:  DITEST (FB)       *
;   *                                   *
;   *************************************
;
;   THIS SUBROUTINE WALKS A '1' THROUGH
;   EACH BIT POSITION OF EACH OUTPUT PORT
;   OF J2, AND READS THE CORRESPONDING INPUT
;   PORTS OF J5.  A 50 CONDUCTOR RIBBON
;   CONNECTS J2 TO J5.
;
;   PORT CORRESPONDENCE:
;
;     J2         J5
;     DO5 ------> DI0
;     DO6 ------> DI1
;     DO7 ------> D12
;     D08 ------> DI3
;     DO9 ------> DI4
;
DITEST:     LD      IX,DO0        ;GET OUTPUT BASE ADDRESS
            LD      A,01H         ;TEST PATTERN
;
T08A:       LD      (IX+5),A      ;OUTPUT TO REGISTERS
            LD      (IX+6),A
            LD      (IX+7),A
            LD      (IX+8),A
            LD      (IX+9),A
;     TEST INPUTS
            LD      HL,DI0        ;POINT TO 1ST DIGITAL INPUT
            LD      BC,05         ;BYTE COUNT
T08B:       CPI                   ;DO THE PATTERNS MATCH?
            JP      NZ,ENDTST     ;NO, GO TO FAILURE ROUTINE
            JP      PE,T08B       ;LOOP UNTIL DONE
            RLCA                  ;SET NEXT PATTERN
            CP      01H           ;FINISHED?
            JP      NZ,T08A       ;NO, CONTINUE
            LD      C,0FFH        ;FLAG DIAGNOSTIC IS SUCCESSFUL
ENDTST:     RET
```

FIGURE 3.3-11. Z80 ASSEMBLY CODE EXAMPLE

Instead of finding the Cyclomatic Complexity of each module, however, suppose the developer added the branch conditions of all the modules and then applied the formula. The number of conditions in each is nine. Nine conditions times 10 modules gives 90. Then, 90 + 1 = 91. This result is lower than the first result of 100. The wrong result occurs because a one is not added for every module.

The module must contain only single-entry and single-exit, structured, blocks of code for this formula to be precise. In single-entry, single-exit blocks of code, any called subroutines must pass the control flow back to the calling module. If the code branches out of or into a loop, or branches out of or into a decision, " $\pi + 1$" will not accurately reflect v(G) because the relationship of nodes to edges is upset. Hence, " $\pi + 1$" will no longer accurately represent the "e - n + 2p" formula from which it is derived.

Besides measuring complexity, the Cyclomatic Complexity Measure indicates the maximum number of test cases that must be generated to achieve branch coverage. For this type of test coverage, a basis set of paths constitutes a set of test cases that ensures branch coverage. However, "the number of paths in a basis can greatly exceed the minimum number of paths required to achieve branch coverage." (Prather 1983). Hence, this method could lead to over-testing. Furthermore, as previously mentioned, the basis set is not uniquely determined.

3.3.2.2 Modified Cyclomatic Complexity

The simplified Cyclomatic Complexity equation assumes that every program has a single exit. Some code, however, has more than one exit. For example, FORTRAN programs may contain "STOP" statements anywhere in the program. The following equation handles this situation for a program with " π " decisions and "s" exits (Harrison 1983):

$$v(G) = \pi - s + 2$$

For example, the Z80 code listed in figure 3.3-11 contained one exit, the "RET" statement. The code in figure 3.3-12 is similar to that code but contains two exits. The beginning sequential code is not shown since it does not affect the calculation.

```
RET       NZ          ;NO, GO TO FAILURE ROUTINE
JP        PE,T08B     ;LOOP UNTIL DONE
RLCA                  ;SET NEXT PATTERN
P         01H         ;FINISHED?
JP        NZ,T08A     ;NO, CONTINUE
LD        C,0FFH      ;FLAG DIAGNOSTIC IS SUCCESSFUL
RET
```

FIGURE 3.3-12. Z80 ASSEMBLY CODE CONTAINING TWO EXITS

In this case, the modified equation should be used to account for the extra exit:

$$\begin{aligned} v(G) &= \pi - s + 2 \\ &= 3 - 2 + 2 \\ &= 3 \end{aligned}$$

The simplified equation would still indicate that v(G) is four. The graph method, however, verifies that v(G) is three. Thus, if code contains multiple exits within modules, the developer should use the " π - s + 2" equation to determine v(G), otherwise the Cyclomatic Complexity will not accurately reflect a module's complexity (Harrison 1983). Like the " π + 1" equation, the modified formula only can be applied to individual modules.

3.3.3 General Analysis of McCabe's Cyclomatic Complexity Metric

In order to produce repeatable, well-behaved measures, McCabe's metric, like Halstead's metrics, requires that the developer define precise counting rules. These rules will differ depending on the language's constructs.

To use McCabe's metric properly, efforts must be made to ensure the following:

- The source code is error-free (e.g., it has compiled correctly).

- Unconditional branching does not contribute to the count.

- Every condition in a statement is considered.

- The formulas are applied properly. To use the "e - n + 2p" formula, treat each subgraph separately. Do no attempt to join the separate modules with the main module by adding edges in the control graph. Instead, obtain the edge and node counts for each module and sum them to get an overall count of the program's Cyclomatic Complexity. To use the " π + 1" formula, apply the formula to each module.

- All edges and nodes are properly counted for the "e - n + 2p" formula. Similarly, in the " π + 1" formula, ensure that all forms of conditions are anticipated by the counting rules and thus are counted.

McCabe's metric is flexible. The complexity of a module can be determined by counting the number of conditions in a program or by counting nodes and edges in a control graph. The former allows the process to be easily automated. Many commercial tools are available that tally a given language's constructs. When the "e - n + 2p" equation is used, the code is graphically represented. The graphs allow the programmer to see the structure of individual modules, and the graphs can be used to diagnose overly-complex structures.

Although McCabe defines three types of nodes (collecting, function, and predicate), he never clearly defines what statements each should contain. Hence, the implementor must judge what constitutes a node and must clearly define it. The measure will yield inconsistent results if implementors define nodes differently. For a particular application, however, results will not be inconsistent if the nodes have been defined precisely.

The " π + 1" equation has several potential implementation problems. The main problem with using this equation stems from the fact that it only applies to structured code. Most real applications contain a portion of unstructured code. So although industry fully embraces this form of the Cyclomatic Complexity Measure because it can be calculated solely from code, its results may be questionable. Further, assuming the code is very structured, the measure may still be implemented incorrectly on multiple modules, as noted earlier.

The formula "e - n + 2p" will yield the most consistent, repeatable results. Because this formula is additive, it can be applied to modules or the entire application. Although graph analysis is required, the analysis can be automated. Again, nodes must be defined precisely. The CE may request that the developer submit definitions of the nodes to ensure that the measure has been properly interpreted.

McCabe's Cyclomatic Complexity Metric claims to measure the complexity of a module by counting the number of decisions in a program. The metric measures the complexity within a module. It does not assess the complexity that might exist in the connections between modules. Further, McCabe's metric measures only the control structures within a module. Although it may prove to indicate a module's complexity in general, because of the metric's narrow focus, there may be exceptions.

For example, Module A contains a simple sequence control structure in which v(G) is 1. Module B also has a v(G) of 1. When the code is inspected, Module A is found to contain 1000 lines of sequential source code; Module B is found to contain one line. According to the Cyclomatic Complexity metric, these two modules have the same complexity. Obviously, the one containing 1000 lines is more complex, even at the most cursory level of reading the module. This situation is atypical, however, since a module containing 1000 lines of code would more likely contain complex control structures as well. In general, studies show a high correlation between McCabe's Complexity Metric and a module's complexity and error-proneness (Myers 1977).

Since 1982, McCabe's Cyclomatic Complexity Metric has received widespread support, both in industry and the government. McCabe's measure provides an upper threshold of complexity for modules. Industry and government have supported his notion that a module containing a v(G) greater than 10 is too complex. Project leaders have used this number to determine the complexity of modules, and to identify complex code in applications.

3.4 RADC's Software Metrics

In the early 1970s, Boehm et al. developed an SQM methodology that could be applied throughout the software life-cycle. Their research efforts were published as Characteristics of Software Quality in 1978. Other researchers, notably RADC, expanded this research. The metrics discussed in this section are based on RADC's continuing development effort, as given by Bowen, Wigle, and Tsai (1985).

The RADC methodology consists of metric elements, metrics, criteria, and factors. The metric element scores are combined to form a metric score and metric scores are combined to form a criterion score. Criterion scores are then combined in various ways to produce several quality factor scores which are correlated with the quality factors presented in chapter 4.

Metric elements are the quantitative measures of software objects. Metrics are the software-oriented details of software characteristics. Criteria are the software-oriented characteristics that contribute to various types of quality. Factors are the user's idea of an aspect of product quality (Bowen, Wigle, and Tsai 1985). Figure 3.4-1 shows a model of part of this hierarchy.

RADC's methodology is based on a set of 327 software metric elements. Each metric element applies to particular objects of a particular software product in the life-cycle. In order to get a cumulative measure of the software, RADC consolidates these metric elements into a single metric. Thus, in the RADC methodology, a software metric is one step removed from the software objects, but it is the lowest level of software measure that can be applied to various software products. The metric elements are combined into 73 software metrics.

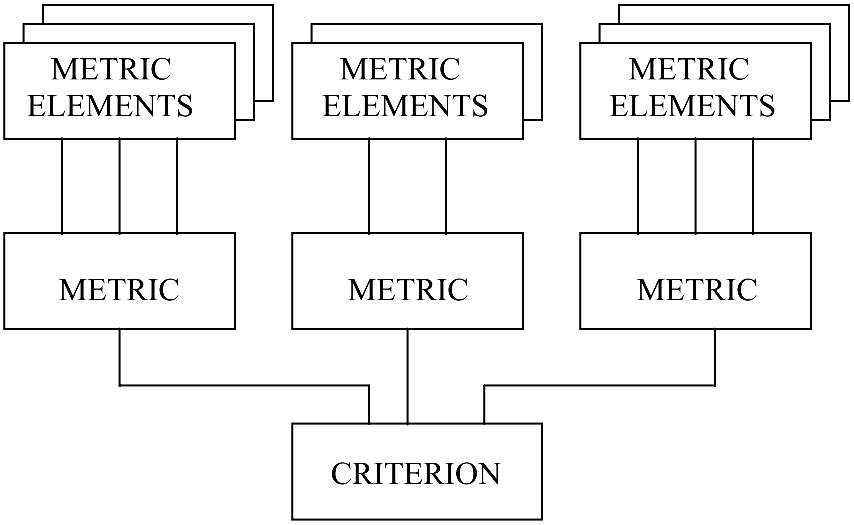


FIGURE 3.4-1. COMPONENTS OF THE RADC METHODOLOGY

Metrics and metric elements are categorized according to the specific criterion which they compose; e.g., the SI (Simplicity) group contains simplicity-related measures, the MO group measures modularity attributes. There are 29 such criteria.

The metric elements are listed on worksheets in the RADC Quality Evaluation Guidebook (Bowen, Wigle, and Tsai 1985). Each worksheet addresses one phase of the software life-cycle. For this report, the metric elements were taken from Metric Worksheets 4A and 4B, which discuss code metrics.

Specifically, Metric Worksheet 4A addresses each metric element at the Computer Software Configuration Item (CSCI) level of the code. Metric Worksheet 4B addresses the unit level of code. A CSCI is a set of units which address a particular function. A unit of code is typically a single module, like a subroutine.

The worksheets consist of questions concerning the appropriate level of code. First the unit questions must be answered, next the CSCI questions. The questions are of two types: those that are binary decisions, and those whose answers are numeric.

The questions must be posed consistently so that when scores are combined, the result is meaningful. Thus, a positive contribution to software quality is represented by a yes or a greater numeric value. Furthermore, these answers are normalized to fall between zero and one. A "yes" answer, or the best numeric value, results in a score of one. A "no" answer, or the worst numeric value, results in a score of zero.

In this report, the SI group of metrics is discussed first, in some detail. Simplicity is based on many metrics, which is typical of the 29 criteria. This criterion is based on six metrics, SI.1 through SI.6. Three of these metrics are composed of one metric element. The other three are composed of three to 14 metric elements each. A sample of each of these types of metrics is discussed. This case is presented in depth to show the qualifying assumptions underlying each metric, to show how each is calculated, and to highlight implementation problems. After this discussion, a representative sample of the remaining metric elements of other groups is presented.

In each of the following metric discussions, the criterion to which the metric contributes is given parenthetically in the metric heading.

3.4.1 SI.3 (Simplicity) Data and Control Flow Complexity Metric

This metric quantifies the data and control flow complexity of the units of a CSCI. The lack of such complexity is used to indicate one part of the CSCI's simplicity. The metric is based on the single metric element, SI.3(1). In spite of the alphabetical ordering of the Worksheet, the metric element is calculated by first answering the unit questions, then the CSCI questions.

SI.3(1) CSCI questions (Worksheet 4A):

a. "How many applicable units (answer of Y or N on 4B)?" (For how many units of this CSCI was item f calculated?) value, N/A

b. "What is total score for all applicable units (add applicable unit scores from 4B)?" (What is the sum of all the item f values?) value, N/A

c. "Calculate b/a and enter score." value, N/A

Unit questions (Worksheet 4B):

d. "How many conditional branch statements are there (e.g., IF, WHILE, REPEAT, DO/FOR LOOP, CASE)?" value, N/A

e. "How many unconditional branch statements are there (e.g., GO TO, CALL, RETURN)?" value, N/A

f. "Calculate 1/(1+d+e) and enter score." value, N/A

From the object measures performed for the unit questions, both types of branching are seen to contribute negatively to simplicity. The numerator in item f is fixed; as the denominator increases, simplicity decreases. The simplest unit would contain no branch statements of either kind, thus the score would be one. This is already a normalized score. The CSCI questions perform a simple direct ratio averaging of the unit scores of the CSCI.

In general, the program simplicity will be a monotonic inverse function of the number of branch statements. In isolated cases, however, adding a branch can increase the simplicity.

This metric element shows several implementation problems. What is meant by the number of applicable units in question a? The parenthetical reference to Worksheet 4B in the quote implies that if the unit question could be answered with a "Y" or "N", as opposed to a "N/A", the unit is an applicable one. But which of the unit questions should be used in this determination? Furthermore, in this particular case, the unit questions are not binary, but are values. The reference is not clear. The reference probably should have specified that an applicable unit is one for which the answer for item f was not "N/A". The authors clarified the reference in the parenthetical note after question a. Many of the metric element questions have this ambiguity. In the remaining metric element discussions, a probable, meaningful, and workable reference is provided parenthetically after each question a.

A similar ambiguity exists in question b. Which score is being referenced? It is not clear which unit question is being referenced. The reference probably should have specified item f. Again, the authors clarified the reference in the parenthetical note after question b. Many of the metric element questions have this ambiguity. In the remaining discussions, a probable, meaningful, and workable reference is provided parenthetically after each question b.

But are these references correctly interpreted? Unless similar clarifications are provided, it is doubtful that two implementors will interpret the metric element questions the same way. Clarified worksheets must be provided to ensure that this metric element is repeatable. Some variability might also occur in interpreting the number of branching statements, particularly in a loop or an "IF-THEN-ELSE" branching structure. To ensure repeatability, the conditional and unconditional branch statements should be specified for every branching construct.

3.4.2 SI.4 (Simplicity) Coding Simplicity Metric

This metric quantifies the simplicity of the coding in the units of a CSCI. This software metric consists of 14 metric elements. Because of the large number of metric elements associated with the coding simplicity measure, only the first six are discussed. To calculate the first metric element, answer the unit question, then the CSCI questions, as follows:

SI.4(1) CSCI questions (Worksheet 4A):

a. "How many applicable units (answer of Y or N on 4B)?" value, N/A

b. "How many units with answer of Y (see 4B)?" value, N/A

c. "Calculate b/a and enter score." value, N/A

Unit question (Worksheet 4B):

d. "Is the flow of control from top to bottom (i.e., flow of control does not jump erratically)?" Y, N, N/A

An analysis of the object measure performed on the unit question implies that upward flow of control contributes negatively to simplicity. In this metric element, the unit question only produces a list of yes or no answers; the question is binary. It does not produce a measure. The CSCI questions convert the answers to a normalized numeric answer. The measure is simply the ratio of the number of units with downward flow to the total number of units containing executable statements.

In general, the program simplicity will be a monotonic inverse function of the amount of upward flow. In isolated cases, however, upward flow can increase the simplicity. For example, in figure 3.4-2, the "GOTO 10" statement produces upward flow. The set of statements on the left has one upward flow, the set on the right has two.

```
C      BEGIN LONG CALCULATION
   10  READ DATA
C      LOOK FOR GOOD DATA
       IF DATA = 0 GOTO 1050
          .
          .
          .
   1050  GOTO 10
```

```
C      BEGIN LONG CALCULATION
   10  READ DATA
C      LOOK FOR GOOD DATA
       IF DATA = 0 GOTO 10
          .
          .
          .
   1050  GOTO 10
```

FIGURE 3.4-2. EXAMPLES OF PROGRAM FLOW

In the former set, the code obeys good practice and passes control to the end of the calculation loop. The existing upward flow is used twice; the number of unique upward flows is minimized. However, the code is not simple since flow is transferred several pages ahead in the program, only to be set right back to the same area again. The additional upward flow in the "IF" statement on the right improves the simplicity of the code.

Because this metric element has only one unit question, the references in questions a and b are unambiguous. Furthermore, question d truly is answered with a "Y" or "N". Nevertheless, this metric element is still ambiguous because the implementor must decide whether the flow is erratic. A loop contains upward flow, but this would probably not be judged as erratic flow. The decision is subjective. Hence, although this measure is easily implemented, it may not be repeatable. The question could be stated clearly, but this might be overly restrictive. For example, "Does the control flow always flow only from the top to the bottom?" would be answered consistently, but loops would then detract from simplicity.

The second metric element is calculated from the following questions:

SI.4(2) CSCI questions (Worksheet 4A):

a. "How many applicable units (answer of Y or N on 4B)?" value, N/A
(For how many units of this CSCI was item f calculated?)

b. "What is total score for all applicable units (add applicable unit scores from 4B)?" value, N/A
(What is the sum of all the item f values?)

c. "Calculate b/a and enter score." value, N/A

Unit questions (Worksheet 4B):

d. "How many lines of estimated source code, excluding comments?"	value, N/A
e. "How many negative boolean and compound boolean expressions are used?"	value, N/A
f. "Calculate 1-(e/d) and enter score."	value, N/A

The equation in item f implies that an increase in the number of negative and compound boolean expressions (question e) detracts from simplicity. The worst case of zero for item f occurs when there is one such expression for every line of executable code. If there are no such expressions, the value for item f is one. This equation will usually produce a normalized result, but a program that has more negative and compound boolean expressions than lines of executable code would produce a negative measure. However, it is unlikely that this situation would occur. If it does occur, some adjustment would be necessary. The e/d factor would never be indeterminate due to a question d value of zero, since, when there are no lines of executable code, the unit questions do not apply. The CSCI questions simply perform a direct ratio averaging of the unit scores of the CSCI.

Questions a and b of this measure have the same implementation problems as those for SI.3(1). Question a refers to a binary decision, even though the unit questions are all values. Question b refers to the count in item f.

When a program contains a compound condition, a simple Boolean expression usually improves the simplicity. Consider the following FORTRAN conditions (where previous program constructs require that A2 is greater than A1):

```
20  IF (A1 .LE. THERM(I)) GO TO 50
    IF (THERM(I) .LT. A2) COUNT=COUNT+1
50  CONTINUE
```

The "COUNT" is only incremented if the value of the array element, "THERM(I)", lies between the values of A1 and A2. Combining these two statements into one using the Boolean operator, ".AND.", gives the following:

```
20  IF ((THERM(I) .GT. A1) .AND. (THERM(I) .LT. A2)) COUNT = COUNT+1
```

This single Boolean expression makes it easier to see the logic of the condition.

However, Boolean expressions should be used judiciously. Negative and compound Boolean expressions tend to create overly-complicated structures, thus reducing simplicity. SI.4(2) addresses these cases.

In compound Boolean expressions the logic can become more difficult to understand. Consider the following FORTRAN statement:

```
IF (VELOC .EQ. NVELOC) .AND. (ACCEL .GT. MAXAC) .OR. (SHEAR .GT. 20) GOTO 10
```

The two Boolean operators, ".OR." and ".AND.", multiply the number of conditions in the expression. This code is confusing. In this case, it would be better to break the expression into a few "IF" statements that each contain simple Boolean expressions.

Consider the following code that contains a negative Boolean expression:

```
IF (.NOT. BLANK) THEN C=D ELSE E=F
```

The negative can be removed, making the condition more direct:

```
IF (BLANK) THEN E=F ELSE C=D
```

The revised code is more straightforward. The "ELSE" condition occurs if the condition is not true.

This metric element is not necessarily repeatable because it allows the practitioner to use an estimate of the number of lines of code. To make it repeatable, the estimate should always be based on a well-defined measure.

This measure is weakened even more because the measurement of question e might not be strictly monotonic. Depending on the logic, negative boolean expressions may be simpler or compound boolean expressions may be simpler. In either case, the metric would not indicate simplicity. These cases, however, are exceptions.

The third metric element is calculated from the following questions:

SI.4(3) CSCI questions (Worksheet 4A):

a. "How many applicable units (answer of Y or N on 4B?)" value, N/A
(For how many units of this CSCI was item f calculated?)

b. "What is total score for all applicable units (add applicable unit scores from 4B)?" value, N/A
(What is the sum of all the item f values?)

c. "Calculate b/a and enter score." value, N/A

Unit questions (Worksheet 4B):

d. "How many loops (e.g., WHILE, DO/FOR, REPEAT)?" value, N/A

e. "How many loops with unnatural exits (e.g., jumps out of loop, return statement)?" value, N/A

f. "Calculate 1-(e/d) and enter score." value, N/A

Analysis of the equation in item f shows that an increase in the number of loops that are exited unnaturally (question e) detracts from simplicity. The worst case of zero for item f occurs when every loop contains an unnatural exit. If there are no such loops, the value for item f is one. This equation produces a normalized result. The CSCI questions simply perform a direct ratio averaging of the unit scores of the CSCI. The e/d factor would never be indeterminate due to a question d value of zero, since, when there are no loops in a unit, the unit questions do not apply.

This metric element determines an aspect of simplicity because it determines how many unstructured loops exist in a program. Unstructured loops decrease the simplicity of a program.

Questions a and b of this measure have the same implementation problems as those for SI.3(1). Question a refers to a binary question, even though the unit questions are all values. Question b refers to the count of item f. If the classifications above are followed, this metric element is repeatable. Furthermore, it is a monotonic measure of simplicity.

The fourth metric element is calculated from the following questions:

SI.4(4) CSCI questions (Worksheet 4A):

a. "How many applicable units (answer of Y or N on 4B)?" value, N/A
(For how many units of this CSCI was item f
calculated?)

b. "What is total score for all applicable units (add value, N/A
applicable unit scores from 4B)?"
(What is the sum of all the item f values?)

c. "Calculate b/a and enter score." value, N/A

Unit questions (Worksheet 4B):

d. "How many iteration loops (e.g., DO/FOR loops)?" value, N/A

e. "In how many iteration loops are indices modified value,
N/A
to alter fundamental processing of the loop?"

f. "Calculate 1-(e/d) and enter score." value, N/A

An iteration loop is one whose exit is based on a count of the number of times the loop executes, rather than on some other condition. The index modification referred to is a modification in the body of the loop instead of by the loop control statements. An analysis of the equation of item f shows that an increase in the number of loops whose indices are so modified (question e) detracts from simplicity.

The worst case of zero for item f occurs when every loop that is based on a count has its index modified. If there are no such loops, the value for item f is one. This equation produces a normalized result. The CSCI questions simply perform a direct ratio averaging of the unit scores of the CSCI. The e/d factor would never be indeterminate due to a question d value of zero, since, when there are no iteration loops in a unit, the unit questions do not apply.

This metric element requires more specific directions to ensure that it is repeatable and monotonic. Although it is easy to detect when an index is modified in the body of a loop, it is not so clear whether a loop is an iteration loop. For instance, "DO/FOR" loops (or "IF-GOTO" loops) that do not contain unnatural exits are always iteration loops. If one exits because of an "IF" statement which checks any condition other than the index size, it is no longer an iteration loop. When such irregularities are accounted for, this metric element is repeatable.

This metric is generally monotonic, but it makes no attempt to quantify the complexity of the index modifications. The simplicity of code containing one infraction in each loop will be monotonically represented by this measure. Loops with several modifications will not.

The fifth metric element is calculated from the following questions:

SI.4(5) CSCI questions (Worksheet 4A):

a. "How many applicable units (answer of Y or N on 4B)?" value, N/A

b. "How many units with answer of Y (see 4B)?" value, N/A

c. "Calculate b/a and enter score." value, N/A

Unit question (Worksheet 4B):

d. "Is the unit free from all self-modification of code (i.e., does not alter instructions, overlays of code, etc.)?" Y, N, N/A

A normalized score is produced by taking the ratio of the number of units free from self-modification to the total number of units of executable code. In this case, simplicity is improved by having straight-forward code that does not modify itself.

To insure the measure is repeatable, code and data must be carefully distinguished. For instance, in Assembly language, data can be embedded in an instruction. The rules to detect this infraction must specify whether modifying such immediate data is acceptable.

The sixth metric element is calculated from the following questions:

SI.4(6) CSCI questions (Worksheet 4A):

a. "How many applicable units (answer of Y or N on 4B)?" value, N/A
(For how many units of this CSCI was item f
calculated?)

b. "What is total score for all applicable units (add value,
N/A
applicable unit scores from 4B)?"
(What is the sum of all the item f values?)

c. "Calculate b/a and enter score." value, N/A

Unit questions (Worksheet 4B):

d. "How many lines of source code, excluding comments?" value, N/A

e. "How many statement labels, excluding labels for value, N/A
format statements?"

f. "Calculate 1-(e/d) and enter score." value, N/A

Analysis of the equation in item f shows that an increase in the number of labels to which control is transferred (question e) detracts from simplicity. The worst case of zero for item f occurs when every statement has such a label. If there are no such labels, the value for item f is one. This equation produces a normalized result. Using the CSCI questions, the practitioner can perform a direct ratio averaging of the unit scores of the CSCI. The e/d factor would never be indeterminate due to a question d value of zero, since, when there are no executable statements in a unit, the unit questions do not apply.

This measure is based on the assumption that increasing the number of labels to which control may pass decreases the simplicity. However, in structured languages a program can have loops and branching without labels. For example, in FORTRAN, a simple "IF-THEN-ELSE" condition contains branches without labels.
The simplicity of reading a program is generally a monotonic function of the number of labels, if all other variables remain constant.

The statements that are considered source lines of code must be specified. Question e is ambiguous on this same point. In order to produce a normalized measure, the statements of question e should be those counted for question d.

3.4.3 AP.2 (Application Independence) Data Structure Metric

The AP.2 metric quantifies the extent to which code is independent of the application in which it is used. In particular, it measures the contribution that the data structure makes to application independence. It is based on four metric elements. Only the first one is covered here.

AP.2(1) CSCI questions (Worksheet 4A):

a . "How many applicable units (score entered on 4B)?" (For how many units of this CSCI was item f calculated?) value, N/A

b. "What is total score for all applicable units (add applicable unit scores from 4B)?" (What is the sum of all the item f values?) value, N/A

c. "Calculate b/a and enter score." value, N/A

Unit questions (Worksheet 4B):

d. "How many parameters in the argument list for the unit?" value, N/A

e. "How many global variables are referenced by the unit?" value, N/A

f. "Calculate d/(d + e) and enter score." value, N/A

An analysis of the equation in item f shows that the ratio of parameters passed to a unit as arguments rather than as global variables indicates an increase in the application's independence. The worst case of zero for item f is approached when the number of global variables is much greater than the number of arguments. If all the parameters are passed as arguments, the value for item f is one.

This equation produces a normalized result. If the answer to question d is zero, it must be reported as "N/A", since there is no longer a reference with which to compare the number of global variables used. The practitioner can use the CSCI questions to perform a direct ratio averaging of the unit scores of each CSCI.

This measure focuses on the extent to which code could be used in other applications. If a unit uses global variables, it cannot be used easily in other applications, which would not likely include the necessary variables in their global data structures. AP.2(1) is not application independent. The ratio calculated by this element indicates the extent of the problem.

This metric element could also indicate the run-time performance of some code. If a unit of code uses several global variables, it will tend to run more quickly than one that relies on local variables, especially where the unit is called frequently. Consider a unit that is called 1,000 times in a large application. If it only contains local variables, each time the unit executes, the variables must be passed to it. On the other hand, if the unit of code uses global variables, these values of the global variables are immediately available. The unit that uses global variables will run more quickly.

3.4.4 AP.3 (Application Independence) Architecture Standardization Metric

This metric quantifies the contribution that the computer architecture makes to application independence. It is based on two metric elements. Only the second is covered here.

AP.3(2) CSCI questions (Worksheet 4A):

a. "How many applicable units (score entered on 4B)?" value, N/A
(For how many units of this CSCI was item f calculated?)

b. "What is total score for all applicable units (add applicable unit scores from 4B)?" value, N/A
(What is the sum of all the item f values?)

c. "Calculate b/a and enter score." value, N/A

Unit questions (Worksheet 4B):

d. "How many lines of source code, excluding comments?" value, N/A

e. "How many non-HOL lines of code, excluding comments (e.g., Assembly language)?" value, N/A

f. "Calculate e/d and enter score." value, N/A

The equation of item f suggests that an increase in the proportion of code that is Assembly language (question e) increases the application independence of the code. This is a mistake. Like the other metric elements discussed, item f should represent the positive contribution to the criterion. Apparently, question e is supposed to ask the number of lines of high-order language (HOL) code. (Or, the item f equation could be 1-(e/d).)

Accepting the second interpretation implies that the worst case of zero for item f occurs when the source code is entirely Assembly language (i.e., there are zero HOL lines of code). If the unit contains no Assembly language, the value for item f is one. This equation produces a normalized result. The e/d factor would never be indeterminate due to a question d value of zero, since, when there are no executable statements in a unit, the unit questions do not apply.

The CSCI questions total the unit scores of the CSCI (an unnormalized value) and then calculate an average by dividing this total by the number of units (which re-normalizes it).

Question e refers to Assembly or any low level language. Because Assembly code is highly machine-dependent, the unit code is more application dependent if Assembly lines are included. Other applications do not necessarily use the same machinery. Thus, this element measures the application dependence that results when code is designed to run on a particular machine. In order to use a unit that contains Assembly language in another application that runs on a different machine, every line of non-HOL code would have to be re-coded. This measure is monotonic for this type of application independence. For this measure to also be repeatable, the software tool or code reader must be able to recognize Assembly code consistently.

3.4.5 CP.1 (Completeness) Completeness Checklist Metric

Completeness is based on the completeness checklist. This checklist is a single metric, consisting of 11 metric elements. Only two of them are discussed here.

CP.1(2) CSCI questions (Worksheet 4A):

a. "How many applicable units (score entered on 4B)?" value, N/A
(For how many units of this CSCI was item f calculated?)

b. "What is total score for all applicable units (add applicable unit scores from 4B)?" value, N/A
(What is the sum of all the item f values?)

c. "Calculate b/a and enter score." value, N/A

Unit questions (Worksheet 4B):

d. "How many data references are identified?" value, N/A

e. "How many identified data references are documented with regard to source, meaning, and format?" value, N/A

f. "Calculate e/d and enter score." value, N/A

Analysis of the equation in item f shows that an increase in the number of documented data references (question e) improves completeness. The worst case of zero for item f occurs when none of the identified data references are documented. If they all are, the value for item f is one. This equation produces a normalized result. The e/d factor would never be indeterminate due to a question d value of zero, since, when there are no data references identified in a unit, the unit questions do not apply. The CSCI questions total the unit scores of the CSCI (an unnormalized value). An average is then calculated by dividing this total by the number of units (which re-normalizes it).

Question d is ambiguous when compared with the other questions. Instead of asking, "how many data references?", the RADC researchers completed the sentence. "How many data references are identified?" should be interpreted as the simple question of how many exist. Then, from question e, code is considered incomplete if a reference to some item of data is not documented in comments with regard to the source of the data, the content of the data element, and its format.

This metric element is a one-to-one, monotonic measure. Every data reference must be documented to be a complete reference. In addition, the comment must document the source of the data, and its content and format. If any one item of information is missing, it is an incomplete data reference.

CP.1(10) CSCI questions (Worksheet 4A):

a. "How many applicable units (answer of Y or N on 4B)?" value, N/A

b. "How many units with answer of Y (see 4B)?" value, N/A

c. "Calculate b/a and enter score." value, N/A

Unit question (Worksheet 4B):

d. "Are all parameters in the argument list used?" value, N/A

Question d implies that using all the parameters in the argument list of a unit improves completeness. The unit question does not yet supply a measure. The CSCI questions convert the answers to a normalized numeric answer. It is the ratio of the number of units that use all their parameters to the number of units which have argument lists. The worst case of zero for question c occurs when none of the units completely use their parameters. If they all use every one, the value for question c is one. This equation produces a normalized result. The b/a factor would never be indeterminate due to a question a value of zero, since, when there are no units with argument lists, the metric element questions do not apply.

In each unit, all parameters in the argument list should be used. Consider the following FORTRAN code:

```
SUBROUTINE GRAD(VECT,TENS,COORD,PLANE)
VECT = TENS * COORD
```

The parameter "PLANE" was not used by the unit. Since subroutine "GRAD" accepts this additional parameter, this unit would seem to be incomplete. Apparently, the code that would use the unused parameter had not yet been put into the subroutine.

Although this measure is highly repeatable, it is not clearly monotonic. Many languages purposely allow fewer parameters to be passed than are defined for a subroutine. In this case, a decrease in the measure does not represent a decrease in the completeness. Alternatively, the reason "PLANE" was not used may be because the parameter is no longer needed, but it has not been removed from the code. Again, a decrease in the measure does not represent a decrease in the completeness.

In compiled languages, a robust compiler might pick up the argument discrepancies. A simple compiler might not. In the case where the compiler does not pick up the inconsistency, this metric element is meaningful. Otherwise it would always give a perfect score.

3.4.6 EP.1 (Effectiveness-Processing) Processing Effectiveness Metric

This metric quantifies the efficiency of processing performed by the units of the CSCI. It distinguishes inefficient processes from inefficient data handling, either of which affect the overall effectiveness of the processing. This metric is based on six metric elements, only one of which is discussed here.

EP.1(3) CSCI questions (Worksheet 4A):

a. "How many applicable units (score entered on 4B)?" value, N/A
(For how many units of this CSCI was item f calculated?)

b. "What is total score for all applicable units (add applicable unit scores from 4B)?" value, N/A
(What is the sum of all the item f values?)

c. "Calculate b/a and enter score." value, N/A

Unit questions (Worksheet 4B):

d. "How many units are required to be optimized for processing efficiency?" value, N/A

e. "How many units are optimized for processing efficiency (i.e., compiled using an optimizing compiler or coded in Assembly language)?" value, N/A

f. "Calculate 1-(e/d) and enter score." value, N/A

These questions have problems. The equation of item f indicates that an increase in the number of optimized units (question e) detracts from the processing efficiency. This is a mistake. This question structure compares to that of CP.1(2), where question e is worded as a positive contribution to the measure. Thus, item f should probably use the same e/d formula. The 1-e/d formula is used when question e tallies the negative contributions, as in SI.4(6).

If this change is accepted, item f would create a normalized score, but the CSCI questions do not maintain normalization. Question c could never be greater than one divided by the number of applicable units. Since questions d and e address bulk properties of each unit, this question probably should follow the pattern of SI.4(5), which has only one unit question.

Because of these problems, this measure cannot be determined, thus it is not monotonic or repeatable. Nevertheless, the intent can be discussed.

This measure should indicate the processing efficiency of code. If the units are optimized, the run-time efficiency increases. It will take the computer less time to execute the program than if it had not been optimized. Efficient processing may be a significant consideration in real-time or on-line transaction processing.

Effectiveness, however, does oppose other desirable software qualities. An optimized program may be very efficient, but may only run under a particular software environment and on a specific computer configuration. For example, a FORTRAN compiler that optimizes FORTRAN code for parallel processing will rearrange the source code to create more optimal blocks of code. Additionally, it may insert compiler directives unique to the particular operating system and run-time environment. The resulting code will probably not run on another system. Hence, run-time efficiency is created at the expense of portability. Additionally, the transcribed source may no longer be easily maintained or very readable. This, and similar conflicts, are addressed when the Effectiveness-Processing criterion is used to produce various factor scores. At this level, the metric only claims to address processing effectiveness.

3.4.7 EP.2 (Effectiveness-Processing) Data Usage Effectiveness Metric

This metric quantifies the data usage effectiveness part of processing effectiveness. It is composed of seven metric elements. Only one is discussed here.

EP.2(5) CSCI questions (Worksheet 4A):

a. "How many applicable units (score entered on 4B)?" value, N/A
(For how many units of this CSCI was item f calculated?)

b. "What is total score for all applicable units (add value, N/A
applicable unit scores from 4B)?"
(What is the sum of all the item f values?)

c. "Calculate b/a and enter score." value, N/A

Unit questions (Worksheet 4B):

d. "How many arithmetic expressions?" value, N/A

e. "How many arithmetic expressions with mixed value, N/A
data types in the same expression (e.g., integer/real/boolean/literal)?"

f. "Calculate 1-(e/d) and enter score." value, N/A

Analysis of the equation in item f shows that an increase in the number of expressions containing mixed data types (question e) detracts from data usage effectiveness. The worst case of zero for item f occurs when every arithmetic expression contains mixed data types. If there are no mixed expressions, the value for item f is one. This equation produces a normalized result. The CSCI questions lead to a direct ratio averaging of the unit scores of the CSCI. The e/d factor would never be indeterminate due to a question d value of zero, since, when there are no arithmetic expressions in a unit, the unit questions do not apply.

Some examples of data types include integer, real, complex, boolean, literal, and double-precision. Arithmetic expressions containing mixed data types can cause unnecessary processing since a compiler has to introduce additional steps into the object code to convert the data. If the data are converted and matched by the programmer in source code, the object code will be more compact. Thus, the program will process data more effectively.

This metric element is not strictly monotonic or repeatable. Since it makes no attempt to quantify the amount of mixture, the effect on the processing effectiveness could vary greatly for a given score. Repeatability may suffer due to different judgments about mixed types. A more specific description should be provided. For example, although the following FORTRAN arithmetic expression contains four integer variables, it must perform a floating point calculation and then truncate the result to an integer:

```
INTEGER ANSWER,A,B,C
ANSWER = A + B/C
```

One person may judge that the expression is mixed, because "B/C" is a real number. Another may say that it is unmixed because all the variables are integers.

3.4.8 GE.2 (Generality) Unit Implementation Metric

The GE.2 metric elements address machine-dependent operations and overly-restrictive limits on the number and values of data. When the elements of the Unit Implementation metric are combined, they measure the extent to which the code within each unit can be applied to a different, broader context. Another GE metric quantifies how generally each unit is used by the other units.

GE.2(2) CSCI questions (Worksheet 4A):

a. "How many applicable units (answer of Y or N on 4B)?"	value, N/A
b. "How many units with answer of Y (see 4B)?"	value, N/A
c. "Calculate b/a and enter score."	value, N/A

Unit question (Worksheet 4B):

d. "Is this unit free from machine-dependent operations?"	Y, N, N/A

Question d implies that units that contain machine-dependent operations are not general. The measure is the ratio of the number of units that contain machine-dependent operations to the total number of units. The worst case of zero for question c occurs when all of the units contain machine-dependent operations. If all are free of such operations, the value for question c is one. This equation produces a normalized result. The b/a factor would never be indeterminate due to a question a value of zero, since, when there are no units with executable code, the metric element questions do not apply.

Each unit in an application should be analyzed to determine whether it contains machine-dependent operations. For example, a call to a real-time clock is a machine-dependent function. The more units containing machine-dependent functions, the less likely it is that the unit can be used to solve the same problem in a broader context.

This metric element is repeatable, but not strictly monotonic. Since it makes no attempt to quantify the number of such operations, the affect on the generality could vary greatly for a given score.

The following two metric elements use the same CSCI questions as did GE.2(2).
GE.2(3) Unit question (Worksheet 4B):

d. "Is this unit free from strict limitations on Y, N,
N/A
the volume of data items it processes (e.g., data volume
limits are parameterized)?"

GE.2(4) Unit question (Worksheet 4B):

d. "Is this unit free from strict limitations on the values Y, N, N/A
of input data (e.g., no error tolerances are specified;
no range tests or reasonableness checks are performed)?"

These questions imply that units limiting the volume of data processed or the value of data entered are not general. These unit questions do not yet supply a measure. The CSCI questions convert the answers to a normalized numeric answer. It is the ratio of the number of units containing limitations to the number of units having data. The worst case of zero for question c occurs when all of the units contain some limitations. If all are free of such limitations, the value for question c is one. This equation produces a normalized result. The b/a factor would never be indeterminate due to a question a value of zero, since, when there are no units that process data or receive input data, the respective metric element questions do not apply.

These measures address restrictive algorithms and programming techniques. An example of GE.2(3) follows:

```
        DO 20 J = 1,100
        SUM = SUM + X(J)
20      CONTINUE
```

By placing a constant (100) as the upper limit to the loop, "SUM" will be incremented 100 times. The unit would be more general if the upper limit were specified by some variable which would adjust the processing to fit any number of data items in the array "X".

The metric element, GE.2(4), addresses the value of the data items entered. A unit with data inputs is usually designed to ensure that users cannot enter out of range or otherwise unreasonable values. This design is necessary to prevent errors. However, if the checks are too restrictive, the unit may not be usable in a broader context. The more restrictive the inputs, the more specific the unit.
For example, a program that calculates an aircraft engine's performance uses the performance curves established for that engine. The program accepts data such as ambient temperature and pressure, revolutions per minute (rpm), air flow, fuel flow, and other pressure and test points on different stages of the engine. After test engineers enter the data, the program calculates whether

the engine meets performance requirements. This unit of code could apply to a specific turbojet engine, or the program could be expanded to evaluate a variety of engines, which may also include turbofan and turboshaft engines. This metric element can be used to determine how easily the unit accommodates the range of data required for other engines.

3.4.9 SD.2 (Self-Descriptiveness) Effectiveness of Comments Metric

The SD.2 metric determines whether code comments effectively explain how the function is implemented. It only addresses the quality of the comments that are present. Another SD metric quantifies the amount of commenting. This metric is based on eight metric elements. Only two are discussed here. These metric elements use the same CSCI questions as GE.2(2).

SD.2(1) Unit question (Worksheet 4B):

> d. "Are there prologue comments which contain all information in accordance with the established standard?" Y, N, N/A

Question d implies that units with prologue comments containing all the standard information are effective comments. The unit question does not supply a measure. The CSCI questions convert the answers to a normalized numeric answer. It is the ratio of the number of units whose prologue comments satisfy the standard to the number of units for which prologue comments are required. The worst case of zero for question c occurs when none of the units contain complete comments. If all contain complete prologue comments, the value for question c is one. This equation produces a normalized result. The b/a factor would never be indeterminate due to a question a value of zero, since, when no units are required to have prologue comments, the metric element questions do not apply.

The comments at the beginning of a unit of code should summarize its function and its components. The standard for these comments might require information such as the unit's name, the version number and date, and the author's name. These comments should also explain the purpose of the unit, its limitations, and assumptions. The header should specify the inputs and outputs.

Figure 3.4-3 shows the header information for a unit of code. This example shows a well-documented header. Pertinent information is included to help the reader understand the function of the code. Good header information makes the source code self-descriptive.

```
.COMMENT\
******************************************

    A COMPUTER COMPANY
    (C)  1991
    330 DEVON ROAD
    LINCOLNSHIRE, ENGLAND
    DEVELOPED BY NANCY VANSUETENDAEL
    APRIL 1991

******************************************

TITLE:      FLIGHT SIMULATOR
            UNIT 10
            REVISION 2

PURPOSE:    THIS UNIT GIVES THE ROLL, PITCH, AND YAW EQUATIONS
            FOR THE SIMULATOR.  THE ALGORITHMS ARE ONLY GOOD FOR
            AN F-16 FIGHTER AND THEY ASSUME THE INPUT DEVICE IS
            A JOYSTICK.

INPUTS:     JOYX - THE X VALUE FROM THE JOYSTICK
            JOYY - THE Y VALUE FROM THE JOYSTICK

OUTPUTS:    ROLL  - THE ROLL VALUE
            PITCH - THE PITCH VALUE
            YAW   - THE YAW VALUE
.COMMENT\
```

FIGURE 3.4-3. HEADER INFORMATION FOR A UNIT

This metric element is repeatable, but it is not monotonic. It makes no attempt to assess the quality of the comments. If a required comment exists, it assumes that the comment is effective. This relationship is not guaranteed.

Other metric elements contribute to the measure of comment effectiveness. SD.2(3) also uses the same CSCI questions, but focuses on the effectiveness of comments used to explain branching.

SD.2(3) Unit question (Worksheet 4B):

d. "Are all decision points and transfer of control commented?" Y, N, N/A

Question d implies that units with comments explaining all decision points and transfers of control contain effective comments. The unit question does not supply a measure. The CSCI questions convert the answers to a normalized numeric answer. It is the ratio of the number of units whose decision points and transfers of control are all commented to the number of units that contain decision points and transfer of control. The worst case of zero for question c occurs when none of the units contain completely commented control decisions and transfers. If all do, the value for question c is one. This equation produces a normalized result. The b/a factor would never be indeterminate due to a question a value of zero, since, when no units contain control decisions and transfers, the metric element questions do not apply.

Commented control flow makes the function of the code more evident. This is an effective use of comments. Figure 3.4-4 shows comments that explain the decisions in a program.

```
REM  Have all records been scanned?
IF TREE$(I,1) = " THEN GOTO 2050

REM  Have all subroutines been scanned in the file?
IF TREE$(SUB,2) < > "1" THEN GOTO 1910
```

FIGURE 3.4-4. COMMENTED DECISION POINTS

3.4.10 Scoring the Metrics

Once the developer chooses metric elements, the questions are answered and the answers recorded on the metric worksheets. This data is then transferred onto factor scoresheets. These scoresheets can be applied at any phase of the life cycle of the project. A factor scoresheet contains the headings shown in table 3.4-1. The first column lists some of the SI metric elements to show how the various scores are calculated for a particular case.

TABLE 3.4-1. FACTOR SCORESHEET EXAMPLE

Factor Scoresheet - Reliability

Phase ______________________________

Metric Element	Metric Element Score	Metric Score	Criteria Score	Factor Score
SI.1(1) SI.1(2) . . .	list of the SI.1 element scores	average of the metric element scores that apply to this metric	average of the metric scores that apply to this criterion	average of the criteria scores that apply to this factor
SI.2(1) SI.2(2) . . .	list of the SI.2 element scores	average of the metric element scores that apply to this metric		

Any of the averages can be simple or weighted. In the fourth column, other criteria may be listed. For example, the full factor scoresheet on reliability also includes the criteria, accuracy (AC) and anomaly management (AM). Several criteria scores may then relate to a particular factor. Each factor score sheet will have one factor's score, e.g., reliability.

3.4.11 General Analysis of RADC's Software Metrics

RADC's software metrics are comprehensive. They measure numerous attributes of code and thus can be related to several quality factors. Most of these measures are monotonic and repeatable provided the implementor follows clarified questions and standard counting rules for each. While each metric is simple, the entire hierarchy is so broad and deep that these metrics are laborious to implement. To be worthwhile, the metrics should be automated.

Experimental validation of these software metrics began with the study by Boehm et al. (1978). They applied 151 candidate metrics to FORTRAN code. Based on an evaluation of the results, they refined this list to 93 that proved to be mutually exclusive and comprehensive. Each metric was evaluated on how well it correlated to the attribute measured, and on its importance, quantifiability, and the ease and completeness of automating it. Of these, 56 were found to satisfy or nearly satisfy all five evaluations.

RADC then validated a similar set of metrics based on the work of Boehm et al. and many others. RADC applied metrics to two large-scale software systems that had been developed for the U.S. Air Force. They identified 41 metrics that could be used to specify quality requirements objectively in the systems requirements documents. Specifically, they found that "the regression analysis showed significant correlation for some metrics with related quality factors." (McCall, Richards, and Walters 1977). However, they further noted that scores were biased toward the high side because the systems were fully developed.

Later, RADC took these metrics from the research arena and refined them so that they could be tested in actual use. This refined set of metrics was applied during the development of two of four large U.S. Air Force programs. The metrics were evaluated. Warthman (1987) found that the metrics proved to be useful and cost effective. He emphasized the importance of clarifying the metric questions and of automating them to improve the implementation efficiency and objectivity.

A similar, independent study was conducted simultaneously on the other two programs. Pierce, Hartley, and Worrells (1987) found that "the metric framework itself is sound and reasonable". They also added that the metric element questions are sometimes ambiguous and confusing. They recommended that examples be provided.

Recently, another study concluded, "The current software quality specification process is too detailed and is not validated." (Lasky, Kaminsky, and Boaz 1990). Nevertheless, the utility of these metrics has been consistently confirmed, despite unresolved problems. However, the metrics still need to be validated by applying them during development.

The following tables list all the software metrics of this methodology. They are copied from Volume II of the Specification of Software Quality Attributes Software Quality Evaluation Guidebook (Bowen, Wigle, and Tsai 1985). Table 3.4-2 lists each software metric, by criterion, with its name and acronym. Table 3.4-3 defines each criterion. The complete methodology relating these criteria to various quality factors is discussed in chapter 5.

TABLE 3.4-2. QUALITY METRICS SUMMARY

Criterion		Metric	
Name	Acronym	Name	Acronym
Accuracy	AC	Accuracy Checklist	AC.1
Anomaly Management	AM	Error Tolerance/Control Improper Input Data Computational Failures Hardware Faults Device Errors Communications Errors Node/Communication Failures	AM.1 AM.2 AM.3 AM.4 AM.5 AM.6 AM.7
Application Independence	AP	Database Management Implementation Independence Data Structures Architecture Standardization Microcode Independence Functional Independence	AP.1 AP.2 AP.3 AP.4 AP.5
Augmentability	AT	Data Storage Expansion Computation Extensibility Channel Extensibility Design Extensibility	AT.1 AT.2 AT.3 AT.4
Autonomy	AU	Interface Complexity Self-Sufficiency	AU.1 AU.2
Commonality	CL	Communications Commonality Data Commonality Common Vocabulary	CL.1 CL.2 CL.3
Completeness	CP	Completeness Checklist	CP.1
Consistency	CS	Procedure Consistency Data Consistency	CS.1 CS.2
Distributedness	DI	Design Structure	DI.1
Document Accessibility	DO	Access to Documentation Well-Structured Documentation	DO.1 DO.2
Effectiveness-Communication	EC	Communication Effectiveness Measure	EC.1
Effectiveness-Processing	EP	Processing Effectiveness Measure Data Usage Effectiveness Measure	EP.1 EP.2
Effectiveness-Storage	ES	Storage Effectiveness Measure	ES.1
Functional Overlap	FO	Functional Overlap Checklist	FO.1
Functional Scope	FS	Function Specificity Function Commonality Function Selective Usability	FS.1 FS.2 FS.3
Generality	GE	Unit Referencing Unit Implementation	GE.1 GE.2
Independence	ID	Software Independence From System Machine Independence	ID.1 ID.2
Modularity	MO	Modular Implementation Modular Design	MO.1 MO.2
Operability	OP	Operability Checklist User Input Communicativeness User Output Communicativeness	OP.1 OP.2 OP.3
Reconfigurability	RE	Restructure Checklist	RE.1

TABLE 3.4-2. QUALITY METRICS SUMMARY (Continued)

Criterion		Metric	
Name	Acronym	Name	Acronym
Self-Descriptiveness	SD	Quantity of Comments	SD.1
		Effectiveness of Comments	SD.2
		Descriptiveness of Language	SD.3
Simplicity	SI	Design Structure	SI.1
		Structured Language or Preprocessor	SI.2
		Data and Control Flow Complexity	SI.3
		Coding Simplicity	I.4
		Specificity	SI.5
		Halstead's Level of Difficulty Measure	SI.6
System Accessibility	SS	Access Control	SS.1
		Access Audit	SS.2
System Clarity	ST	Interface Complexity	ST.1
		Program Flow Complexity	ST.2
		Application Functional Complexity	ST.3
		Communication Complexity	ST.4
		Structure Clarity	ST.5
System Compatibility	SY	Communication Compatibility	SY.1
		Data Compatibility	SY.2
		Hardware Compatibility	SY.3
		Software Compatibility	SY.4
		Documentation for Other System	SY.5
Traceability	TC	Cross Reference	TC.1
Training	TN	Training Checklist	TN.1
Virtuality	VR	System/Data Independence	VR.1
Visibility	VS	Unit Testing	VS.1
		Integration Testing	VS.2
		CSCI Testing	VS.3

TABLE 3.4-3. QUALITY CRITERIA DEFINITIONS

Acquisition Concern	Criterion	Acronym	Definition
PERFORMANCE	Accuracy	AC	Those characteristics of software which provide the required precision in calculations and outputs
	Anomaly Management	AM	Those characteristics of software which provide for continuity of operations under and recovery from non-nominal conditions
	Autonomy	AU	Those characteristics of software which determine its non-dependency on interfaces and functions
	Distributedness	D	Those characteristics of software which determine the degree to which software functions are geographically or logically separated within the system
	Effectiveness-Comm.	EC	Those characteristics of the software which provide for minimum utilization of communications resources in performing functions
	Effectiveness-Processing	EP	Those characteristics of software which provide for minimum utilization of processing resources in performing functions
	Effectiveness-Storage	ES	Those characteristics of the software which provide for minimum utilization of storage resources
	Operability	OP	Those characteristics of software which determine operations and procedures concerned with operation of software and which provide useful inputs and outputs which can be assimilated
	Reconfigurability	RE	Those characteristics of software which provide for continuity of system operation when one or more processors, storage units, or communications links fail(s)
	System Accessibility	SS	Those characteristics of software which provide for control and audit of access to the software and data
	Training	TN	Those characteristics of software which provide transition from current operation and provide initial familiarization
DESIGN	Completeness	CP	Those characteristics of software which provide full implementation of the functions required
	Consistency	CS	Those characteristics of software which provide for uniform design and implementation techniques and notation
	Traceability	TC	Those characteristics of software which provide a thread of origin from the implementation to the requirements with respect to the specified development envelope and operational environment
	Visibility	VS	Those characteristics of software which provide status monitoring of the development and operation
ADAPTATION	Application Independence	AP	Those characteristics of software which determine its nondependency on database system microcode, computer architecture, and algorithms
	Augmentability	AT	Those characteristics of software which provide for expansion of capability for functions and data
	Commonality	CL	Those characteristics of software which provide for the use of interface standards for protocols, routines, and data representations
	Document Accessibility	DO	Those characteristics of software which provide for easy access
	Functional Overlap	FO	Those characteristics of software which provide common functions to both systems
	Functional Scope	FS	Those characteristics of software which provide commonality of functions among applications
	Generality	FE	Those characteristics of software which provide breadth to the functions performed with respect to the application
	Independence	ID	Those characteristics of software which determine its nondependency on software environment (computing system, operating system, utilities, input, output routines, libraries)
	System Clarity	ST	Those characteristics of software which provide for clear description of program structure in a non-complex and understandable manner
	System Compatibility	SY	Those characteristics of software which provide the hardware, software, and communication compatibility of two systems
	Virtuality	VR	Those characteristics of software which present a system that does not require user knowledge of the physical, logical, or topological characteristics
GENERAL	Modularity	MO	Those characteristics of software which provide a structure of highly cohesive components with optimum coupling
	Self-Descriptiveness	SD	Those characteristics of software which provide explanation of the implementation of functions
	Simplicity	SI	Those characteristics of software which provide for definition and implementation of functions in the most noncomplex and understandable manner

3.5 Albrecht's Function Points Metric

Albrecht developed a metric to determine the number of elementary functions, hence, the value, of source code. His metric was developed to estimate the amount of effort needed to design and develop custom application software (Albrecht and Gaffney 1983).

The Function Points (FP) Metric has been used in industry (most notably in IBM) as an alternative to using various size metrics to estimate the amount of function the software is to perform (Albrecht and Gaffney 1983). It is essentially a complexity software metric. They link the results of the FP Metric to productivity, to effort required for a project, and to the amount of time necessary to complete a project.

While the FP Metric has been used extensively in code applications, including avionic code projects (Quantitative Software Management Inc. 1990), it has also been applied to Management Information Systems applications. It has been used to give an estimate of project size. By extension, the project size has been related to productivity. Managers use the measure as a productivity tool to estimate the number of hours of labor needed for a project, and to estimate future code maintenance.

The functions of a program are divided into five function types or attributes of source code. The number of functions within each type is counted. These five totals are then assigned a weight, based on experimental evidence that reflects each function's importance to the user.

The weighted totals are added together and this new total is adjusted to account for other factors that influence the application, or the processing complexity. This adjustment can vary up to plus or minus 35 percent.

Albrecht recommends applying weights to the five types of functions for programs of average complexity in the following manner:

- Number of external inputs X 4
- Number of external outputs X 5
- Number of logical internal files X 10
- Number of external interface files X 7
- Number of external inquiries X 4

The following paragraphs explain the types of functions, and the complexity levels associated with each. Counting rule considerations are examined.

3.5.1 External Input Type

External inputs are unique user data or user control inputs that enter the application being measured, and that add or change data in a logical internal file (Albrecht and Gaffney 1983). An input crosses the boundary of an application when it is written into any memory location that the program accesses. Examples of external inputs include terminal screen fields, scanner forms, and punch cards.

For this function type, each unique input is counted as a function. An input is considered unique if it has a different format or requires different processing logic than other inputs (Albrecht and Gaffney 1983). For example, in a COBOL application, an "ADD" transaction, which uses two different screen formats, would be counted as two functions. In another application, a "DELETE" transaction may use the same screen format as another "DELETE" transaction, but different processing logic. Each "DELETE", in this case, would be counted as a separate function.

Each external input function must then be classified as fitting into one of three levels of complexity: simple, average, or complex. Inputs are simple when they contain few data element types, and when they reference few logical internal files. Inputs are complex when they contain many data element types, and when they refer to many logical internal file types. Average external input functions are those not clearly simple or complex.

For counting external input functions, do not include the following:

- External inputs that provide no function, but occur because of the technology used. For example, on one system, the user enters a value and the next screen is displayed automatically. Another system requires that the user press CR/LF after a value is entered. Pressing CR/LF provides no additional function. It is simply a keystroke required by the software technology.

- Inputs from files, because these are counted as external interface files.

- The input part of the external inquiries.

To ensure correct counts, the metrician must understand the application. In some cases, whether the screen provides a function or not may depend on the way the software uses the information entered at that screen.

For example, consider an application that displays software change reports to users. The initial screen of this application requires that if the user wishes to enter a change to a report, he or she must enter "C" followed by CR/LF to get to the next screen. The program interprets the "C" and gives read/write privileges in the next menu. The same initial screen interprets a "V" as a request to view a screen. If the user were to select this latter option, the program would give only a read privilege. Hence, the internal processing of data provided to the initial screen differs, depending on the user's selection. Considering the screen as a way of getting to the next screen is incorrect. In this case, the second screen should be counted twice.

If a program reader incorrectly undercounts the number of input screens, the measure will underestimate the actual number. Conversely, if a program reader incorrectly adds a screen to the count, the measure will overestimate the actual number.

Another possible source of error is found if a data form is not clearly defined. The rules for defining input forms and screens must be consistently followed. Also, it is likely that different people will define the forms differently.

Finally, large variations in function counts (FCs) may occur if practitioners assign different complexity levels to the function. In fact, differences in how one perceives a function's complexity can have a dramatic effect on the total FP count, since the weight for a complex function is on average approximately double that of a simple user function (Low and Jeffery 1990).

3.5.2 External Output Type

An external output provides data or control information to a user or to another application. An external output leaves the boundary of the application (Albrecht and Gaffney 1983). Examples include printed reports and terminal screens.

For this type of function, each unique output is counted. An output is considered unique if it has a format that differs from other external outputs or if it requires unique processing logic for calculating the output data (Albrecht and Gaffney 1983).

Each external output type should be classified into one of the three levels of complexity: simple, average, and complex. A simple external output has a one-or two-column format. It has single-step data element transformations. An average external output has a multiple-column format with subtotals. It may contain multiple-step data element transformations. An example of average output would be a program that converts the density of a material from the mks (meters/kilograms/seconds) system to the centimeters/grams/seconds (cgs) system. A complex output may have intricate data element transformations. It may need to correlate multiple and complex file references. This last category may have significant performance considerations (Albrecht and Gaffney 1983).

For counting this function, do not include the following:

- External outputs that provide no function, but occur because of the technology used.

- Outputs to files, because these are counted as external interface files.

- Output responses of external inquiries.

The same counting considerations should be examined for output screens as for input screens.

3.5.3 Logical Internal File Type

A logical internal file is a machine-readable logical file or major logical grouping of data and control information that is generated, used, and maintained by the application (Albrecht and Gaffney 1983). Examples include card files, disk files, tape files, and memory resident files.

Each major data group within a database constitutes one logical internal file. The functions of the logical internal file type should be classified into one of three levels of complexity: simple, average, or complex. A simple internal file is one containing few record types. It also has few data element types. There are no significant performance considerations (Albrecht and Gaffney 1983). Complex has many record and data element types. Average is neither simple nor complex.

For counting this function type, do not include logical internal files that the user cannot access through external inputs, outputs, interface files, or inquiries.

3.5.4 External Interface File Type

External interface files are those files that are passed or shared between applications (Albrecht and Gaffney 1983). Each major logical group of user data or control information that enters or leaves the application is counted as one function. These functions should be classified by complexity, using the definitions listed in the previous section, Logical Internal File Type.

An interface file type may be defined differently, depending on where it is found in the application. On the module level, a module may interact with another set of modules, and thus be defined as an external interface file. On a system level, this same module becomes an input to the other modules in the application; it is now considered an external input file. How one classifies a module depends on the context.

Each external interface file is counted twice, once as an internal file, and a second time as an external interface file. For example, Application A contains a module called COSINE. This module is counted as an internal file because it is internal to the application. However, it also is an external interface to Application B. Hence, it is counted twice because it is both an input and an output, depending on which application is being analyzed.

Intuitively, one might assume that each file should be counted only once. But Albrecht clearly states that all functions are counted once. Since the file performs two separate functions, it is counted twice.

3.5.5 External Inquiry Type

An external inquiry is an input and response couplet where an input generates and directly causes an immediate output (Albrecht 1979). Inquiries can be generated by users or by another application. Each uniquely formatted or processed inquiry that searches a file or summary and presents the results of the search should be counted as one.

External inquiries should be classified into a level of complexity, by first classifying the input part of the query using the complexity definitions for the external inputs. The output part of the query is then classified using the definitions for the external outputs. The complexity of the inquiry is the more complex of the two classifications.

An external inquiry searches for specific data and is usually based on a single key. A query facility is more complex, usually requiring many keys and operations (Albrecht and Gaffney 1983). Every inquiry of a query facility should be counted separately.

This measure may not yield monotonic results in cases where the rules for defining an inquiry are not clear. For example, with an inquiry facility, a program reader may judge that one of the inquiries consists of a single, complex key operation. Another reader may refine the same inquiry into several subordinate inquiries, with simple keys. The rules must account for how finely inquiries should be resolved.

3.5.6 Obtaining the Function Count

To obtain the FC, the total count for each function type must be multiplied by its complexity weight. Each category's subtotal is then added together to obtain the FC, also called the Total Unadjusted FP on the worksheet. Albrecht created and supports the example work sheet IBM used for their FP analyses (Albrecht 1983), shown in figure 3.5-1.

Function Count:					
Type			Complexity		Total
ID	Description	Simple	Average	Complex	
IT	External Input	_ X 3 = _	_X 4 = _	_ X 6 =	
OT	External Output	_ X 4 = _	_ X 5 = _	_ X 7 =	
FT	Logical I. File	_ X 7 = _	_ X 10 = _	_ X 15 =	
EI	Ext Interface F.	_ X 5 = _	_ X 7 = _	_ X 10 =	
QT	Ext Inquiry	_ X 3 = _	_ X 4 = _	_ X 6 =	
FC		Total Unadjusted Function Points:			

FIGURE 3.5-1. SAMPLE FUNCTION COUNT WORK SHEET

This work sheet shows the weights that IBM determined provide the best results from their measurements. These weights are constants in the form. Although the number of complexity levels is arbitrary, the three function complexity categories in this table are used by IBM in many of their software projects. A developer could design his or her own complexity scale using different categories, e.g., very simple, simple, average, complex, or very complex. The developer must then determine weights for each experimentally.

After the FC has been calculated, it must be adjusted for Processing Complexity effects.

3.5.7 Calculating the Processing Complexity

While the FC accounts for most code attributes, this count may not represent the "amount of function" in a particular application. Other factors may affect the count. For instance, an application may have complex data communications. This complexity is not detected by any of the function types. Albrecht and Gaffney list 14 characteristics that could influence the FP calculation (1983):

1. Data Communications. The extent to which an application sends and receives data and control information over communication lines.

2. Distributed Functions. The extent to which the software contains distributed data or functions.

3. Performance. The extent to which performance objectives, such as the application's run time performance, influenced the design of the program. These objectives can apply to the requirements phase through the maintenance phase.

4. Heavily Used Configuration. The extent to which it is anticipated the equipment will be used.

5. Transaction Rate. The extent to which the transaction rate influenced the design, development, installation, and support of the application.

6. Online Data Entry. The extent to which the application allows data to be entered online.

7. End User Efficiency. The ease with which users can enter data. (Depends on characteristic 6.)

8. Online Updating. The extent to which an application allows its logical internal files to be updated while the application is running.

9. Complex Processing. The extent to which the application contains complex processing, such as

   - many control interactions and decision points,
   - extensive logical and mathematical equations, and
   - much exception processing due to uncompleted transactions.

10. Reusability. The extent to which the design or code development was influenced by making it reusable for other applications. For example, if the application contained a special math run-time library, the designer may have used constructs in the library routines that were less machine-dependent so the library could be used on several system configurations.

11. Installation Ease. The extent to which a plan was created and tested to specify how the code should be installed or converted.

12. Operational Ease. The extent to which effective initializing, back-up, and recovery procedures were provided, and were tested during the system test phase. It assumes the

software is self-sufficient. It also addresses the extent to which a user must intervene in the program.

13. Multiple Sites. The extent to which the application can be used at many sites and by different organizations.

14. Facilitating Change. The extent to which a programmer can make changes to the data that a program accesses.

These characteristics are weighted according to their relative significance (or degree of influence) in the project. Albrecht and Gaffney created the example work sheet shown in figure 3.5-2 for IBM.

Processing Complexity:

ID	Characteristic	DI		ID	Characteristic	DI
C1	Data Communications	__		C8	Online Update	
C2	Distributed Functions	__		C9	Complex Processing	
C3	Performance	__		C10	Reusability	
C4	Heavily Used Configuration	__		C11	Installation Ease	
C5	Transaction Rate	__		C12	Operational Ease	
C6	Online Data Entry	__		C13	Multiple Sites	
C7	End User Efficiency	__		C14	Facilitate Change	
PC					Total Degree of Influence	

Degree of Influence (DI) values:

- Not present, or no influence	=	0	- Average influence	=	3
- Insignificant influence	=	1	- Significant influence	=	4
- Moderate influence	=	2	- Strong influence throughout	=	5

FIGURE 3.5-2. SAMPLE PROCESSING COMPLEXITY WORK SHEET

As the worksheet indicates, the Processing Complexity is given by the Total Degree of Influence of the application characteristics. This value is used to adjust the FC. The characteristics are a guideline for the practitioner to follow. Other characteristics can be defined and can be included in the list. The Degree of Influence categories have been established experimentally. They provide guidelines only. A practitioner could establish different degrees of influence, such as insignificant, moderately significant, or extremely significant, if desired.

3.5.8 Obtaining the Adjusted Function Count Value

Once weights have been assigned to the characteristics, the following equation can be used to obtain the Processing Complexity Adjustment (PCA):

$$PCA = 0.65 + (0.01 \text{ X Processing Complexity})$$

The FP Metric then can be calculated:

$$FP = FC \text{ X } PCA$$

where the FC is the Total Unadjusted FP, as noted in the worksheet of figure 3.5-1.

This metric produces a real number greater than or equal to 1.30 (two multiplied by 0.65). This number is the lower limit, assuming the simplest algorithm will contain only two functions, an input and an output. It also assumes that each function weight is one, and the Processing Complexity is 0.

3.5.9 General Analysis of Albrecht's Function Points Metric

In general, if a program reader incorrectly identifies an input, output, internal file, interface file, or inquiry, the measure will not be monotonic or repeatable. In the case where this type of error has been made, an increase in the measure may not represent an increase in the amount of function, or vice versa. Also, different readers may define the function types differently.

Errors may occur as each step of the calculation is performed. Typically, these errors result when two practitioners assign different weights to either the function types or the influences. The following paragraphs discuss the errors that may affect the metric result.

In the first step where the Total Unadjusted Function Points are calculated, estimates can vary depending on the complexity assigned to each function type. Since the function types are weighted, and the weighting for a complex type is on average approximately double that of a simple type, the mean estimate may vary significantly (Low and Jeffery 1990). For example, if practitioner A assigns a simple complexity to outputs, and practitioner B assigns a complex rating to the same function type in the same application, B's assignment will give the function type twice as much weight in the results as compared with A's assignment.

Another error occurs when the actual number of functions are estimated. One program reader may think two reports are similar enough that only one function is performed. A different reader may insist that they are not identical, and that the reports constitute two functions.

In the second step where the Processing Complexity is calculated, practitioners may differ on the degree of influence that should be assigned to each characteristic. Since the PCA can influence the Total Unadjusted Function Point count up to ±35 percent, differences in evaluating these influences can have a dramatic effect on the FP value. Again, different practitioners may assign different degrees of influence to the characteristics.

Another problem occurs when practitioners define the boundaries of an application differently (Low and Jeffery 1990). Albrecht never clearly defines what constitutes a boundary, assuming the definition to be self-evident. If boundaries are perceived differently, functions may be interpreted differently, leading to results that are not monotonic.

For example, files that may be logical internal files to a system may be considered external interface files to an individual program (Low and Jeffery 1990). If practitioners do not agree on the function types, each FC could be misrepresented. Should a fast batch response count as an inquiry, or as an input and an output? In the former case, the count is one; in the latter, two. However, if these problems are addressed, it is possible to obtain a monotonic and repeatable measure.

The FP Metric was designed to measure the amount of function in a project. Since the function of an application typically will remain invariant, Albrecht (1985) maintains that this software metric, by definition, has the following properties:

- The measure is independent of technology. It gives the same number for functionally equivalent applications, regardless of the technology used to implement the applications.

- The measure is language-independent. As long as the function remains invariant, a program implemented in any language will produce the same FP measure.

- The measure is independent of programmer style.

- The measure will be invariant, regardless of the development environment, e.g., human factor considerations.

What appears to be this measure's greatest strength, turns out to be one of its weaknesses with respect to our criteria: it need not be calculated from code. The FP Metric is based on the design. To use the metric after code has been developed is to use it in a reverse engineering way. The metric ultimately reflects back to the original purpose of the software, found in the design specifications. Hence, the FP calculation can easily be performed using just these documents. But, to use the code to determine this metric means the practitioner must deduce the functional aspects that were determined in the design.

It is possible to use this measure on code without having the original design documents. In this case, it is absolutely essential that the practitioner have the code with which the module interacts. For example, if only modules of one part of an application are available for analysis, the practitioner may not be able to define the external interfaces to the code. Without this information, the FP analysis cannot be completely calculated. Even if the entire application were available, large applications would require that the practitioner spend a significant amount of time evaluating functions.

Albrecht claims the metric is language-independent. Since the function of an application remains invariant, presumably any language will yield the same result for the same function. Although studies support its use in a wide range of applications (Albrecht and Gaffney 1983; Behrens 1983;

and Quantitative Software Management, Inc. 1990), no studies have been published to substantiate whether this invariance holds.

3.6 Ejiogu's Software Metrics

Ejiogu's family of metrics uses language constructs to determine the structural complexity of a program. The syntactical constructs are nodes, which form a tree. A node is composed by grouping together the constructs that form a unit of activities or functions. In Ada, for example, procedures, packages, subroutines, and functions might define nodes. In FORTRAN, "DO", "IF", "DOWHILE", "PRINT", "READ", and "WRITE" constructs might define nodes. The control flow determines the connections among nodes.

The root node is the node from which all subsequent nodes branch. Based on this definition, a node might be a main program with all its subroutines, or just a subroutine, depending on the code the developer defines to represent a node.

Ejiogu uses two sets of terms to describe his methodology. When he speaks of height, he refers to a tree where the lowest level is the root node, and higher level nodes occur higher in the tree (up from the root). When he speaks of nesting nodes, the root node is the highest node, and subsequent nodes are lower than the root, i.e., the nesting structure forms an inverted tree. In this report, the nesting structure terms are used.

Four major software metrics have been developed, based on the node structure of the tree:

- Height of a Tree
- Twin Number
- Monadicity
- Size

In general, Ejiogu's metrics have not been empirically substantiated. Also, although they claim to measure structural complexity, they have not been correlated with this factor by independent experimental evidence.

3.6.1 Height of a Tree

The height (h) for an individual node is the number of levels that a node is nested below the root node. Thus, the root node has a height of zero, and any child of the root node has a height of one. The Height of a Tree (H) is the height of the deepest nested node. This metric produces an integer greater than or equal to one, since the Height of a single node tree is defined to be one.

When a developer implements the counting strategy for this measure, he or she must consider the following points to ensure the results of the measure will be monotonic and repeatable:

1. Nodes must be carefully defined to apply to a given context. Once what constitutes a node has been determined, the counting rules must be applied consistently to achieve a repeatable measure.

   For example, consider a FORTRAN program with a "DO" loop in a subroutine. One person defines a node to be the code contained within a "DO" and its corresponding "CONTINUE" statement. Another determines that a node consists of all statements that do not pass flow of control to other parts of the program.

   If both definitions are accidentally implemented, the counts will not be repeatable where the definitions diverge. The definitions would diverge if the "DO" loop contained a "CALL" statement. The "CALL" statement would pass control to another part of the program. In the first case, this "CALL" statement would not be considered another node. In the latter case, the "CALL" statement would be another node.

2. Nodes must be carefully evaluated and correctly interpreted. For example, a program reader may erroneously nest a node in drawing the tree. If this occurs, an increase in the measure would not represent an increase in the depth of nesting. A different reader may judge nesting differently.

   Where a tool has been developed to draw nodes, the tool must contain rules that properly detect the defined nodes. If a tool improperly detects the nodes, although the count may not be correct, it will be repeatable.

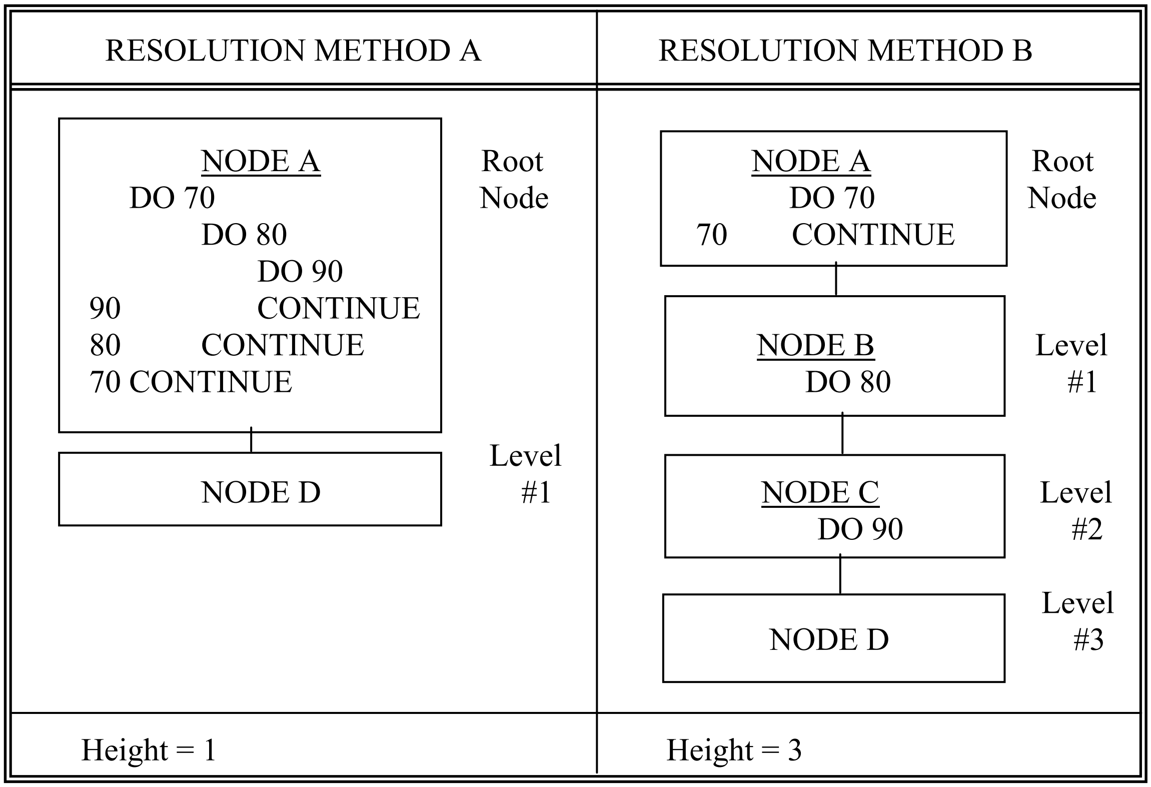
The metric's theoretical foundation is questionable. The count depends on how finely nodes are defined or "packaged". Different definitions will produce different results.

A depth resolution problem occurs if the nesting structure is not consistently refined. Consider the FORTRAN nested "DO" loops illustrated in figure 3.6-1.

```
      DO 70 I = 1,L
         DO 80 K = 1,M
           DO 90 J = 1,N
             MATC(I,K) = MATC(I,K) + MATA(I,J) * MATB(J,K)
 90        CONTINUE
 80      CONTINUE
 70   CONTINUE
```

FIGURE 3.6-1. FORTRAN NESTED DO LOOPS

The code in figure 3.6-1 can be refined differently, depending on the way the nesting is assigned as nodes. Figure 3.6-2 shows two ways of assigning nodes for this code.



```
NODE A
    DO 70
        DO 80
            DO 90
90          CONTINUE
80      CONTINUE
70 CONTINUE
```

```
NODE A
      DO 70
70    CONTINUE

NODE B
      DO 80

NODE C
      DO 90
```

FIGURE 3.6-2. TWO INTERPRETATIONS OF RESOLVING NESTED DO LOOPS

In figure 3.6-2, Method A keeps the embedded "DO" loops together as one node. Method B counts each embedded "DO" loop as a separate node. In the first case, the Height of the tree equals one because the lowest node is one level down from the root node. The second case shows the lowest node three levels down from the root node, thus the Height is three.

The metric does not account for the different resolutions. There is no mathematical relationship that has been devised to rectify the difference.

3.6.2 Twin Number

After the root node, the tree branches, acquiring breadth as the number of branches from that node increases. The Twin Number is the number of nodes that branch out from a higher level node (see figure 3.6-3, reading from top to bottom). In this figure, the higher level node is the parent. Each node stemming directly from it is assigned a number. The maximum number is the Twin Number of the parent node.

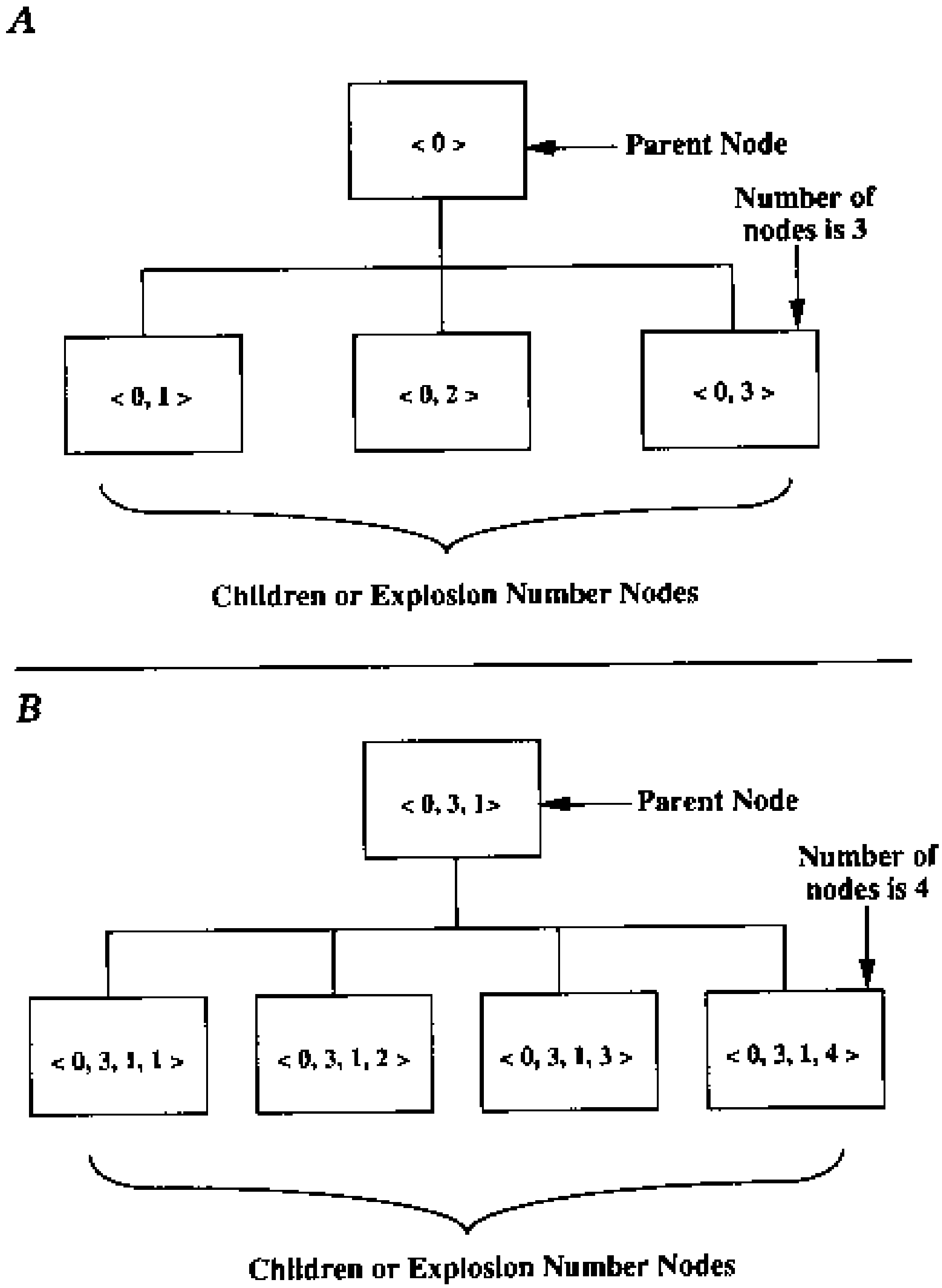


FIGURE 3.6-3. RELATIONSHIP BETWEEN PARENT AND CHILD NODES

A special case of the Twin Number is the Twin Number for the root (R_t). This special case is used as a factor to calculate structural complexity. In earlier literature, R_t was referred to as the "explosion number", perhaps to indicate how the first node can be expanded to show its first level of subordinate nodes. In the more recent literature, "explosion number" is a specific child node's number.

This metric produces an integer greater than or equal to zero, the Twin Number of a node that has no child nodes emanating from it. However, the Twin Number of the node in a single node tree is defined to be one.

Some of the same problems found in using Height also apply to the Twin Number; i.e., node definition ambiguities, misinterpretation of nodes, program reader errors. If the reader places a node in the wrong level of a tree, the measure will not properly represent the number of child nodes. Hence, the Twin Number would be misrepresented.

3.6.3 Monadicity

A monad is a fundamental, indivisible unit. Monads, in this context, are nodes that do not have branches emanating from them. They also are referred to as "leaf nodes". The monadicity is calculated by summing the number of monad nodes.

In figure 3.6-4, the top level node is the root. The Height of the Tree (H) is calculated by counting the levels. The Twin Number of the Root (R_t) is three, because the first level contains three nodes. From counting the monads (the circled nodes in the drawing) the monadicity is six. Each node is assigned an address to indicate its relative position within the tree. The root node is always assigned zero. Each node branching from the root is assigned two numbers; the next level (Level 2) is assigned three numbers. As the branching continues, node addresses increase by one number for each level.

Monadicity is an integer greater than or equal to one, the Monadicity of a single-node tree. This measure is repeatable and monotonic if the counting rules are followed consistently. If the program reader accidentally divides a monad into a node with children, an increase in the measure will not reflect an increase in the number of monads.

3.6.4 Structural Complexity

"The Structural Complexity metric gives a numerical notion of the distribution and connectedness of a system's components" (Ejiogu 1988). The S_c for a software module or system is as follows:

$$S_c = S_c (H, R_t, M) = H \text{ x } R_t \text{ x } M$$

where H is the Height of the tree, R_t is the Twin Number of the Root, and M is the Monadicity (Ejiogu 1990). An increase in any of these factors causes increased complexity. For example, the data from figure 3.6-4 shows H as four, R_t as three, and M as six. The structural complexity is, therefore, 72.

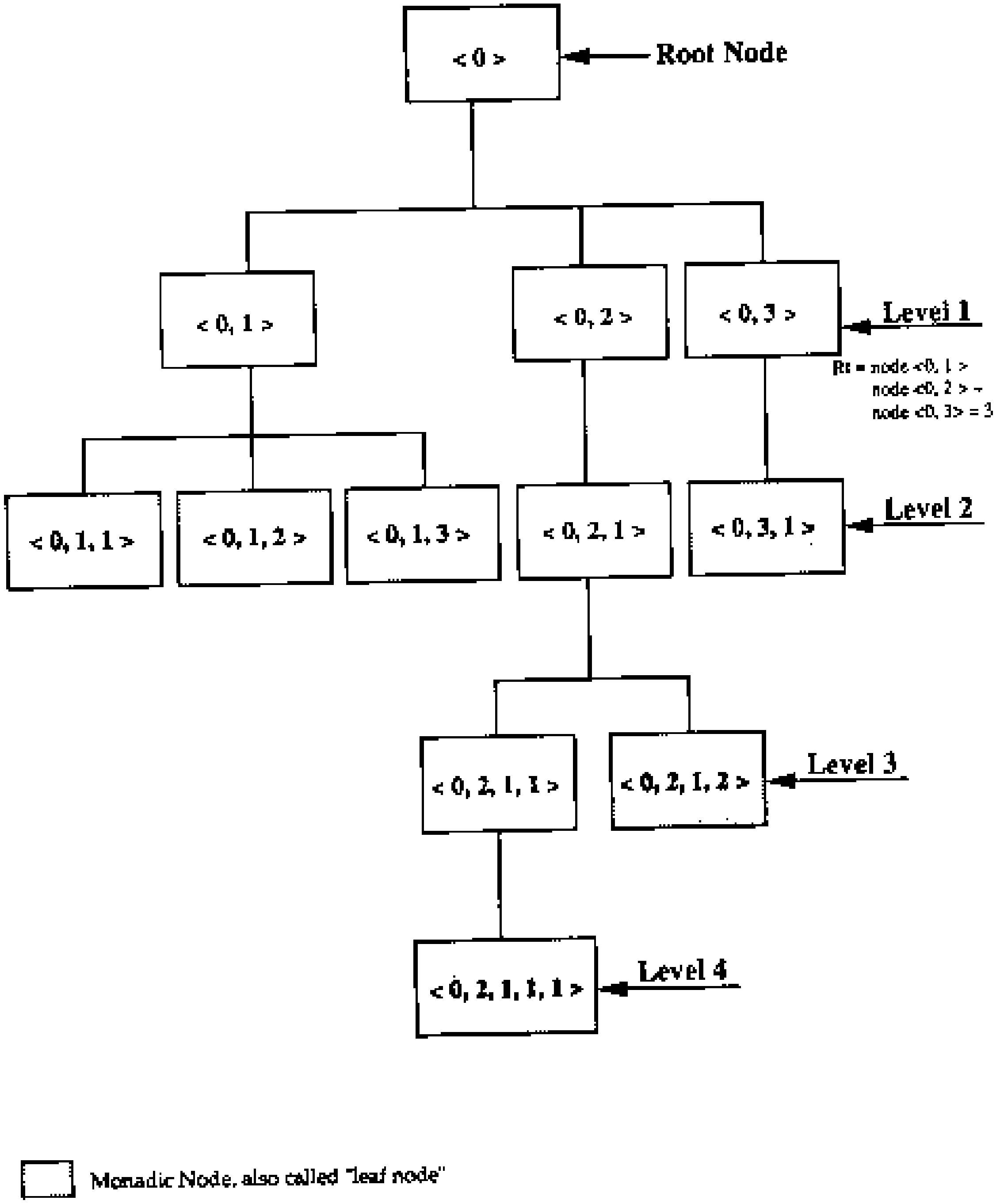


FIGURE 3.6-4. TREE STRUCTURE OF A PROGRAM

The Structural Complexity metric is based on structured programming techniques. If a program lacks structure, this measure reflects the lack by having a large S_c value. In general, a large value indicates a greater divergence from the ideal, a program consisting of one node, where S_c is one. The measure is related to a program's complexity: the higher the value of the Structural Complexity, the more complex the program.

Consider two programs, both written in the same language. The nodes are defined and the structural complexity calculated. The Structural Complexity is 4 for Program 1 and 100 for Program 2. What do these numbers mean? If the Structural Complexity is calculated for several other programs (assuming each program is written in the same language and the same node definitions are used), a scale of structural complexity can be devised.

Although the devised scale will be relative, it will show which programs have Structural Complexities that deviate most from the mean. The developer can then review the programs and can determine why they deviate. The developer may determine that modules with a significantly higher Structural Complexity number are overly-complex and that they should be recoded.

However, high values may not necessarily indicate excessive complexity. When two nodes call the same subroutine, S_c increases even though there is no increase in function. Yet calling the same subroutine twice is not poor programming practice. Rather, it makes the program more modular.

The metric's theoretical foundation is questionable. The count depends on how finely nodes are defined or "packaged". Different definitions will produce different results.

Consider an application where nodes have been defined at a specific level, so that a subroutine constitutes a node. Another view has the nodes defined at a lower level, where smaller groupings of code or single statements constitute nodes. The first calculation will be lower than the second. Figure 3.6-5 illustrates this difference.

In figure 3.6-5, the node counts are different because nodes are resolved at different levels. This is a breadth resolution problem because the number of nodes found in a horizontal level differ. The problem affects the Twin Number and the Monadicity counts.

As with Ejiogu's other software metrics, the measure will not be monotonic if more than one module calls the same subroutine.

RESOLUTION METHOD A	RESOLUTION METHOD B
NODE A Other Code Sine Function Code Cosine Function Code Tangent Function Code	NODE A Other Code NODE B Sine Function Code NODE C Cosine Function Code NODE D Tangent Function Code
Total = one node	Total = four nodes

FIGURE 3.6-5. TWO INTERPRETATIONS OF RESOLVING NODES

3.6.5 Software Size

Software size is the size of a set of nodes of source code. It is calculated using the number of nodes in the tree:

S = total # of nodes - 1

where the "1" represents the root node. For example, if a program contained 111 nodes, S would equal 110. This metric produces an integer greater than or equal to zero, the size of a single function. (A single function is considered zero because presumably it would be represented by a one node graph.)

The Size metric is not monotonic. If several modules call the same subroutine, that subroutine is represented as a child of each of the calling modules. Thus, depending on the number of modules that call the subroutine, it will be duplicated in the hierarchy tree that number of times. Consequently, the same code will be represented as different nodes, and the size will increase even though the function does not.

A program reader may incorrectly draw a node that should be a monad as a node with a child node. In this case, an increase in the measure does not represent an increase in the number of nodes. This metric also has the same problem the other metrics face: resolving nodes differently will result in a measure that is neither monotonic nor repeatable.

Additionally, the Size metric will not be monotonic or repeatable if several modules call the same subroutine. Each time the module is called, the hierarchy tree will show the subroutine. The count will increase by that number. Hence, height will increase even though the function does not.

3.6.6 General Analysis of Ejiogu's Software Metrics

Ejiogu's metrics are based on two questionable underlying assumptions:

1. The nodes will be defined consistently, based on the constructs of the language. This assumption does not account for people interpreting nodes differently.

2. Everyone will refine nodes to the same degree. For example, given a language and a set of programs, two implementors will establish the same number of nodes when they draw the hierarchy tree for a given program. This assumption implies that there is a standard degree of refinement. In practice, however, and given Ejiogu's loose definition of a node being an "aggregate of thought", this degree of refinement is not clear. Hence, it is unlikely two implementors will identify the same structures as nodes in a program. Further, once the trees have been drawn at different node resolutions (i.e, the trees for the same program differ in their numbers of nodes), Ejiogu does not provide a mathematical vehicle by which one can be brought into agreement with another.

In theory, these metrics are related to the structural complexity of a program. They are also related to other quality factors, such as usability, readability, and modifiability.

Ejiogu's metrics have not been experimentally correlated with any quality factor. No documented experiments have been conducted to verify them empirically. Additionally, no independent experiments have been performed.

To ensure all the measures will be monotonic and repeatable, the following points should be considered:

- The rules must take into account all the language constructs that might affect a node's grouping.

- If an automatic tool is developed to produce the counts, the tool must be programmed to implement the results established for defining nodes correctly.

- When the trees are drawn, the rules for defining nodes must be consistently followed.

- All subroutines must be called by only one node.

- If the counts are performed manually, one person should perform them all for different programs, if possible. Additionally, that person should be well-versed in the language to ensure the counts are correct, as well as repeatable.

3.7 Henry and Kafura's Information Flow Metric

The Information Flow Metric calculates the complexity of a module of source code based on its size and the flow of information contained within the module. It is a structural metric because it is based primarily on the structure of the flow of information.

The procedures that update or retrieve information from a data structure are grouped together to form a module. The flow of information between procedures consists of three types:

- Direct local information flow
- Indirect local information flow
- Global information flow

3.7.1 Direct Local Information Flow

Direct local flows of information are created when one procedure passes parameters to another procedure. Direct flows are the ones observed in the calling structure (Henry and Kafura 1981).

For example, if Procedure A passes a parameter through its calling arguments to Procedure B, the parameter passing is counted as one local flow, as demonstrated in figure 3.7-1.

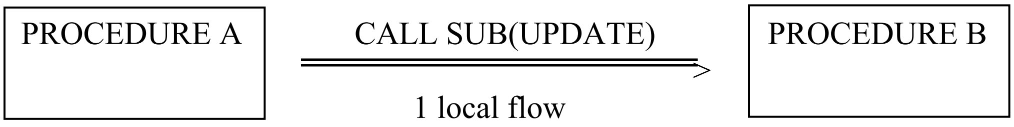


FIGURE 3.7-1. EXAMPLE OF DIRECT LOCAL INFORMATION FLOW

Direct information flow analysis clearly shows which module sends the information and which one receives it. Hence, program readers are less likely to make errors on this analysis. Similarly, an automated tool can be designed that will also produce repeatable results for this part of the count.

Depending on the size of the application, however, it is possible that a program reader may mistake inputs and outputs. It is also possible that a tool may not be designed to cover all code possibilities for inputs and outputs sufficiently. In either case, an increase in the measure may not represent an increase in the amount of information flow, or vice versa. Hence, the measure will not be totally monotonic or repeatable.

3.7.2 Indirect Local Information Flow

There are two types of indirect local information flow. The first type occurs when a called procedure passes information back to its calling procedure, which the calling procedure then uses in a computation. The second type occurs when a calling procedure receives information from its called procedure, and sends that information unchanged to another procedure. Figure 3.7-2 illustrates both types of indirect local flow.

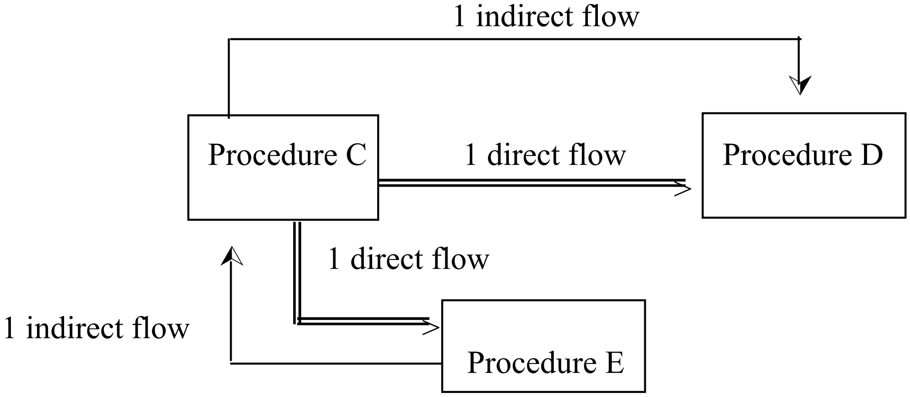


FIGURE 3.7-2. EXAMPLE OF INDIRECT LOCAL INFORMATION FLOW

When procedure E returns values to procedure C, which procedure C uses in a computation, this is an indirect local information flow. When procedure C passes unchanged information it received from procedure E to procedure D, this is the other type of indirect local information flow.

Counting indirect local information flows can produce several errors because as its name implies, it is not directly assessed. The program reader must understand the intricacies of the program to correctly analyze the inputs and outputs. Similarly, an automated tool must be able to determine the possibilities presented by analyzing indirect flows.

3.7.3 Global Information Flow

Not all types of data flow contribute to the measure. Any flow that is a global information flow does not contribute. Global information flow occurs when modules access or update information in a global data structure.

A program reader or an automated tool must distinguish local flow from global information flow.

3.7.4 Calculating the Information Flow Metric

The Information Flow Complexity (IFC) is the sum of the IFCs for each procedure contained in the module. The IFC is calculated as follows:

$$\text{IFC} = \text{Length x (Fan-in x Fan-out)}^2$$

where Length is the number of lines of source code in the procedure. In implementing this count, embedded comments are also counted, but not comments preceding the beginning of the executable code. The Fan-in of a procedure is the number of local flows that terminate at that procedure. The Fan-out is the number of local flows that emanate from the procedure.

Each type of local information flow is counted if the flow terminates within a procedure or emanates from the procedure. The Fan-in and Fan-out counts can be obtained by conducting a procedure-by-procedure analysis.

This metric produces an integer greater than or equal to one, since the smallest IFC consists of a one statement procedure that has one input and one output.

The formula is based on the assumption that the complexity of a procedure depends on two factors: the complexity of the procedure's code, and the complexity of the procedure's connections to its environment (Henry and Kafura 1984). Based on this assumption, the Length component of the formula represents the procedure's complexity, while the (Fan-in x Fan-out)^2 component represents the complexity of the procedure's connections to its environment.

Further, the equation is a quadratic because the authors assumed this "connectivity" component, (Fan-in x Fan-out)^2, accounted for the complexity of the interrelationships among modules in a system. This squared relationship has proved successful for other areas of computer analysis, such as programmer interaction and system partitioning (Henry and Kafura 1984).

In the study conducted by Henry and Kafura on the Unix^{TM} operating system (1984), the code did not contain many embedded comments, or comments that occurred before the procedure statement. Hence, the lack of comments allowed the authors to equate the total number of lines of source code to the sum of the total number of statements counted in each procedure. Where code contains many comments, Length should take into account the embedded comments. If the rules do not anticipate embedded comments, the Length count may be overestimated. The rules for inputs and outputs must be carefully defined so the count is repeatable and monotonic.

3.7.5 General Analysis of the Information Flow Metric

The Information Flow Metric can be used at several stages in the software life-cycle. If it used immediately after the design phase, the measurements require sufficient information to determine the inter-procedure data flows, and estimates of the code length (Henry and Kafura 1984). Whether the metric is used at the requirements stage or after code has been implemented, the rules must be carefully defined to generate reliable counts.

Henry and Kafura conducted an experiment where they applied this metric to the UNIX^{TM} operating system. From this analysis, they determined that a large number of information flow paths meant a larger system complexity. Further, the greater the complexity, the more problems will occur if the code is changed. The study found a significant statistical correlation of 0.95 between errors and procedure complexity as measured by the Information Flow metric (Kafura and Reddy 1987).

When Henry and Kafura analyzed the most complex modules in the system, they found that by dividing a module into more procedures, they were able to reduce the overall system complexity. This idea differs from other measures of complexity, such as a Lines of Code count, where adding lines of code to create separate modules indicates increased program complexity.

The measure indirectly applies to code. It uses source code to detect the information flow within a program, but it measures the logical structure, rather than the code itself. Hence, it is language independent and programmer-style independent.

The UNIX^{TM} study confirms that the Information Flow metric can form the basis for SQM because it is indirectly correlated with quality factors such as complexity, reliability, and maintainability. However, it has not been independently substantiated. Kafura and Reddy confirm this shortcoming:

> "While the consistency of our experience in this study leads us to believe that we have not been merely lucky in identifying complex components, we cannot, at this point, offer experimental evidence to soundly justify this conviction." (1987)

The IFC metric has been validated empirically to correlate with complexity on a large, complex operating system. However, it still needs further independent empirical substantiation. Its chief strengths stem from its composite design: it looks at both the lines of code of an application and the interrelationships among modules to assess the complexity of an application.

4. SOFTWARE QUALITY FACTORS

Software Quality Factors (SQFs) are those attributes that users consider important for software to possess (Bowen, Wigle, and Tsai 1985). When these factors are correlated with software metrics, the result is an SQM.

Quality factors are numerous, and the meaning of each depends on a given definition and measure. For example, one practitioner may measure complexity by counting the decisions in a program. Another may measure it by counting the source lines in a program. Which type of complexity is more valid? The converse also exists: two practitioners call the same software attribute two different names. Should the two names be combined? What are the qualifying assumptions in each case?

To address these issues, this section discusses the various definitions of a representative group of quality factors. The definitions are analyzed and interwoven to present a cohesive set of quality factors.

A quality factor can be selected at any time in the life-cycle of the product. The procedure for selecting a quality factor consists of four steps (Bowen, Wigle, and Tsai 1985):

1. Identifying functions

2. Assigning quality factors and goals

3. Considering interrelationships

4. Considering costs

The quality factors are selected based on the function of the software and the application's objectives. For example, a mission-dependent function will need to be more rigorously tested for reliability than one that is not critical to the mission.

The interrelationships among quality factors must be analyzed. Some SQFs may conflict. An application designed to be efficient in handling real-time transactions may be extremely difficult to maintain. In cases where the factors conflict and the practitioner still wishes to analyze opposing attributes, the metric scores can be weighted, giving the more important factors more significance in the results.

There can be several levels at which a quality factor contributes to SQM. For example, RADC's SQM definitions include a level of factors that occurs between the software metric and quality factor levels. The factors composing this level are called criteria. A quality factor is a user-oriented view of an aspect of product quality, while a criterion is a software characteristic that is assigned to the quality factor to which it contributes. This is an example of a hierarchy that separates quality factors into two groups: intermediate and high-level.

Most of the factors discussed in chapter 4 can be divided into other quality factors. Complexity is often divided into at least two types, structural and psychological. These categories tend to shift over time, as more correlations are found between software metrics and quality factors.

Complementary factors are those that share some criteria with another factor (Bowen, Wigle, and Tsai 1985). Hence, they have a cooperative relationship with a specific quality factor. Any project should include complementary factors in the quality specifications to ensure that the metric scores are true indicators of the total quality present.

Antagonistic factors, on the other hand, have attributes that conflict. Hence, the antagonistic factor is inversely proportional to the factor being discussed. A quality factor may be antagonistic under certain conditions, but may be complementary in other conditions. For example, conciseness normally is complementary to understandability. But, if a program is excessively concise (cryptic), one's understanding of the program is no longer enhanced. Hence, while no hard and fast rules exist to determine whether the specific attributes are complementary or antagonistic, guidelines for determining the relationship based on the "usual case" are provided.

The following quality factors are presented alphabetically. These factors represent important attributes of software products.

4.1 Accuracy

Accuracy is the extent to which a program's outputs are sufficiently precise to satisfy their intended use (Boehm et al. 1978; McCall, Richards, and Walters 1977; and Gilb 1977).

The degree of accuracy required for a program depends on its function. Because computer calculations can contain round-off errors, one must understand the application to know the margin of error allowed. In cases where great precision is necessary, higher level languages usually provide constructs that allow a larger number of digits to be stored to obtain greater accuracy. For example, in FORTRAN, the double-precision data type might allow a programmer to specify that computations hold values of 16 significant decimal digits, rather than only 10. The round-off error is thus reduced, and the accuracy of the calculation improved (although the error may still be significant, depending on the algorithm).

Quality factors antagonistic to accuracy include efficiency, simplicity, and portability. Complementary quality factors include complexity and precision.

4.2 Clarity

Clarity measures how clearly a person can understand a program (McCall, Richards, and Walters 1977). Clarity is the extent to which a program contains enough information for a reader to determine its objectives, assumptions, constraints, inputs, and outputs.

The clearer a program is, the easier it is to modify or maintain. A clear program is also easier to test, since the tester is able to understand the program's functions quickly. A clear program contains well-commented modules; variables that have meaningful descriptor names; and straightforward, traceable logic.

A clear program may not execute efficiently, however. In general, program readers find a program containing many modules easier to understand. This type of program, however, requires more Central Processing Unit (CPU) time to execute than it would if it were written as a single module. The extra CPU time is due to the extra instructions in the program that direct the program flow to individual modules.

Modularity, understandability, usability, and efficiency should all be considered in the measurements along with clarity because they are aspects of clarity, or are affected by the degree of clarity in a program.

A usable program contains a certain level of clarity. As a programmer writes code, he or she should try to use the constructs of a language effectively to increase clarity. Figure 4.2-1 shows an example of FORTRAN code that produces a table of squares and cubes of integers from 1 to 50.

```
          DO 90     I = 1,50
                    ISQ = I*I
                    ICUBE = I*I*I
                    WRITE(6,101) I,ISQ,ICUBE
 101            FORMAT(' ',I3,I6,I9)
 90             CONTINUE
                STOP END
```

FIGURE 4.2-1. EXAMPLE OF CLEAR FORTRAN CODE

This code is clear; its logic straightforward. The "DO" construct is used effectively to increment "I" from an initial value of "1" to a final value of "50".

It is possible to reduce clarity by using the constructs of a language clumsily. Figure 4.2-2 shows another program that uses an "IF" statement to create the loop, rather than a "DO" construct.

In this case, the program reader is forced to read through the tedious details of initializing the counter, incrementing it, and testing it. In both programs, the results are the same, but the extra complications in the second version

```
30      ISQ = I*I
        ICUBE = I*I*I
        WRITE(6,101) I,ISQ,ICUBE
101     FORMAT(' ',I3,I6,I9)
        I = I+1
        IF(I .LE. 50) GOTO 30
        STOP
        END
```

FIGURE 4.2-2. EXAMPLE OF COMPLICATED FORTRAN CODE

provide additional opportunities for error and reduce the clarity of the program. Clarity should be considered when a programmer initially designs a program. Measuring the degree of clarity found in an existing program is valuable to pinpoint problem areas that may need to be recoded. Since clarity generally opposes complexity, measuring clarity indicates aspects of complexity.

4.3 Completeness

Completeness is the extent to which software fulfills the overall mission requirements (McCall, Richards, and Walters 1977). Thus defined, completeness covers a broad area. It is a measure of how satisfied customers are that the software performs its functions. As a broad definition, completeness could be related to almost all the other quality factors. Hence, a narrower definition is needed.

Boehm et al. (1978) state that a software product is complete to the extent that all of its parts are present and each of its parts is fully developed. A complete program is one in which all external references and input data are supplied. An incomplete program, on the other hand, may contain unresolved external references, missing parameter values, or incomplete data format specifications.

Completeness implies that the software meets the degree of accuracy and precision required for the function for which it is intended. The linker will normally flag unresolved external references. Most compilers will flag incorrect data format specifications and missing parameter values. Input data is normally supplied during testing. Hence, a complete program will be more easily tested, since the tester does not have to generate input data values. Depending on the language and the compiler used, incomplete or incorrect data format specifications may not be detected.

4.4 Complexity

Complexity is one of the most widely addressed quality factors found in the SQM literature. In some articles, complexity is applied loosely to mean the entire group of SQM. In other articles, it is seen as a criterion, which is a level of attribute that falls somewhere between the measurable software metrics and quality factors. If complexity is viewed as the degree of decision-making found in a program (Gilb 1977), this quality factor could be measured using software metrics.

Complexity generally means that which is involved or intricate, or is composed of many parts. Computer scientists, in attempting to understand the software process and products, have related complexity to many attributes, including how error-prone the software is likely to be and how easily the software can be understood. Managers of software development projects must understand the complexity of the project so that factors such as cost, time, and effort can be predicted, and the project can be budgeted properly.

Structural complexity is associated with the software product. Psychological complexity relates to how easily people can understand a program.

Structural complexity relates to data set relationships, data structures, and control flow (McCall, Richards, and Walters 1977). McCabe's Cyclomatic Complexity Metric attempts to measure the structural complexity of a program by counting the decisions in it. Halstead's Length metric attempts to measure this type of complexity by counting the operators and operands in the program.

Psychological complexity is the ease or difficulty with which a person can use a software product (Lennselius, Wohlin, and Vrana 1987). This type of complexity can also be defined as assessing how people perform on programming tasks (Curtis 1979). Halstead's Programming Effort and Difficulty Measures attempt to assess this type of complexity.

Another definition of complexity weaves in the concepts of time. Static metrics measure the product at one particular point in time, while history metrics measure the product and the process taken over time. Static metrics are measures that are based on the physical attributes of a software product. These, in turn, are divided into three basic categories: volume, control organization, and data organization. Volume metrics measure the size of the software, control organization metrics measure the control structures, and data organization metrics measure the use of data and the interactions among data in a program (Basili and Hutchens 1983).

While Basili's definitions are useful because they allow researchers to categorize metrics based on their relationships to structures in the program, the static metric definition actually refers to the software metrics described in chapter 3 of this report. In essence, complexity is thus interpreted as a physical measure. It is important to keep the distinction clear: the quality factor is the subjective quality people consider important for software to possess. A software metric measures some physical attribute of software, like lines of code.

No metric covers all aspects of complexity. It is also important to consider the interrelationships among complexity and other quality factors. Complexity relates to maintainability because the more complex a program is, the harder it is to maintain. Additionally, complex programs tend to have more bugs, leading to more maintenance time later. Complexity conflicts with simplicity, understandability, modifiability, and modularity.

4.5 Conciseness

Conciseness is the ability of a program to satisfy functional requirements using a minimum amount of software (McCall, Richards, and Walters 1977). Software is concise to the extent that no excessive information is present (Boehm et al. 1978).

An excessively commented program can lose the reader's attention. The extra information may obscure the objectives or functions of a program. A program containing circuitous logic may confuse a reader, since the logical flow may be lost in an excessive nesting structure. Complex program structures have been linked with structural complexity. Hence, complexity may be inversely proportional to conciseness. If a program is too concise, however, it may be cryptic and may interfere with a person's ability to understand it. In this unusual case, conciseness is then proportional to complexity. Thus, one of the factors, conciseness or understandability, may be more important to an application and may need to be weighted more heavily.

Conciseness should be evaluated with other needs of the project. If code has already been written and its conciseness is found wanting, there are ways to "tighten" the program to achieve more concise software. One method is to shorten the program by placing repetitive sequences of code in functions, subroutines, or macros. (The macro method is preferred where efficiency [e.g., the execution time] may be an important consideration. Macros save on the overhead associated with calling a function or subroutine.) Another method involves analyzing a program's logic flow and recoding complex modules to create a better logical structure.

Efficiency may be related to conciseness: an increase in conciseness may yield an increase in efficiency. Accuracy also may be related to conciseness. For example, a program that approximates differential equations may be concise, but may not provide a sufficiently powerful approximation algorithm to obtain the required accuracy to graph a solution.

The other extreme is an overly-concise program that is cryptic at best and fragmented at worst. To create an adequate SQM package, the developer should include modularity as another quality factor. In the case of a fragmented program, modularity would conflict with conciseness.

Conciseness and other quality factors should be weighted according to the needs of the application. For example, if a critical design goal is to minimize execution time, a complex but fast algorithm may be needed. However, if minimizing storage space is a higher priority, effort should be made to eliminate unnecessary information and to compress the algorithm.

4.6 Consistency

Consistency can be divided into two types: internal and external. Internal consistency is the degree to which software satisfies specifications (McCall, Richards, and Walters 1977). External consistency is the extent to which a software product contains uniform notation, symbols, and terminology (Boehm et al. 1978). For the code phase, internal consistency is important.

Catching internal consistency errors may depend on the specific constructs of the compiler, linker, and interpreter used on the language. Most references are resolved at link time. Consistency errors might also occur in the way comments are presented. A compiler cannot detect consistency in the way comments, variable names, and variable definitions are implemented. Hence, the programmer must strive to create consistent variable names and definitions.

4.7 Correctness

Correctness measures the extent to which the software design and implementation conform to specifications and standards (Bowen, Wigle, and Tsai 1985). Factors associated with correctness include completeness, consistency, and traceability. Correctness is rated in terms of the number of specifications-related and standards-related errors that occur after the formal release of the specifications, divided by the total number of executable lines of source code.

Correctness is related to reliability. If software is reliable, it executes correctly over a given period of time. Correctness is also an aspect of a program's performance. A tester must understand the program to determine whether it is correct. Hence, the program should be readable and understandable. If the application is large, it should be modular, so the tester can run particular modules with test data.

Efficiency has been defined as the following:

- The measure of the execution behavior of a program (execution speed, storage speed) (McCall, Richards, and Walters 1977).

- The extent to which the software fulfills its purpose without waste of resources (Boehm et al. 1978).

- The ratio of useful work performed to the total energy expended (Gilb 1977).

- The reduction of overall cost - run time, operation, maintenance (Kernighan and Plauger 1974).

Efficiency is considered to be a system quality factor (Bowen, Wigle, and Tsai 1985). It is the extent to which resources are utilized. High system efficiency implies both high software efficiency and usability.

The efficiency of a program depends on the machine on which it runs and on the implementation language. If a program is efficient, it is generally less portable. Consider the FORTRAN code shown in figure 4.8-1. This code is a traditional nonoptimized, nested "DO" loop.

```
              DO 70 I = 1,200
                   DO 80 K = 1,200
                        DO 90 J = 1,200
                        MATC(I,K) = MATC(I,K)+MATA(I,J)*MATB(J,K)
      901               CONTINUE
      801          CONTINUE
      701     CONTINUE
```

FIGURE 4.8-1. EXAMPLE OF NESTED DO LOOP

Figure 4.8-2 shows how the code in the previous figure was changed to make better use of the system. The "DO" loop has now been partitioned into four copies. (Only the first copy is shown in the figure.) The second copy has a "DO" loop range of 51 to 100. The third copy goes from 101 to 150; the fourth from 151 to 200. These copies will run concurrently on four separate processors, thus decreasing the execution time of the code.

```
              DO 701 I1 = 1,50
                   DO 801 K1 = 1,200
                        DO 901 J1 = 1,200
                        MATC(I1,K1) = MATC(I1,K1)+MATA(I1,J1)*MATB(J1,K1)
      901               CONTINUE
      801          CONTINUE
      701     CONTINUE
```

FIGURE 4.8-2. OPTIMIZED PARALLEL FORTRAN CODE

The changed code now depends heavily on a particular compiler (and version), a particular machine, and a specific operating environment. Hence, efficiency for the program is gained at the expense of other quality factors, such as portability and generality.

Efficiency can oppose maintainability. Optimized code, incorporating complex techniques and Assembly language, typically provides problems for the maintainer because Assembly language is machine-dependent. The software may not run correctly, if it runs at all. On the other hand, using high-level code such as Pascal or Ada to increase the maintainability of a system usually increases the overhead, resulting in less efficient operation.

Efficiency is related to storage. Reducing the run time of a program (i.e., increasing the efficiency) will generally increase storage requirements and increase programming effort.

4.9 Expandability

Expandability is the amount of effort required to increase the software's capabilities or performance (Bowen, Wigle, and Tsai 1985). Expanding a program is associated with making changes to a program. A program that is difficult to change will also be difficult to expand. If the code is not very readable and is complex, it will take more effort to change or to expand it.

Storage requirements may control how easily a program can be expanded. Consider a FORTRAN software product that has been designed to run on a SUN workstation. The initial software design allowed room for the compiler, run-time library files, and special files associated with the product. However, when the compiler was rewritten to accommodate new user requests, it exceeded the original space allocated for it. Thus, insufficient storage space remained for old backup files or working files. Additional storage had to be purchased for the workstation before the program could be expanded.

In the case where system storage space is not a problem, the constructs of the program, or the control flow of the program, affect how much effort is required to expand it. A modular application will generally be easier to expand, since individual modules can be enhanced to meet the additional requirements. Programs that contain machine-dependent code are less easily expanded because these dependencies may force a programmer to "work around" that particular section of code.

4.10 Flexibility

Flexibility is defined as the amount of effort required to change the software's mission, functions, or data to satisfy other requirements (Bowen, Wigle, and Tsai 1985). This definition is broad enough to include adaptability and modifiability. It also could include expandability, if the software's capabilities or performance need to be increased to suit a new requirement. By definition, flexibility implies a certain amount of machine-independence, since machine-dependent code is not very adaptable to other applications. Flexibility might also imply a degree of usability and generality, since both these quality factors measure aspects of a program's general usefulness. When flexibility applies to changing the data in the program to satisfy other requirements, this factor is closely linked with integrity, since integrity measures the ability of a program to perform correctly on different sets of input.

Languages have constructs that enhance a program's flexibility. For example, the FORTRAN "READ" statement allows programs to be run with different sets of data. Figure 4.10-1 shows a simple inventory program.

The program reads data values into the variables "NOITEM" (number of items) and "UNITC" (unit cost), multiplies the values to compute a price, and stores the values in "PRICE". The program loops back after printing to read a new set of values. The program does not depend on specific values for "NOITEM" and "UNITC". This flexibility makes the program more general. It also means the program is more usable.

```
10    READ *, NOITEM,UNITC
      PRICE = NOITEM*UNITC
      PRINT *, PRICE
      GOTO 10
      END
```

FIGURE 4.10-1. FORTRAN INVENTORY PROGRAM

Other FORTRAN constructs that add flexibility are the formatted "PRINT" and "READ" statements. These statements allow a programmer to format Input/Output (I/O). Otherwise, if the free-format statements, "READ *" and "PRINT *", are used, the values of variables would be restricted to the print statement of the form, "PRINT *, printlist". The formatted "PRINT" and "READ" statements allow the programmer to specify exact spacing and the type of data to be printed.

A higher order language allows a programmer to express operations using fewer constructs than if he or she had used a low level language. Hence, the program will take less effort to change, and will be more flexible. For example, computations can be expressed more easily in a higher level language, since the programmer does not have to worry about the registers used to store data. In a low level language, the programmer must write additional statements to show how registers are used. In this example, complexity, understandability, and testability are proportional to flexibility.

Another case of improving a program's flexibility involves using "EQUIVALENCE" statements in FORTRAN. If a programmer inadvertently switches the names of variables, he or she can equate the two names, rather than having to retype the program statements. This may save the programmer time. The effect of the "EQUIVALENCE" statements, however, can become complicated, especially if the variables appear in "COMMON" statements. In this case, flexibility is inversely proportional to complexity, understandability, and testability.

4.11 Integrity

Integrity is the measure of the ability of a program to perform correctly on different sets of input (Kreitzberg 1982). In a sense, integrity is a measure of how well a program has been tested. A program may lack integrity if it does not account for all data options, as in the case where a program does not account for possible division by zero.

A program with a high degree of integrity should check input values to determine whether they are within practical bounds. Consider a program that automatically adjusts the cabin pressure in an airplane to the expected atmospheric pressure. The program uses the altimeter reading for its input. The pressure is adjusted according to the sensor reading of the altimeter. To ensure that the cabin pressure is within comfort range, the program periodically range checks the altimeter readings before it adjusts the cabin pressure.

Assume the commercial airplane never flies above 40,000 feet. The program checks to see if the sensor range is within bounds:

```
360     IF ALTITUDE > 40000 GOTO METERFAIL
370     LET CABIN = (CONSTANT * ALTITUDE)^2
```

The program could also check some lower limit, thus providing a check of values that are acceptable within some range. If the input data for this program is within an acceptable range, the pressure is adjusted. If the program did not account for all possible conditions, a serious problem could occur that would not be handled by the program. If the sensor malfunctions and provides a negative input number, the program must be able to handle this situation. This example illustrates the importance of measuring the integrity of a program.

4.12 Maintainability

Maintainability is a process that includes the following definitions:

- It is the measure of the effort and time required to fix bugs in the program (McCall, Richards, and Walters 1977).

- During a maintenance fix, it is the probability that a failed system will be restored to operable condition within a specified time (Gilb 1977).

- It is the ease or difficulty with which software can be updated to satisfy new requirements (Boehm et al. 1978).

- It is a measure of how easily software can be changed because of bugs encountered during operation, user requirements that were not satisfied, changing requirements, and upgrading or "obsoleting" a system (McCall, Richards, and Walters 1977). In this view, maintenance encompasses not only the time needed to fix errors, but also time needed to enhance the system's operation (Kafura 1987).

Maintainability is a quality factor that addresses how well the software can be maintained after it has been developed. Implied in the definitions is the question of how well the software adapts to changing requirements or to enhancements made to the product. In such terms, maintenance may be closely aligned with flexibility and expandability.

Software maintenance is a critical quality factor because studies indicate programmers spend more than half their time on maintenance. Software maintenance consumes as much as two-thirds of the total life-cycle resources.

Complexity hinders maintainability. The more complex a system is, the more difficult it is to understand, and therefore to maintain. Complexity is related to maintaining software in another way. Complex programs typically contain more bugs. These bugs can remain hidden until late in the life cycle. At that stage in the life cycle, budgeting constraints may force companies to market the product with some of these bugs. Hence, the quality of the software may be compromised.

The relationship between maintainability and complexity also implies that not only must the user understand the program, but he or she must be able to modify and test it. While researchers (Boehm et al. 1978) point out that this quality factor does not depend on the program's current reliability, efficiency, or portability, the quality factors may all be related. Efficiency can oppose maintainability. For example, optimized code may be a problem to maintain because the code is then very machine-dependent. If the code is not very portable, it could be very difficult, if not impossible, to maintain it on some other system. Additionally, maintainability could be proportional to reliability. If code is not very reliable, it will cause problems during a given maintenance period; the two quality factors are intertwined.

Maintainability is a quality factor derived from consistency, simplicity, conciseness, modularity, and self-descriptiveness (Browne and Shaw 1981). Yet there are cases where modularity and conciseness conflict. The code could be modular at the expense of conciseness. (One main program might have been more concise than separate modules). Self-descriptiveness can also conflict with conciseness. All these quality factors are considered to be components of a higher quality factor, maintainability.

Maintainability is also viewed as the ease with which software failures can be located and fixed (Bowen, Wigle, and Tsai 1985). The rating formula using this definition considers the average number of labor-days to locate and fix an error within a specified time. If this definition is used, reliability should also be considered as one of the application's quality factors.

The amount of resources a program uses is also related to maintenance. A program that uses all available resources (memory, disk, tape, etc.) will be less maintainable than one that does not. In this view, maintainability is closely related to portability and adaptability. It may also be related to generality and usability: the more general a program tends to be, the more usable it is, and the easier it is to maintain.

4.13 Modifiability

Modifiability measures the cost of changing or extending a program (McCall, Richards, and Walters 1977). It is the extent to which changes can be incorporated into an existing program (Boehm et al. 1978).

How easily code can be modified depends on other factors such as complexity, control structures, and self-descriptiveness. A program that uses cryptic names will not be as easily modified as one that uses meaningful ones. Even if the same programmer must change the code, he or she may have problems changing it if time has elapsed since the program was created and the programmer forgets the special mnemonics. Consider the COBOL code illustrated in figure 4.13-1.

```
TEST-STK-CD.
        IF STK-CAT = "A2", GO TO BUMP-C1.
        IF STK-CAT = "A3", GO TO BUMP-C1.
        IF STK-CAT = "A4", GO TO BUMP-C1.
        IF STK-CAT = "A1", GO TO BUMP-C2.
        IF STK-CAT = "B2", GO TO BUMP-C3.
        IF STK-CAT = "B3", GO TO BUMP-C3.
        IF STK-CAT = "B4", GO TO BUMP-C3.
        IF STK-CAT = "B1", NEXT SENTENCE, ELSE
        GO TO J1.
        COMPUTE C = 6, GO TO TEST-ROUTE.
        BUMP-C1.
        IF ROUTE = 2, GO TO BUMP-C1A.
```

FIGURE 4.13-1. COBOL CODE EXAMPLE

This code is not easily modified (especially by a different programmer) for several reasons. First, the labels for jumping to routines are not very helpful; they do not reveal much about the program's functions. Note that the condition in the routine BUMP-C1 is equally terse, "IF ROUTE = 2, GO TO BUMP-C1A". Secondly, the repetitive "IF" tests create a complicated control flow, thus impairing one's ability to read and to understand the program. Because the program must be understood before it can be modified, complexity increases the amount of time needed to change the program.

Figure 4.13-2 shows another COBOL example. In this case, some of the variable names are more meaningful, like "UNIT-PRICE" and "COMP-PRICE". Hence, the program is more readable. The flow of logic is still complex. Also note that this program segment mixes "GO TO" constructs and "PERFORM" constructs. The "PERFORM" is like a call to a subroutine. The use of both constructs increases the complexity of the control flow, since the programmer must interpret the meaning of two types of constructs, rather than one. Modifying the code will take more time and effort.

```
IF UNIT-PRICE > 9.999
           COMPUTE C = 1, GO TO TEST-ROUTE.
COMPUTE C = 2, GO TO TEST-ROUTE.
BUMP-C1A
IF TC = 20, IF T-UNIT-PRICE = O,
           PERFORM COMP-PRICE.
IF T-UNIT-PRICE > 9.999
           COMPUTE C = 1, GO TO TEST-ROUTE.
```

FIGURE 4.13-2. COBOL EXAMPLE SHOWING CONDITIONALS

Maintainability is also an issue, since the ability to maintain a program depends on the complexity and on how easily a program can be understood. Depending on the application, both types of construct might be necessary. Using "PERFORM" statements might be a positive attribute if a more modular program is needed. From that standpoint, maintainability might be enhanced.

Generality is a subset of modifiability, which measures how easily a program can be changed to perform a slightly different task.

4.14 Modularity

In the computer industry, applications are becoming increasingly complex and diverse. The high-powered technology that continues to spawn faster and more efficient computers has led to larger, more complex software applications. This complexity must be reduced to manageable chunks so it can be understood.

Modular programming techniques have been developed to cope with this complexity. Functions are kept in libraries for general use, and are only called when they are needed in the program. Subroutines are rewritten to exist as separate modules. In its broader sense, program modularity is expressed in terms of program subunits that are developed independently and then interfaced (Baker 1979). Modularity has also been defined as the extent to which a program is organized around its data and control flow structures (Kernighan 1974).

Investigators have established correlations between modularity and maintenance. Modularity is also associated with flexibility, testability, portability, reusability, and interoperability.

4.15 Performance

Performance is perhaps the broadest quality factor. It encompasses several of the other quality factors, and is concerned with how well a software product functions. Performance asks the following questions (Bowen, Wigle, and Tsai 1985):

- How well does the software utilize a resource?

- How secure is the product?

- What confidence can be placed in what the product does?

- How well will the product perform under adverse conditions?

- How easy is the product to use?

This factor is concerned with how well the software has met certain performance goals. These performance goals, in turn, are measured by SQM. Hence, in the hierarchy of quality factors, performance is placed at the top, the most encompassing level. In fact, RADC lists performance, design, and adaptation as the three overriding concerns, of which the other quality factors are sub-categories.

Before SQM, performance was typically linked to how quickly a program executed. The engineer was only concerned with whether the software met some minimum requirement. The degree to which the software met the requirement was not an issue. When Software Quality Assurance (SQA) became an important aspect of delivering final software, metrics were created to measure the degree to which software met various quality standards. The measurement of how well a product performed became important.

For example, real-time applications must process user data quickly. Additionally, the programs must be able to accurately interpret this information. Thus, the overall quality of the code must be "good" in terms of processing time and correctness.

4.16 Portability

Portability measures how easily a software product will run on a computer configuration other than the current one (Boehm et al. 1978).

To develop portable software, the product should use standard library functions and subroutines that are "universally" available, that have similar argument lists, and that provide a similar accuracy. The program should avoid calling operating system functions. For example, job control, data set definitions, and I/O operations are all highly system-dependent (Boehm et al. 1978).

Where operating system dependencies cannot be avoided, a measure of portability can rely on the software or on the instructions provided to the customer. For example, in a sophisticated parallel processing product, the user needs to know in what order libraries were linked. The documentation for the product should cover this area by providing the user with an example of how to build a link command file to ensure that user-created object libraries were searched before the standard run-time libraries.

Portability may oppose efficiency. For example, software may run better on its initial system because the algorithms take advantage of the system's idiosyncrasies or default set-up.

Hardware dependencies are a key consideration in transporting software. Most programs are portable at the source level, but are usually not portable at the object code and executable level. At the source level, hardware and process differences have to be considered. For example, if a program is ported from a mainframe to a personal computer (PC), the computer logic and the functions may be the same, however, device names will probably be different. Additionally, the methods for calling the devices and for compiling the program will vary.

Software depends on the hardware configuration and may not be readily transferred. Consider a program that contains print functions or other device-dependent routines. If the device is not connected to the correct port or is not connected at all, the program will suspend execution without knowing where to send the data. Auxiliary devices usually incorporate electronic signals, such as RS-232 interface pin voltages to indicate a device is present. The software recognizes the electronic signal and acknowledges it. The computer can then send data to those devices. The software must take into account the hardware and its configuration in the system. These issues are tied in with portability considerations.

A program is more portable if it is written in a well-known language. In cases where an implementation language is not readily known, the software's portability would be limited to those systems that accept the language. The language might have to be converted into one for which a compiler or interpreter is available. In this case, the accuracy of the product might then be compromised, depending on the ability of the conversion program to translate the constructs of the unknown language to the accepted language.

4.17 Reliability

Reliability, like maintainability, includes several definitions. The two main groups of definitions generally are hardware- or software-oriented.

In the first group of definitions, reliability is simply a measure of the number of errors encountered in a program (McCall, Richards, and Walters 1977). This definition would include the "correctness" of a program.

A broader definition relates reliability to the function the software is designed to perform. It is the extent to which a program can be expected to perform its intended functions satisfactorily (Thayer 1976). Another critic adds a qualifier: the software must perform its functions correctly in spite of computer component failures (McCall, Richards, and Walters 1977). Reliability also implies the software must work over a given period of time.

These ideas place the focus of reliability on the overall functions of the software. In this view, reliability is closely related to correctness and to the performance of the software product. For example, a program that monitors air traffic patterns for a particular airport needs to be reliable so that air traffic controllers can use the information to schedule flights.

Hardware reliability has a certain physical, tangible element. Hardware components are designed with the knowledge that they will wear out after a given length of time. Manufacturers test components for wear and devise reliability time charts based on these tests. Under stated conditions, a component or product will thus have a particular life-span (mean-time between failures).

Proponents for lumping software reliability into this category maintain that software reliability also can be predicted based on the number of bugs found within a particular time frame. Based on this evidence, a test for reliability is made. It is assumed the software, unlike hardware, will not degrade further after a given time.

In these terms, program reliability is the probability that a given (software) program operates for a certain time period, without a logical error, on the machine for which it was designed (Gilb 1977). More specifically, software reliability is the probability that a software fault does not occur during a specified time interval (or specified number of software operational cycles) which causes the results to deviate from specified tolerances (Thayer 1976).

Most researchers believe software reliability is a subset of system reliability, and view reliability as a composite quality factor derived from error tolerance, consistency, accuracy, and simplicity (Browne and Shaw 1981). Others conclude that reliability becomes a question of correctness, confidence, accuracy, and precision rather than the time to the next failure (Fisher and Light 1979).

In this view, software reliability is the extent to which the system will perform without any failure (Bowen, Wigle, and Tsai 1985). High system reliability implies high software reliability, correctness, and integrity. Both software reliability and correctness contribute to the system's performing its intended functions.

According to Boehm et al. (1978), reliability is first the measure of whether the product meets its requirements. Will the program produce answers with the needed accuracy? The second part of the definition applies when the software is fielded. Does the program still operate correctly outside the laboratory setting? For code that is being tested for certification, only the first part of Boehm's definition applies. Once the code has been integrated into the avionic equipment, the testing is assumed complete, and only periodic maintenance tests are done.

Reliability is related to correctness, integrity, and performance.

4.18 Reusability

Reusability is the extent to which a system can be applied to other environments (Gilb 1977). A second definition states it is the effort needed to convert software for another use (Bowen, Wigle, and Tsai 1985). Consider a math routine that is found in a general-purpose math library. Reusability addresses how easily that routine can be customized for another application.

Reusability is a major issue today as more and more code designed for a specific purpose must support other projects. It is far easier to use existing code than to create code from scratch. Or is it? If the code is usable and readable, it will have a certain degree of reusability. Modular programs tend to be more reusable than other programs because modules may clearly be separated by their functions. Hence, a function can be identified easily, and can be copied and modified to fit other application needs. Reusability is related to portability and to generality, since portable code can be used on different machines, and general code can be changed quickly to suit a different purpose.

4.19 Simplicity

Simplicity is the ease with which functions can be implemented and comprehended (McCall, Richards, and Walters 1977). Software possesses simplicity to the extent that it uses data and control structures for organizing the program, and uses easily understood constructs.

Simplicity is related to conciseness to the degree that the concise program is easy to understand. Understandability is proportional to simplicity. A program that expresses the content of an algorithm succinctly and clearly has simplicity. It usually has a straightforward control flow structure. Modularity is related to simplicity. Programs that are modular allow the reader to understand the application one module at a time.

For example, consider a program that collects communication, weather, and maintenance data. This program evaluates the data, processes it, and disseminates it to users, such as air traffic controllers and maintenance personnel. The volume of data and the complexity of the systems make the program too complex to create and to maintain as one module. Instead, several modules with specific functions are created to handle this information. While the total application is complex, the individual modules are built with simplicity in mind.

Simplicity is the opposite of complexity.

4.20 Testability

Testability is the extent to which software facilitates the establishment of acceptance criteria and supports evaluation of its performance (Boehm et al. 1978). "Testability is the measure of our ability to test software" (McCall, Richards, and Walters 1977).

Does the source code support testing? If the code is divided into functional modules, the tester can devise template tests for each module. In this way, the tester can begin to test easier modules first, if he or she is unsure of the exact application of the program. Thus, the tester gradually learns the functions of the entire application. If the application has been partitioned into seemingly arbitrary modules, the tester may have a more difficult time creating tests for each module, since the meaning of each is obscured.

This quality factor related to one's ability to understand a program. It also relates to the complexity of the application. Additionally, as an application grows in complexity, it may be harder to maintain. Testability, then, relates to many complementary and to some discordant quality factors.

A program that is not very testable may be one that hides a straightforward algorithm in complex programming constructs. For example, a program that multiplies a given matrix by another might mask the matrix constructs and process so that the function is not clear. In this case, the tester has to test the program without clearly understanding its function. The metric scores for understandability and maintainability on this program would be lower, while its complexity scores would be higher.

4.21 Understandability

Understandability is the ease with which a program can be understood (McCall, Richards, and Walters 1977). It can also be viewed as the extent to which the purpose of the product is clear (Boehm et al. 1978).

To be understandable, a program should be simply written, be free of jargon, have clearly defined variables, and be self-descriptive. If the program contains sufficient comments and has consistent and meaningful names, it will be more understandable.

Understandability is related to all the quality factors, except reliability and performance.

4.22 Usability

Usability measures the effort required to train a person to use the software (Bowen, Wigle, and Tsai 1985).

According to Boehm et al. (1978), usability consists of two aspects: one is product-related, the other user-oriented. In the context of a product, can the product be used elsewhere? For example, consider a typical math library. This library could be developed to support a symbolic algebra program. Can this same library be used to support another product, say an electronic scratch pad? In this example, usability is related to portability.

Usability is also related to adaptability. For example, consider a set of run-time library routines that have been developed for a parallel processing product. The product is originally targeted for FORTRAN users. Can the same run-time library routines be used in an enhanced product that includes C language users? In this case, adaptability is a subset of usability.

The second aspect of usability measures how easy or convenient it is to use a program. For example, are functions and procedures well-commented? Is the logic of the program straightforward? Other quality factors, such as clarity and complexity, are analogous to usability; these quality factors should also be considered in the SQM analysis to get a more meaningful result.

This second aspect is important in rating source code in terms of human factors. Because people enter information into databases, interpret results, and maintain code, this quality factor has gained importance in industry.

Good examples of usable code include code that allows free-form input. For example, a program that accepts integer and real values for a particular input (where the function is not impaired by the accuracy of the input) is considered more usable, than one that does not.

A notorious example of less usable code involves rigidly-designed input screens. For example, consider a program that allows a user to access the next screen by using the TAB key. If the user tries an ARROW key, the program produces an unexpected result. This rigidity has an effect on maintainability, and on the number of errors that can be entered by the user. Usability also relates to how well the software can be understood. In this case, the metric for understandability is likely to correlate directly with this aspect of usability.

Depending on the application, the program may have to restrict input data in some ways. If these restrictions are adequately defined (e.g., screen messages) the program may still be usable.

To rate high in this quality factor, the output data should be clear and understandable. Additionally, a program which gives the user English language error messages will be more usable than one that gives messages similar to the following:

```
ERROR: 3485948.log
```

which, at the most, implies that the user should check the indicated log file to analyze the problem.

English messages will not always create a more understandable program. Consider the following UNIX$^{\text{TM}}$ operating system message:

```
too many processes
```

This cryptic message tells the user little. The user has to know something about how the UNIX$^{\text{TM}}$ operating system has been designed to make any sense of it. The message indicates that no new processes can be initiated by the user until the spooler queue is ready to accept more requests. Because the message is so cryptic, the user may not realize that some requests have been canceled. He or she may try to send more print requests, thus further bogging down the system. A better message would be as follows:

> The queue is full and cannot accept more requests. Please enter the word 'status' to receive more information on the processes that are running. Enter 'help status' for information on what status reveals.

Usability is also related to generality, a measure of how easily a program may be modified to perform a slightly different but closely-related task.

5. SOFTWARE QUALITY METRICS

SQM are composed of software metrics and SQFs. Most of the SQM presented in this chapter have been validated and are discussed in light of the correlational studies that have been conducted on each SQM. Their practical applications in industry and government are also discussed.

The three major families of software metrics (Halstead's, McCabe's, and RADC's), upon which most of the prominent SQM are based, are presented first. These are considered major systems because they satisfy the criteria presented in chapter 3. Additionally, long-term empirical studies and their continued use support their validity.

Of the first three presented, McCabe's is the most-narrowly focused SQM. However, his software metrics have been used extensively in industry and have been correlated with several quality factors.

The most comprehensive, integrated approach is RADC's metric methodology. RADC is concerned with the overall quality of a product. Hence, its metrics are comprehensive, covering the entire software life-cycle. Although this report focuses on the code phase, RADC developed several groups of metrics that apply to software development (from the design phase through integration testing).

Not all the SQM presented have been empirically validated, or pass all of the other criteria listed in chapter 3. Three additional families of software metrics include promising SQM, but the software metrics and their relationships to quality factors require further substantiation. This group is discussed next, and includes Albrecht's, Ejiogu's, and Henry and Kafura's families of software metrics. Within this group, Albrecht's and Henry and Kafura's SQM have been more widely applied, Ejiogu's less. For the whole group, applications in the field have produced some qualitative results. For some of the software metrics in this group, experimental validation consists of using metric tools that provide a relative assessment of several quality attributes of code.

This report identified practical SQM that consist of experimentally-verified relationships between some of the software metrics discussed in chapter 3 and the quality factors discussed in chapter 4. In this chapter, these relationships are presented in tables in the beginning of each family of software metrics. Each relationship is then discussed along with its corroborating evidence.

5.1 SQM Based on Halstead's Software Metrics

Perhaps of all SQM, Halstead's family of metrics has been tested the most extensively. Table 5.1-1 lists the experimentally validated SQM.

TABLE 5.1-1. SQM BASED ON HALSTEAD'S FAMILY OF SOFTWARE METRICS

Software Metric	Quality Factor
Implementation Length, N	Maintainability Number of Bugs Modularity Performance Reliability
Volume, V	Complexity Maintainability Number of Bugs Reliability Simplicity
Potential Volume, V^*	Conciseness Efficiency
Program Level, L	Complexity Simplicity
Intelligence Content, I	Conciseness Efficiency
Programming Effort, E	Clarity Complexity Maintainability Modifiability Modularity Number of Bugs Performance Reliability Simplicity Understandability
Number of Bugs, $\hat{B}$	Maintainability Number of Bugs Testability

5.1.1 Implementation Length

Munson and Khoshgoftaar (1990) correlated each of the operator and operand counts with complexity and modularity as shown in table 5.1-2. The table indicates that all counts (except for the number of unique operands) were highly correlated with complexity. Conversely, only the number of unique operands count was highly correlated with modularity.

TABLE 5.1-2. QUALITY FACTOR CORRELATIONS

Count	Complexity	Modularity
η_1	0.802	0.301
N_1	0.900	0.346
η_2	0.343	0.860
N_2	0.865	0.425

Curtis et al. (1979) found length to be correlated with performance. For subroutines, the correlation between length and performance was 67 percent. For programs, the correlation between the two was 52 percent. This study showed that Halstead's length measure loses effectiveness as the size of the code increases. However, the study also showed that Halstead's Effort Equation became more highly correlated with performance as the size of the program increased.

5.1.2 Volume

In an experiment conducted by Curtis (1980), nine programmers created three simple programs. Halstead's Volume metric was the best predictor of the time required to develop and to run the programs successfully.

5.1.3 Potential Volume

Basili, Selby, Jr., and Phillips (1983) concluded that the Potential Volume, which is a minimum representation of an algorithm in a specific language, was correlated to the actual Volume, V, by 67 percent. They concluded that a good relationship exists between a program's Volume and its specification (the Potential Volume). Unlike the Volume metric, however, the Potential Volume is only related to the efficiency of the algorithm, or its conciseness.

5.1.4 Program Level

Because the Program Level is a ratio of the Potential Volume to the size of the implementation (its Volume), this software metric is validated indirectly by the validation of its component metrics, as discussed previously.

5.1.5 Intelligence Content

Halstead has validated this software metric by publishing the results of two experiments that support the invariance of the Intelligence Content. In the first experiment, Euclid's algorithm, which computes the greatest common denominator, was implemented in several languages, including Algol, PL/1, FORTRAN, and Assembly. The results indicated that "I" remained constant, within 10 percent of its average (Halstead 1977). The second experiment used a different algorithm. This new algorithm was implemented in Algol 58, FORTRAN, COBOL, BASIC, SNOBOL, APL, and PL/1. Again, the Intelligence Content remained within 10 percent of its average (Halstead 1977).

These results show that the Intelligence Content does measure the content of a program, thus its complexity. However, independent experimentation (Shen, Conte, and Dunsmore 1983) indicates that the measure does not stay within 10 percent of its average. But this study has some limitations: it used students' programs, rather than those written by professional, experienced programmers. More experiments should be conducted to substantiate this metric independently. From a qualitative standpoint, the Intelligence Content does relate to the efficiency and conciseness of a program because it is an estimate of the Potential Volume (Halstead 1977).

5.1.6 Programming Effort

Curtis, Sheppard, and Milliman (1979) showed Halstead's Effort Equation was related to the difficulty programmers experienced in understanding the software. For subroutines, there was a 66 percent correlation between Effort and Understandability. For programs, the correlation was 75 percent.

Most of the early research on Effort as a measure of program complexity was empirical and provided evidence that Effort is related to programming time, the number of bugs encountered during program development, and program clarity. A more analytical evaluation of Effort demonstrated that the measure's behavior agrees with modularity principles (Baker and Zweben 1979).

Harrison[2] (1988) used the Halstead Effort Equation to determine the correlation between effort and number of bugs found in a module. This information was used to determine the amount of maintenance time that should be allocated for each module. For the study, a module was defined as a component of software. Typically, a module was composed of at least a few subprograms. He found the effort was correlated to the number of bugs found in a program.

5.1.7 Number of Bugs

Researchers have proposed a relationship between software errors, Effort, and Volume:

$$\text{Errors} = L \times (E/3200) = V/3200$$

If this relationship could be experimentally proven, the information could be used to estimate the amount of testing that will be needed for programs of different Volumes.

Bailey and Dingee (1981) collected error information for 27 programs written for a real-time switching system. The programs were assumed to possess the same relative quality when they were submitted for testing. The data showed that the relationship between errors and volume was not linear. Hence, they concluded Halstead's metrics could not be used to estimate errors in the switching system software.

In maintaining software, industry generates software change reports identifying problems with fielded software. While the formats for these reports may differ, the purpose is usually the same: to identify existing bugs, and to allow the developer to fix them. Halstead's number of bugs estimate continues to be applied to various software projects.

5.1.8 Field Support for Halstead's SQM

Halstead's family of metrics continues to be validated, as they continue to be applied. Companies use the following measures and counts in their software tools:

- The total number of operator occurrences
- The total number of operand occurrences
- Vocabulary
- Implementation Length
- Volume
- Program Level
- Program Difficulty
- Intelligence Content
- Programming Effort
- Programming Time
- Language Level
- Number of Bugs

The greatest support for his measures stems from their continued use in industry. The measures provide SQM when they are correlated with the following quality factors:

- Complexity (structural and psychological)
- Conciseness
- Correctness
- Expandability

- Efficiency
- Maintainability
- Modularity
- Readability
- Simplicity
- Testability
- Understandability

5.2 SQM Based on McCabe's Cyclomatic Complexity Metric

McCabe's Cyclomatic Complexity Measure has been correlated with several quality factors. These relationships are listed in table 5.2-1.

TABLE 5.2-1. SQM BASED ON MCCABE'S CYCLOMATIC COMPLEXITY METRIC

Software Metric	Quality Factor
Cyclomatic Complexity, v(G)	Complexity Maintainability Modularity Modifiability Number of Bugs Reliability Simplicity Testability Understandability
Essential Complexity, ev(G)	Complexity Conciseness Efficiency Simplicity

Kitchenham (1981) conducted a study that correlated the Cyclomatic Complexity Metric to the number of paths in a program. While her results indicated there is a correlation between the size of a program and the number of paths, she concluded that this measure did not gauge how to rank programs of various complexities. Hence, she felt it was a narrow measure that did correlate to complexity, but that only measured one aspect of this quality factor.

Weyuker (1988) concluded similarly that the metric measures one aspect of complexity because it only correlates the size of a program to complexity. McCabe's Cyclomatic Complexity Metric is similar to the Lines of Code Metric because the SQM is based on complexity as it correlates to size. Neither metric considers how constructs are used within the program.

Conversely, the Information Flow Metric measures complexity from the context. The measure depends on how the statements are ordered and on how the modules interact. Effective complexity measures should incorporate both types of metrics.

Harrison and Magel (1986) contend that McCabe's measure falls short of providing a complete measure of complexity. The measure does not account for the complexity found within sequential blocks of code. It simply counts the number of conditional statements in a program to assess complexity.

However, Curtis et al. (1979) showed that McCabe's Cyclomatic Complexity Metric is related to performance. Eighty-one FORTRAN programs, having different versions of control flow, were developed. The programs ranged from 25 to 225 lines of code. Bugs were inserted into each program.

Fifty-four programmers from six locations participated in the experiment. Thirty were civilian industrial employees; the others worked for the U.S. Armed Forces. The participants averaged 6.6 years of programming experience in FORTRAN, ranging from 0.5 year to 25 years.

The results from the experiment demonstrated that McCabe's Cyclomatic Complexity Measure was correlated with the complexity of a program. The study showed that as a program grows in size, its psychological complexity (which the experimenters related to understandability) is better assessed by measuring constructs other than lines of code. Specifically, for subroutines, the correlation was 67 percent between McCabe's metric and complexity. For programs, the correlation was 65 percent for the same SQM.

This experiment also indicated McCabe's Cyclomatic Complexity Metric is correlated with the effort required to locate bugs. The results of the metric provide information to programmers about the complexity of code, since the more complex the code, the more difficult it is to locate errors. Since fixing bugs is part of maintenance, the metric results can be used to help managers plan the resources and effort needed to maintain code.

Munson and Khoshgoftaar (1990) showed that the Cyclomatic Complexity Metric was highly correlated with the structural complexity of a program when volume and control structure were used as criteria for complexity. The study showed an 89.2 percent correlation. The same study also indicated that v(G) was not very well correlated with modularity; the correlation was 12.7 percent.

Perhaps the most useful SQM stemming from the Cyclomatic Complexity Metric is the v(G)-testability SQM. Industry continues to support this SQM.

Several companies have developed tools that incorporate McCabe's Complexity Metric. The metric was paired with the quality factors listed in table 5.2-1. As with Halstead's measures, the major support for McCabe's Cyclomatic Complexity measure is the fact that it is still one of the most widely-applied SQM measures. Most tools that incorporate Halstead's measures also incorporate McCabe's. By this hybrid approach, companies hope to be able to provide metrical measurements that encompass more of the attributes of particular quality factors, such as complexity and maintainability. In this way, the tools provide better SQM.

5.3 RADC's SQM Methodology

RADC's family of SQM is the most integrated set of software metrics and quality factors. Their approach has been to apply metrics at every stage of software development to ensure that a particular level of quality can be measured and maintained throughout the project. These tables are copied from Volume II of the Specification of Software Quality Attributes (Bowen, Wigle, and Tsai 1985).

Table 5.3-1 shows the quality factor definitions RADC has developed. This table also shows the rating formula for each factor, and the higher-level factor (i.e., the acquisition concern) to which each quality factor is related.

Table 5.3-2 shows the fundamental relationships among quality factor criteria, and acquisition concerns. Criteria fall into one of four acquisition concerns: performance, design, adaptation, or general. Shared criteria are those that are attributes of more than one quality factor (Bowen, Wigle, and Tsai 1985). For example, simplicity is a criterion for six of the 13 quality factors; consistency is shared by two quality factors.

The factors listed in the columns correlate with several criteria. Reusability, for example, correlates with almost all the criteria in the adaptation concern, and all the criteria in the general acquisition concern.

Table 5.3-3 shows the additional beneficial and adverse relationships that occur among criteria and quality factors. Criteria that are basic attributes of a factor are identified with an x. Criteria that are in a positive or cooperative relationship with a quality factor are denoted by triangles. Criteria that conflict with quality factors are represented with inverted triangles. An empty space indicates no relationship has been established between a criterion and a quality factor.

For example, completeness is a criterion of correctness, and is shown as having a cooperative relationship with maintainability and reusability, although it is not an essential characteristic of these latter two. Thus, the effect of completeness should be considered when scores are calculated for maintainability and reusability.

Factor scores (introduced in section 3.4) can be calculated to give SQM relationships for several quality factors. The developer can then plot the scores as data points. Graphing the data points will allow the developer to see quality trends throughout the software development process. From a code level, scores indicate to what degree specific qualities (certain factors) are present. This degree may or may not meet the initial design expectations.

TABLE 5.3-1. SOFTWARE QUALITY FACTOR DEFINITIONS AND RATING FORMULAS

Acquisition Concern	Quality Factor	Definition	Rating Formula
Performance	Efficiency	Relative extent to which a resource is utilized (i.e., storage space, processing time, communication time)	$1-\frac{\text{Actual Resource Utilization}}{\text{Allocated Resource Utilization}}$
	Integrity	Extent to which the software will perform without failures due to unauthorized access to the code or data within a specified time period	$1-\frac{\text{Errors}}{\text{Lines of Code}}$
	Reliability	Extent to which the software will perform without any failures within a specified time period	$1-\frac{\text{Errors}}{\text{Lines of Code}}$
	Survivability	Extent to which the software will perform and support critical functions without failures within a specified time period when a portion of the system is inoperable	$1-\frac{\text{Errors}}{\text{Lines of Code}}$
	Usability	Relative effort for using software (training and operation) (e.g., familiarization input preparation, execution output interpretation)	$1-\frac{\text{Labor-Days to Use}}{\text{Labor-Years to Develop}}$
Design	Correctness	Extent to which the software conforms to its specifications and standards	$1-\frac{\text{Errors}}{\text{Lines of Code}}$
	Maintainability	Ease of effort for locating and fixing a software failure within a specified time period	1-0.1 (Average labor-days to fix)
	Verifiability	Relative effort to verify the specified software operation and performance	$1-\frac{\text{Effort to Verify}}{\text{Effort to Develop}}$
Adaptation	Expandability	Relative effort to increase the software capability or performance by enhancing current functions or by adding new functions or data	$1-\frac{\text{Effort to Expand}}{\text{Effort to Develop}}$
	Flexibility	Ease of effort for changing the software missions, functions, or data to satisfy other requirements	1-0.05 (Avg.labor-days to change)
	Interoperability	Relative effort to couple the software of one system to the software of another system	$1-\frac{\text{Effort to Couple}}{\text{Effort to Develop}}$
	Portability	Relative effort to transport the software for use in another environment (hardware, configuration, and/or software system environment)	$1-\frac{\text{Effort to Transport}}{\text{Effort to Develop}}$
	Reusability	Relative effort to convert a software component for use in another application	$1-\frac{\text{Effort to Convert}}{\text{Effort to Develop}}$

NOTE: The rating value range is from 0 to 1. If the value is less than 0, the rating value is assigned to 0.

TABLE 5.3-2. SOFTWARE QUALITY FACTORS AND CRITERIA

Acquisition Concern			Performance					Design			Adaptation				
ACQUISITION CONCERN	Factor/Acronym		EFFICIENCY	INTEGRITY	RELIABILITY	SURVIVABILITY	USABILITY	CORRECTNESS	MAINTAINABILITY	VERIFIABILITY	EXPANDABILITY	FLEXIBILITY	INTEROPERABILITY	PORTABILITY	REUSABILITY
	Criterion/Acronym		EF	IG	RL	SV	US	CR	MA	VE	EX	FX	IP	PO	RU
PERFORMANCE	Accuracy	AC			X										
	Anomaly Management	AM			X	X									
	Autonomy	AU				X									
	Distributedness	DI				X									
	Effectiveness-Communication	EC	X												
	Effectiveness-Processing	EP	X												
	Effectiveness-Storage	ES	X												
	Operability	OP					X								
	Reconfigurability	RE				X									
	System Accessibility	SS		X											
	Training	TN					X								
DESIGN	Completeness	CP						X							
	Consistency	CS						X	X						
	Traceability	TC						X							
	Visibility	VS							X	X					
ADAPTATION	Application Independence	AP													X
	Augmentability	AT									X				
	Commonality	CL											X		
	Document Accessibility	DO													X
	Functional Overlap	FO											X		
	Functional Scope	FS													X
	Generality	GE									X	X			X
	Independence	ID											X	X	X
	System Clarity	ST													X
	System Compatibility	SY											X		
	Virtuality	VR									X				
GENERAL	Modularity	MO				X			X	X	X	X	X	X	X
	Self-Descriptiveness	SD							X	X	X	X		X	X
	Simplicity	SI			X				X	X	X	X			X

TABLE 5.3-3. EFFECTS OF CRITERIA ON SOFTWARE QUALITY FACTORS

Acquisition Concern			Performance					Design			Adaptation				
ACQUISITION CONCERN	Factor/Acronym Criterion/Acronym		EFFICIENCY EF	INTEGRITY IG	RELIABILITY RL	SURVIVABILITY SV	USABILITY US	CORRECTNESS CR	MAINTAINABILITY MA	VERIFIABILITY VE	EXPANDABILITY EX	FLEXIBILITY FX	INTEROPERABILITY IP	PORTABILITY PO	REUSABILITY RU
PERFORMANCE	Accuracy	AC	▼		X										
	Anomaly Management	AM	▼		X	X	▲								
	Autonomy	AU				X									
	Distributedness	DI		▼		X									
	Effectiveness-Communication	EC	X						▼	▼				▼	
	Effectiveness-Processing	EP	X						▼	▼				▼	
	Effectiveness-Storage	ES	X						▼	▼				▼	
	Operability	OP	▼				X		▲	▲					
	Reconfigurability	RE	▼			X			▲		▼	▼		▼	▼
	System Accessibility	SS	▼	X											
	Training	TN					X								
DESIGN	Completeness	CP						X	▲						▲
	Consistency	CS						X	X	▲	▲	▲			▲
	Traceability	TC						X	▲	▲	▲	▲			▲
	Visibility	VS							X	X					
ADAPTATION	Application Independence	AP												▲	X
	Augmentability	AT									X				
	Commonality	CL	▼						▲				X		
	Document Accessibility	DO		▼											X
	Functional Overlap	FO											X		
	Functional Scope	FS													X
	Generality	GE	▼	▼	▼	▼					X	X	▲		X
	Independence	ID	▼										X	X	X
	System Clarity	ST													X
	System Compatibility	SY		▼									X		
	Virtuality	VR	▼								X				
GENERAL	Modularity	MO	▼			X			X	X	X	X	X	X	X
	Self-Descriptiveness	SD	▼						X	X	X	X		X	X
	Simplicity	SI	▼		X				X	X	X	X			X

NOTE: X = Basic Relationship
▲ = Positive Effect
▼ = Negative Effect
Blank = None or Application Dependent

By providing quantitative analyses of the quality factors, this SQM methodology enables the developer to focus on problem areas. For example, analysis of scoring at the CSCI level could reveal a factor score is consistently low for one function. If so, analysis at the unit level should show the same pattern for the same factor. If the developer then reviews the low scoring metric elements, the nature of the problem should become evident. RADC recommends submitting the results and analysis of the SQM application in a Software Quality Evaluation Report. Additionally, they recommend listing causes for each scoring deficiency separately.

Anticipating the relevant quality factors during the software design phase allows developers to tailor their programs to meet these quality factors. RADC's family of SQM is based on saving labor, costs, and development time later in the project by having the quality factors defined early.

The RADC metrics that are related to code are a subset of this general set. In this Report, a representative set of software metrics and quality factors were extracted and analyzed. Studies indicate high correlations exist between the metric elements and selected quality factors, as listed in table 5.3-4.

TABLE 5.3-4. SQM BASED ON RADC'S METHODOLOGY

Software Metric	Quality Factor
AC.1 through VS.3 (see Bowen, Wigle, and Tsai 1985)	Completeness Consistency Correctness Efficiency Expandability Flexibility Integrity Interoperability Maintainability Modularity Portability Reliability Reusability Simplicity Survivability Usability Verifiability

This set allows the metrician to weight the importance of each quality factor. Weighting the factors makes the results more meaningful, since practitioners will select metrics based on how important they are to the application. For example, if a software application is being designed to run on several different PCs, portability is a major concern. Conversely, if software is being designed to help fly an airplane, reliability will be far more important than portability.

The metrics are built upon a model of eight software quality attributes whose definitions are found in Boehm's National Bureau of Standards research program. In 1976, RADC adopted the methodology and continued to develop it. In 1986, a private consulting firm used these SQM to support six major military defense programs (Murine 1985).

RADC has spent over 10 years developing a method for specifying and evaluating software quality attributes. A similar methodology has been designed into a tool that is used to measure various SQM for Ada mission-critical systems (Keller 1989).

The Jet Propulsion Laboratory, Pasadena, California, has chosen this tool to monitor the quality of Ada code for the FAA's Real-Time Weather Processor (RWP) (Reed 1988). The RWP, designed to facilitate civil air-traffic control, is part of the Advanced Automation System. The tool (based on 153 software metrics that RADC has defined) measures the following SQM:

- Clarity
- Complexity
- Modularity
- Portability
- Reliability
- Simplicity

Besides the government, private companies (including Martin Marietta, Ford Aerospace, TRW, and IBM) are using this tool on Ada development projects. Companies also have used RADC's methodology to perform metric analysis on software developed for the following programs or areas:

- U.S. Air Force Cruise Missile Program
- NASA Space Shuttle/Centaur
- U.S. Navy Navigational Sonar System
- U.S. Air Force Over-The-Horizon Program
- Military and Commercial Jet Engine Quality Assurance Program
- Computer Automated Design/Computer Automated Machines (CAD/CAM)

The empirical evidence supporting this methodology is extensive. Industry correlates RADC's software metrics with the following quality factors:

- Correctness
- Completeness
- Consistency
- Modularity
- Simplicity
- Testability

5.4 SQM Based on Albrecht's Function Points Metric

Albrecht's FP Metric has been viewed as a size metric (Low and Jeffery 1990). Since size has been related to complexity, the metric should measure an aspect of complexity.

Beyond this general implication, the metric is more encompassing than a simple measure of size, such as lines of code. The metric also includes a weighting system, allowing the FP to be weighted, based on their importance. Additionally, it adjusts for a "processing complexity influence". This adjustment includes areas such as performance, the configuration of the system on which the software is running, and complexity processing factors. The last category includes determining the control structure and the decision points in the program and determining the extent to which the program contains mathematical equations.

From the above considerations, Albrecht's metric can be related to several quality factors, and thus can produce several SQM. These SQM are listed in table 5.4-1. Albrecht considers the metric to be a general productivity measure. It was designed to measure program complexity and maintainability, and to be used as a management tool to promote higher productivity (Albrecht 1985).

TABLE 5.4-1. SQM BASED ON ALBRECHT'S FUNCTION POINTS METRIC

Software Metric	Quality Factor
Function Points	Complexity Maintainability Modularity Performance Simplicity Understandability

The FP Metric has received extensive testing. It has been applied to 2212 software systems, including: 1520 business, 153 real time, 150 scientific, 137 system software, 112 command and control, 67 telecommunications, 37 avionic, 15 process control, and 14 microcode (Quantitative Software Management, Inc. 1990). Because it has been used to determine maintainability requirements, it is also a measure that correlates with complexity.

While the metric has been used predominantly in business applications (Drummond 1985), recently, the FP Metric has gained substantial independent empirical experience for its scientific and avionic code applications. The Electronic Systems Division (ESD) at Hanscom Air Force Base has been using an automated tool developed for the FP Metric since October 1987 (Quantitative Software Management, Inc. 1990).

The FP Metric is used at ESD to evaluate contractor performance on software development efforts under contract to the Air Force. First, the metric tool was applied to 22 large scale systems built for ESD from 1981 to 1987. Then, the tool was calibrated to meet the needs of this contracting environment, and was tuned to measure the efficiency with which software systems were built for the Air Force. Currently, the tool is being applied to radar, avionics, real time, and command/control systems to obtain accurate estimates for future developments and enhancements (Quantitative Software Management, Inc. 1990).

This recent empirical evidence shows the FP Metric has been applied in diverse environments. The evidence indicates that it can be tailored to apply to avionic code. Additionally, the metric has received much empirical substantiation.

The empirical evidence provides some qualitative results for the SQM listed in table 5.4-1. The Function Points Metric showed a 93.5 percent correlation to work hours (Albrecht and Gaffney 1983).

For these relationships to be meaningful, one could relate work-hours to development time and maintainability. One could further relate the amount of function to complexity. By further analysis, since simplicity is the inverse of complexity, the FP measure provides an SQM that measures simplicity. One could also consider understandability to be a subset of complexity. Hence, a measure that quantifies complexity also provides an indication of how easily code can be understood.

While this study provides quantitative results, it does not provide independent substantiation of these SQM. An independent study by Behrens (1983), however, does provide empirical evidence that supports the SQM correlation of the metric with productivity. When productivity is defined in terms of quality factors, this study determined that the FP Metric does provide the viable SQM mentioned in the previous paragraph.

Because complexity and maintainability encompass several other quality factors, the measure can produce SQM that support these other quality factors as well. The FP Metric correlates with the following quality factors, thus yielding SQM that estimate:

- Clarity
- Conciseness

- Complexity (Structural)
- Efficiency
- Maintainability
- Performance
- Simplicity
- Testability
- Understandability

5.5 SQM Based on Ejiogu's Software Metrics

Ejiogu's metrics will produce repeatable, monotonic, counts provided that nodes are defined consistently. His metrics measure the structural complexity in a module. It is a more integrated measure than McCabe's, because while it measures the branching structure of the program, S_c, it also considers its size. Thus, the SQM measures more aspects of complexity.

While the theory behind the SQM is promising, correlations have not been empirically substantiated. When such experiments are conducted, the SQM resulting from correlating Structural Complexity with the quality factors listed in table 5.5-1 will prove to be useful.

TABLE 5.5-1. SQM BASED ON EJIOGU'S STRUCTURAL COMPLEXITY METRICS

Software Metric	Quality Factor
Structural Complexity, S_c and Size (S)	Complexity Maintainability Modularity Performance Simplicity Understandability

5.6 SQM Based on Henry and Kafura's Information Flow Metric

This metric uses the flow between procedures to show the data flow complexity of a program. The metric is more widely applicable, therefore, than a metric that just looks at the flow within procedures. The SQM address structural complexity in terms of the application's data relationships.

Kafura and Reddy (1987) found a correlation of 95 percent between errors and procedure complexity. No other results have been published, although the metric claims to measure structural complexity and other related quality factors, such as those listed in table 5.6-1. The theory behind the Information Flow Metric is sound; the metric only lacks independent experimental validation.

TABLE 5.6-1. SQM BASED ON HENRY AND KAFURA'S INFORMATION FLOW METRIC

Software Metric	Quality Factor
Information Flow Complexity (IFC)	Complexity Maintainability Modifiability Modularity Performance Reliability Simplicity Understandability

CONCLUSIONS

6.1 SQM Certification Issues

SQM address software quality and attempt to quantify it. The SQM built from the software metrics discussed provide developers with tools to assess the quality of software.

SQM provide a quantified approach to developing software. Specifically, SQM results allow the CE to determine if:

- The software was developed according to a disciplined approach (RTCA/DO-178A, paragraph 1.1).

- The software was developed and tested by a process appropriate to the software level (RTCA/DO-178A, paragraph 5.2.1 and 6.2.1).

- The SQM-based requirements analysis substantiates that the associated requirements are fulfilled (RTCA/DO-178A, paragraph 6.2.5.3.1).

The major, overriding purpose of SQM is to create measures that attempt to quantify software quality. In the six families of software metrics discussed, each family contains measures that are repeatable and monotonic. These measures, in turn, have been correlated with SQFs to yield viable SQM. The SQM provide numbers that signify the extent to which a particular software quality is present.

The SQM approach attaches numbers to software attributes. These numbers are then used in correlations to provide relative, quantitative measures of software qualities. This approach differs from the traditional approach, which is based on subjective evaluations.

Curtis (1980) believes correlational studies are not hard science, and that correlational knowledge is closer to the behavioral sciences than the physical. However, most critics agree that important statistical inferences can be drawn from such studies. From these inferences developers can measure the quality of software.

The statistical approach is used with striking success in other practical applications, such as circuit design. Using statistical analysis to determine software quality allows a developer to have a degree of confidence that software has met a certain quality level. Certainly SQM estimates are a more disciplined approach than traditional management estimates.

Valid SQM can also assess whether the software was developed and tested by a process appropriate to the software level. The developer must devise a scale by which to rate critical modules, essential modules, and nonessential modules. For example, a module is considered to be overly-complex if it has a Cyclomatic Complexity number greater than 10. Within the range of 1 to 10, however, a developer can determine that all critical modules must not have a Cyclomatic Complexity number greater than six, all essential modules must not exceed a v(G) of eight, and all nonessential modules' v(G) shall not be greater than 10. By establishing this scale, a developer can determine whether various modules "pass" this requirement.

These scales can be incorporated in software requirements documents, thus strengthening SQM's role in software development. SQM can strengthen requirements analysis because they provide a way to assess the degree to which the requirements are fulfilled. Hence, a CE can place a greater degree of confidence that the software has met a specific level of quality.

While all viable SQM can be applied effectively to the above areas, the next section shows how SQM can help the CE to determine if:

- The SQM-based test coverage analysis substantiates test coverage (RTCA/DO-178A, paragraph 6.2.5.3.2).

- The software possesses the required software qualities.

6.2 SQM Based on Halstead's Software Metrics

Several of Halstead's software metrics have been applied, and have been substantiated by empirical study to correlate with SQFs. Quantitative studies indicate that the Implementation Length, N, has been successfully correlated with maintainability, number of bugs, and reliability. Implementation Length is also correlated with flexibility, testability, portability, reusability, and interoperability (McCall, Richards, and Walters 1977; and Bowen, Wigle, and Tsai 1985). This software metric may be related to understandability, since this SQF is a subset of maintainability. Further, since understandability is complementary to complexity, the measure is related to complexity (Munson and Khoshgoftaar 1990).

Volume and Programming Effort have also produced several SQM. They have each been successfully correlated with complexity, maintainability, modifiability, modularity, number of bugs, reliability, simplicity, and understandability.

Implementation Length, Volume, and Effort form the basic software metric set, from which the other Halstead measures are derived. While correlational and other empirical studies quantify several attributes of software, do these measures suffice to give the overall software quality of a product? Their limitations suggest that they do not cover all the attributes that compose even one quality factor.

Halstead's software metrics provide practical SQM. When developers use his metrics, the resulting SQM indicate the structural and psychological complexity of a module. The software metrics, however, are all built on the fundamental counts of operators and operands. The resulting SQM thus have a narrow focus because the measures are not context-sensitive. The metrics do not address other aspects of code complexity, such as control flow.

While the empirical evidence supports the SQM-claim that Halstead's metrics measure an aspect of complexity, the metrics are primarily concerned with the size or volume of an application. While size is an indicator of complexity, it is not the only aspect of this quality factor. The developer should consider combining metrics to capture more of the total quality factor complexity.

As a partial indicator of code complexity, Halstead's SQM are well-supported and provide a good, disciplined approach to aid in developing software. They do not, however, provide structural test coverage analysis.

6.3 SQM Based on McCabe's Software Metric

McCabe's SQM, like Halstead's, is narrowly focused. McCabe's metrics look at the control flow to determine the structural complexity of a module. The metrics are not very context-sensitive. They only address conditional statements.

Kitchenham (1981) casts some doubts on the generality of McCabe's metrics. She failed to find strong correlations between complexity and McCabe's and Halstead's metrics. Kitchenham concedes that more research is required to identify the particular conditions under which these metrics can be applied.

The narrow focus means the v(G)-Complexity SQM provides only a partial assessment of the complexity of a module. Other SQM pairs, such as v(G)-Simplicity and v(G)-Understandability provide a partial assessment for the same reason. While they are viable SQM, they should be used as a measure of how control-flow affects each of the software qualities.

McCabe's Cyclomatic Complexity Metric fares better with respect to the following SQM:

- v(G)-Maintainability
- v(G)-Modularity
- v(G)-Testability

Industry supports these SQM as valid indicators of the extent to which a program is maintainable, modular, and easily tested. The v(G)-Testability SQM continues to receive the most attention in private industry and the government.

McCabe's Cyclomatic Complexity Measure addresses the structural coverage requirements for assessing test coverage. However, although it determines the maximum number of test cases required to generate branch coverage, it does not provide a number that will indicate path coverage. Hence, if this SQM is used to calculate the number of test cases, it should not be used to calculate the critical modules' test cases, which require path coverage.

6.4 SQM Based on RADC's SQM Methodology

RADC's methodology attempts to quantify software qualities at each stage of the software development life cycle. DOD-STD-2167A requires that the requirements for important quality factors, such as reliability and maintainability, be specified in the System Specification (DID-

CMAN-80008A, paragraphs 10.1.5.2.5, 10.1.5.2.5.1, 10.1.5.2.5.2, and 10.1.5.2.5.3). Additionally, paragraph 10.1.5.2.5.4 of the DID requires that the requirements for other quality factors such as integrity, efficiency, and correctness be specified. RADC's methodology has been supported by industry and the government. In fact, the IEEE Computer Society has incorporated a similar methodology in its draft ISO standard for an SQM methodology.

This methodology addresses certification concerns because its SQM determine whether particular software qualities are present in software, and if present, to what degree. For example, software designed to fly a plane must be very reliable. To measure the software with respect to this quality factor, developers would choose the criteria and complementary factors associated with reliability and would weight the criteria appropriately. Other factors of lesser concern, such as maintainability, would not have to be considered, or would be weighted lower to affect the result accordingly.

RADC's SQM involve all quality factors discussed in chapter 4, and others that are listed in chapter 5. The majority of the evidence in support of RADC's SQM stems from SQM applications. Milliman and Curtis' (1980) and Pierce, Hartley, and Worrells' (1987) studies also substantiate RADC's SQM. This methodology helps managers assess a project's overall effort, cost, and quality.

Shumskas (1990) emphasizes that RADC's methodology can be used as a management tool. He refers to SQM as software indicators. The indicators, though not absolute, use a combination of planning goals and trend data to enhance the evaluation process and to provide status assessment tools. Originally published by the Air Force Systems Command (AFSC) as AFSC Pamphlet (AFSCP) 800-14 and AFSCP 800-43, the Army Materiel Command (AMC) has adopted RADC's methodology (AMC-P 70-13 and AMC-P70-14). The Navy is preparing its versions of these documents stressing the use of software indicators to assess software quality.

RADC's methodology is best used by applying the appropriate metrics to every phase. In this way, the software project's quality is monitored throughout its development. The methodology was developed with this approach in mind.

RADC's methodology provides formulas to measure aspects of code. Other software metrics may be used to provide the counts needed in the formulas. For example, a tool that incorporates McCabe's Cyclomatic Complexity Metrics might be modified to produce counts for conditional branch statements and unconditional branch statements, thus providing the counts needed to assess SI.3(1). Hence, the methodology is the most flexible of all the families.

Unfortunately, there are several problems with implementing the software metrics. The language of the metric elements is often confusing or ambiguous. Further, there are consistency problems that occur when one tries to combine the metric elements of Part A with their associated questions in Part B. Unless future revisions of these documents resolve the semantic and practical problems that exist, developers cannot use these software metrics easily.

6.5 SQM Based on Albrecht's Software Metric

Albrecht's FP Metric was designed to be a productivity measure. His development tool, however, has been widely used in industry and government applications as SQM (Behrens 1983; Drummond 1985; Low and Jeffery 1990; and Quantitative Software Management, Inc. 1990) to measure quantitative aspects of software quality. The SQM are based on the idea that productivity is related to maintainability and complexity. Further, since complexity encompasses other quality factors such as simplicity and understandability, the SQM can be used to assess these attributes of software as well.

The FP Metric has been used in business, and more recently in avionic code applications. RADC has applied Albrecht's metric to several of their projects. Private companies as well, most notably IBM, have incorporated this metric into their SQA program.

The studies, however, do not indicate whether code was used to obtain the resulting SQM. The developer can also derive SQM using other information, such as Software Change Reports and Design Documents as input to this metric. Because the studies do not clearly identify the input, it is unclear whether the results are repeatable or monotonic among studies.

This metric is much broader in scope than McCabe's and Halstead's, since it includes a size measurement and adjusts for processing complexity influences. Assuming the metric is applied to code, the resulting SQM will be valid, as long as the type definitions are followed consistently when counts are conducted.

The major difference between Albrecht's SQM and the ones listed in section 6.2, is that Albrecht's FP Metric has not undergone formal experimentation. Because the metric and its resulting SQM have been widely used in industry for over ten years, the SQM have been substantiated. Subjecting the metric to formal scientific experiment will further strengthen its validity.

6.6 SQM Based on Ejiogu's Software Metrics

Ejiogu's Structural Complexity Metric, S_c, is more integrated than McCabe's or Halstead's complexity metrics because it measures the branching structure of the program, and considers its size. Hence, the resulting SQM give a broader assessment of a program's complexity.

While this software metric has its strengths, the theory upon which the metric is based contains two questionable assumptions. Ejiogu assumes metricians will define nodes consistently. Further, he assumes everyone will refine nodes to the same degree. Because Ejiogu does not provide rules for refining nodes, the counts upon which the SQM are based may not be monotonic or repeatable. Hence, the metric may not be reliable. As of this writing, Ejiogu is creating a book on his SQM that may clear up the ambiguities mentioned.

Because the metrics measure the size and the branching structure of a program, they could produce the following pairs of SQM:

- Structural Complexity-Maintainability
- Structural Complexity-Modularity
- Structural Complexity-Performance
- Structural Complexity-Simplicity
- Structural Complexity-Understandability

The lack of empirical evidence is the greatest problem here. While in-house experiments have been conducted, no studies have been published. The SQM are promising because they attempt to measure various aspects of complexity and other quality factors. Ejiogu's methodology needs to be empirically substantiated, however, before developers can rely upon these SQM.

6.7 SQM Based on Henry and Kafura's Software Metric

The Information Flow Metric considers the lines of code and the relationships among modules to assess the complexity of a program. It measures the structure of the code, rather than the code itself. Hence, the SQM are not bound to a particular language or programming style.

Like Albrecht's FP Metrics, this metric needs further independent empirical evidence. The greatest problem with the SQM produced from this metric lies in implementing the metric. The programmer must understand the program in order to identify correctly the types of information flows. Anyone evaluating information flows must be well-trained to ensure the counts will be correct. Additionally, while a tool will produce a repeatable count, it must be able to handle all the possibilities of indirect flows.

To show it is a meaningful measure, Henry and Kafura (1984) applied this metric to the UNIX^{TM} operating system. The study corroborated the information flow-complexity SQM. It remains to be seen, however, whether independent studies will produce similar statistical proof. If a developer plans to use this measure and its resulting SQM, the CE might request that definitions of the information flow types be included in the certification package.

6.8 The Hybrid Approach

RADC and the Space and Missile Test Center (SAMTEC) at the Vandenburg Air Force Base jointly developed a plan that used SQM to improve software development. The plan was called the Advanced Systematic Techniques for Reliable Operational Software (ASTROS). A study by Curtis and Milliman (1980) compared two software projects: one that had been developed using traditional software testing techniques, and one that had been developed using ASTROS. The two were compared to determine which software was overall superior in quality.

The Data Analysis Processor (DAP) code was developed conventionally, while the Launch Support Data Base (LSDB) was developed using the ASTROS plan. The researchers concluded that the LSDB code exhibited a higher Halstead Program Level, indicating the algorithms were more succinctly represented. Additionally, the quality of the code was superior to that of the DAP project's.

McCabe's Cyclomatic Complexity Metric was used to determine the control flow of the code in both projects. The LSDB program showed uniform breaks in structured programming, thus allowing program readers to isolate particular constructs that caused the break. The DAP code, on the other hand, exhibited more varied departures from structured principles. The resulting control flow was convoluted, and was more difficult to comprehend and to trace.

The most impressive finding of the study concerned the number of post-development errors. The LSDB contained two and a half times as many executable lines, but only two-thirds as many post-development errors, compared with the DAP project's code. In this study, Halstead's Number of Bugs estimate accurately predicted the reliability of the LSDB software.

The researchers further concluded that software developed using metrical analysis will perform better and will exceed the quality of that produced by conventional techniques. This study strongly supports using a hybrid metric approach for developing and testing software. The study used Halstead's, McCabe's, and RADC's metrics to measure different attributes of the software. Tools on the market typically use the hybrid approach, also.

6.9 Directions for SQM in the Certification Field

In a meeting in December 1990, the ISO working group on metrics reported on current metric activities. According to the minutes of the meeting, the U.S. has taken steps to keep the lead in SQM and standards. In the U.S., practitioners are focusing on quality indicators (or factors) for process control and end item testing. Japan and Germany, however, are interested in the buyer aspects of software products, not on the production of software. They want to use consumers as the basis for evaluating whether the software contains the quality users expect. This emphasis on end-user requirements is a good check-and-balance system for assuring quality objectives have been met. While U.S. software companies continue to stress the production of software, they could benefit from placing more importance on user satisfaction in assessing software quality.

The hybrid approach for using metrics seems to work best in practice. As the government standards attest, it is impossible to measure the complete quality attributes of an application by using just one metric (Bowen, Wigle, and Tsai 1985). Combining metrics allows the developer to target critical software objectives. Once the objectives are established, the metrician will know which metrics meet the needs of the application, and will be able to use metrical analysis to its fullest advantage.

Many metrics focus on narrow pieces of software quality. By combining these metrics with other families, the practitioner will be able to assess the overall quality of the software that is submitted for certification. If the developer documents and provides complete metrical analysis in the package, the CE will have a greater degree of confidence that the avionic software meets its required level of quality in those areas that are important to the application and the mission.

6.9.1 Future Experiments

SQM continue to be validated. Confidence in SQM increases over time as more projects use the valuable information SQM provides, and as the sample size increases. Greatest confidence occurs when metrics have been validated based on data collected from previous projects.

As researchers continue to assess software quality with SQM, the thrust of future experiments should be to verify the correlations that form the useful SQM. While practical SQM have been applied to software for at least 15 years, they, too, should be subjected to more formal verification. SQM do provide measures of quality, although most of the measures are relative and very context-dependent. Since SQM are composed of composite quality factors, which in turn, are related to several aspects of a software quality, composite SQM reflect the need to quantify several aspects of software quality.

APPENDIX A - SOFTWARE METRIC DATA SHEETS

This appendix contains individual data sheets on each of the most prevalent software metrics encountered in this study. The data sheets are a summary of the concepts, constraints, criticisms, and other data needed to understand and critically evaluate the SQM that rely on these metrics.

Although the data sheets are not primarily intended as a guide for applying the metrics, some of the data sheets contain sufficient information that the metric could be properly and fully calculated by following the data sheet. On the other hand, some of the metrics are so extensive that the data sheets could not encompass all the necessary parts. See the references listed on the data sheet for a complete coverage of a particular metric.

Each data sheet covers only one metric, and they all contain the same fields of information. The fields are defined only as they apply to measuring source code. Many of the metrics can be applied more broadly, but the other possibilities are not addressed here.

The "SQM RELATIONSHIPS" field lists those quality factors that are known to be related to the particular software metric. This serves as a cross-reference to the applicable Software Quality Factor Data Sheet of appendix B.

The "TOOL" information is supplied on an as-available basis. It is not to be taken as a comprehensive list of tools available. Nor is the information to be taken as a recommendation.

Metrics that share a common heritage of theory are grouped together by family. The family name is given at the top of each sheet. The data sheets are arranged alphabetically by family name. Within each family they are arranged in the order in which they build upon each other, from the fundamental level to the highest level. When there is no hierarchical relationship within a family, the metric data sheets are arranged alphabetically by metric name.

Function Points

This metric calculates a weighted sum of the number of elementary functions supported by a module of source code. The functions are categorized into five types. The number of Function Points (FP) provided to the user by a module of source code is given by the following rule:

RULE: FP = FC x PCA

FC is the function count produced as follows:

Function Count = Number of External Inputs x 4, plus
Number of External Outputs x 5, plus
Number of Logical Internal Files x 10, plus
Number of External Interface Files x 7, plus
Number of External Inquiries x 4

where the weights given were determined experimentally. They are the values that made the measure most accurately reflect the perceived amount of function delivered in real projects of average complexity. PCA is the Processing Complexity Adjustment Factor, calculated as follows:

PCA = 0.65 + (0.01 x PC),

where PC is the Processing Complexity. It is based on an estimate of the degree of influence that each of 14 specified application characteristics (Albrecht and Gaffney 1983) has on the complexity of the functions counted. Each characteristic is assigned a "0", "1", or "2" degree of influence. Then the PC is the sum of these 14 numbers. This produces a PCA which adjusts the function count by ±35 percent for application and environment complexity.

DOMAIN: This measure only applies to source code that implements an algorithm; the code must perform some function that relates to user inputs, user outputs, internal files, interface files, and/or user inquiries.

RANGE: This metric produces a real number greater than or equal to 1.3, the number of function points in the simplest algorithm which must contain an input and an output (assuming the lowest possible weight of one for each category and the lowest possible PCA of 0.65).

NECESSARY CONDITIONS: The criteria for determining the inputs, outputs, internal files, interface files, and inquiries from code must be defined for the context. The criteria for determining whether they are internal or external must be defined for the context. In order to use the weights given, the practitioner must also use Albrecht's definitions for the five types of function (Albrecht and Gaffney 1983).

QUALIFICATIONS: This metric has been validated primarily in a data processing system environment, rather than a real-time flight control environment. The given weights probably do not apply to code in avionic equipment.

The given weights are for programs that fall into a qualitatively chosen category of average complexity. Different sets of weights are given for simple or complex programs (Albrecht and Gaffney 1983).

The FP of an application is not necessarily equal to the sum of the FP of each of its sub-applications. Thus, FP is not an additive quantity.

CRITICAL ANALYSIS: This measure is not necessarily strictly monotonic or repeatable. A program reader may erroneously identify an input, output, internal file, interface file, or inquiry. In this case, an increase in the measure does not represent an increase in the amount of function. Furthermore, a different reader will likely judge the attributes differently. This effect has been observed in experimental analysis (Low and Jeffery 1990).

A problem with repeatability can also result from the arbitrariness of establishing application boundaries. The larger the application size, the more functions will be included. Furthermore, as the boundary is enlarged files will shift from being external interface files to being logical internal files. It is important that all measurements be based on an unambiguous specification for establishing application boundaries.

A software tool can also produce non-monotonic results. Although it has well-defined rules for identifying the attributes, the rules will likely misinterpret some uncommon realization. Whether the count produced by a tool is right or wrong, it will certainly produce a repeatable result.

Despite these problems, if the program reader or the software tool follows carefully defined rules, either can generate highly reliable counts.

The FP metric uses the source code to deduce the characteristics of the algorithm solved by the program. The rules used for this step are dependent on the programming language and technology used. But it is the algorithm that is measured, not the program. Thus, as Drummond (1985) also points out, the FP metric has the advantage of being language independent, as well as being independent of programmer style or experience. It is even independent of any changes in technology. Once the size of the algorithm is calculated, any program in any language that performs the same algorithm, whether poorly or well written, will provide the same number of function points.

SQM RELATIONSHIPS: Complexity, Simplicity

TOOLS: Checkpoint, Software Productivity Research, Inc., Burlington, MA
SIZE PLANNER, Quantitative Software Management, Inc., McClean, VA

REFERENCES: Albrecht 1979, 1985; Albrecht and Gaffney 1983; Behrens 1983; Drummond 1985; Jones 1988; Low and Jeffery 1990

H - Height of a Tree

This metric quantifies the depth of nesting in a set of modules of source code. It is based on the number of levels in a hierarchy tree of the control flow between modules.

RULE: The Height of a hierarchy tree is the maximum height attained by any node in the hierarchy tree, except that the Height of a single node tree is defined to be one.

The height of a node is the number of levels of nesting below the root node. Thus, the root node has a height of zero and any child node of the root node has a height of one, being nested one level below the root node. Because the tree is drawn upside-down, height increases with lower levels of nesting.

A module (or node) is defined to be a set of code that embodies an aggregate of thought representing a single function. A module must be a well-defined entity in order to ensure that the count is repeatable.

DOMAIN: This measure only applies to source code that implements an algorithm; the code must perform some function which may or may not be composed of subfunctions.

RANGE: This metric produces an integer greater than or equal to one, the Height of the tree for source code that has no nested modules.

NECESSARY CONDITIONS: The combinations of characters that constitute a node, the source code constructs that constitute nesting, and the boundaries of a module must be defined for the context.

QUALIFICATIONS: None.

CRITICAL ANALYSIS: This measure is not necessarily strictly monotonic or repeatable. A program reader may erroneously nest a node when drawing the hierarchy tree. In this case, an increase in the measure does not represent an increase in the depth of nesting. Furthermore, a different reader will likely judge nesting differently.

A software tool can also produce non-monotonic results. Although it has well-defined rules for identifying nodes and nesting, the rules will likely misinterpret some uncommon realization. Whether the count produced by a tool is right or wrong, it will certainly produce a repeatable result.

Despite these problems, if the program reader or the software tool follows carefully defined rules, either can generate highly reliable counts.

SQM RELATIONSHIPS: Complexity, Performance, Simplicity, Understandability

TOOLS: COMPLEXIMETER, Softmetrix, Inc., Chicago, IL

REFERENCES: Ejiogu 1984^1, 1984^2, 1987, 1988, 1990

Twin Number

This metric quantifies the breadth of explosion of a module (or node) of source code. It is based on the number of nodes nested directly below it (child nodes) in a hierarchy tree of the control flow between nodes.

RULE: The Twin Number of a node is the number of child nodes of which it is the parent, except that the Twin Number of the node in a single node tree is defined to be one.

Every node that is immediately related to a particular node at the next lowest level (one level closer to the root) is considered a child of that node.

A module (or node) is defined to be a set of code that embodies an aggregate of thought representing a single function. A module must be a well-defined entity in order to ensure that the count is repeatable.

DOMAIN: This measure only applies to source code that implements an algorithm; the code must perform some function which may or may not be composed of subfunctions.

RANGE: This metric produces an integer greater than or equal to zero, the Twin Number of a module with no child nodes (monad).

NECESSARY CONDITIONS: The combinations of characters that constitute a node, the source code constructs that constitute nesting, and the boundaries of a module must be defined for the context.

QUALIFICATIONS: None.

CRITICAL ANALYSIS: This measure is not necessarily strictly monotonic or repeatable. A program reader may erroneously nest a node when drawing the hierarchy tree. In this case, an increase in the measure does not represent an increase in the number of child nodes. Furthermore, a different reader will likely judge nesting differently.

A software tool can also produce non-monotonic results. Although it has well-defined rules for identifying nodes and nesting, the rules will likely misinterpret some uncommon realization. Whether the count produced by a tool is right or wrong, it will certainly produce a repeatable result.

Despite these problems, if the program reader or the software tool follows carefully defined rules, either can generate highly reliable counts.

SQM RELATIONSHIPS: Complexity, Simplicity

TOOLS: COMPLEXIMETER, Softmetrix, Inc., Chicago, IL

REFERENCES: Ejiogu 1984^1, 1984^2, 1987, 1988, 1990

M - Monadicity

This metric quantifies the number of irreducible modules (or nodes) in a set of modules of source code. It is based on the number of childless nodes in a hierarchy tree of the control flow between modules.

RULE: The Monadicity of a hierarchy tree is the number of nodes that have no children (a monad), except that a tree with a Height of one is constrained to have a Monadicity of one.

Every node that is immediately related to a particular node at the next lowest level (one level closer to the root) is considered a child of that node.

A module (or node) is defined to be a set of code that embodies an aggregate of thought representing a single function. A module must be a well-defined entity in order to ensure that the count is repeatable.

This is a measure of the number of leaves on the tree, where the node at the end of each branch is a leaf. It is a measure of the "bushiness" of the tree.

DOMAIN: This measure only applies to source code that implements an algorithm; the code must perform some function which may or may not be composed of subfunctions.

RANGE: This metric produces an integer greater than or equal to one, the Monadicity of tree with a Height of one.

NECESSARY CONDITIONS: The combinations of characters that constitute a node, the source code constructs that constitute nesting, and the boundaries of a module must be defined for the context.

QUALIFICATIONS: None.

CRITICAL ANALYSIS: This measure is discontinuous between hierarchy trees of Height one and two. A tree with five child nodes at level one and one child node at level two has a monadicity of five. But if the single monad at level two is removed, the monadicity becomes one.

Even for hierarchy trees with a Height greater than one, this measure is not necessarily strictly monotonic or repeatable. A program reader may erroneously divide a monad into two monads when drawing the hierarchy tree. In this case, an increase in the measure does not represent an increase in the number of monads.

Furthermore, a different reader will likely judge nodes differently.

A software tool can also produce non-monotonic results. Although it has well-defined rules for identifying nodes and nesting, the rules will likely misinterpret some uncommon realization. Whether the count produced by a tool is right or wrong, it will certainly produce a repeatable result. Despite these problems, if the program reader or the software tool follows carefully defined rules, either can generate highly reliable counts.

SQM RELATIONSHIPS: Complexity, Simplicity

TOOLS: COMPLEXIMETER, Softmetrix, Inc., Chicago, IL

REFERENCES: Ejiogu 1984^1, 1984^2, 1987, 1988, 1990

S - Software Size

This metric calculates the size of a set of modules (or nodes) of source code. It is based on the number of nodes in the hierarchy tree of the control flow between modules.

RULE: Count the number of nodes above the root node.

A module (or node) is defined to be a set of code that embodies an aggregate of thought representing a single function. A module must be a well-defined entity in order to ensure that the count is repeatable.

DOMAIN: This measure only applies to source code that implements an algorithm; the code must perform some function which may or may not be composed of subfunctions.

RANGE: This metric produces an integer greater than or equal to zero, the size of a single-node tree.

NECESSARY CONDITIONS: The combinations of characters that constitute a node, the source code constructs that constitute nesting, and the boundaries of a module must be defined for the context.

QUALIFICATIONS: None.

CRITICAL ANALYSIS: This measure is not monotonic. When two different modules call the same subroutine, it appears as two distinct nodes on the hierarchy tree. This increases the size even though there is no increase in function.

Even if a program contains no such problem, this measure is not necessarily strictly monotonic or repeatable. A program reader may erroneously divide a monad when drawing the hierarchy tree. In this case, an increase in the measure does not represent an increase in the number of monads. Furthermore, a different reader will likely judge nodes differently.

A software tool can also produce non-monotonic results. Although it has well-defined rules for identifying nodes and nesting, the rules will likely misinterpret some uncommon realization. Whether the count produced by a tool is right or wrong, it will certainly produce a repeatable result.

Despite these problems, if the program reader or the software tool follows carefully defined rules, either can generate highly reliable counts.

Very little experimental validation has been performed for this metric and none of it has been published. No independent experiments have been performed.

SQM RELATIONSHIPS: Complexity, Performance, Simplicity, Understandability

TOOLS: COMPLEXIMETER, Softmetrix, Inc., Chicago, IL

REFERENCES: Ejiogu 1984[1], 1984[2], 1987, 1988, 1990

S_c - Structural Complexity

This metric calculates the structural complexity of a set of modules of source code. It is based on the number and relationship of nodes in a hierarchy tree of the control flow between nodes,

RULE: $S_c = H \times R_t \times M$

where H is the Height of the tree, R_t is the Twin Number of the root node, and M is the Monadicity of the tree. Each of these arguments is defined on a previous data sheet.

A module (or node) is defined to be a set of code that embodies an aggregate of thought representing a single function. A module must be a well-defined entity in order to ensure that the count is repeatable.

The Structural Complexity reflects the nature of "the relation of conglomerates of thoughts expressing the functional composition of the system" (Ejiogu 1984). It combines into one measure both the number of related modules of thought and the closeness of their relationship. This measure was designed to be applied to issues relating to programming productivity, since it quantifies the amount of complexity in a program.

DOMAIN: This measure only applies to source code that implements an algorithm; the code must perform some function which may or may not be composed of subfunctions.

RANGE: This metric produces an integer greater than or equal to one, the Structural Complexity of a single-node tree.

NECESSARY CONDITIONS: The combinations of characters that constitute a node, the source code constructs that constitute nesting, and the boundaries of a module must be defined for the context.

QUALIFICATIONS: While this metric is based on structured programming concepts, unstructured programs do not render it useless. Any effect that lack of structure has on the value of the measure is to be taken as a reflection of the increased complexity that results when a program deviates from the ideal. Excessive values alert the developer to the presence of unstructured code so that it can be determined whether its presence is an asset or a detriment. For instance, when two nodes call the same subroutine the count is increased, even though there is no increase in function. But since calling a subroutine twice is not a detrimental effect, this apparent increase in complexity is allowed to remain.

CRITICAL ANALYSIS: This measure is not necessarily strictly monotonic or repeatable. The problems are due to the counts that compose it. See the respective software metric data sheets for details. Very little experimental validation has been performed for this metric and none of it has been published. No independent experiments have been performed.

SQM RELATIONSHIPS: Complexity, Performance, Simplicity, Understandability

TOOLS: COMPLEXIMETER, Softmetrix, Inc., Chicago, IL

REFERENCES: Ejiogu 1984[1], 1984[2], 1987, 1988, 1990

η_1 - Number of Unique Operators

This metric quantifies the number of unique operators that occur in a module of source code.

RULE: Count the number of unique operators in the defined module.

An operator is any word of source code that represents an operation to be performed. (A word of source code is any sequence of characters set off as such by the delimiters defined for the particular language.) The operation is performed on the objects that are the arguments of the operator. In computer program source code, most operators are the keywords that define a program statement. Some keywords have multiple parts that must occur as a set. They constitute one compound operator. The classification of each word of code must be based on an analysis of what that word of code does, from a functional point of view.

Uniqueness is to be determined according to the uniqueness of function rather than syntactical form. A unique function is rarely represented by more than one form. When this does occur, the various forms are not to be counted as unique operators. For example, if an exponent is denoted by either an "E" or a "^", these two forms represent one function. Conversely, two unique functions may be represented by the same form. When this occurs, the form is to be counted as two distinct operators, as indicated by the context. For example, if an asterisk denotes either multiplication or a comment, this one form represents two functions.

A module is defined to be a set of code that is to have a measure assigned to it. It must be a well-defined entity in order to ensure that the count is repeatable.

DOMAIN: This measure only applies to that part of source code that implements an algorithm; the code must contain solely operators and operands. An algorithm with less than two operators and one operand is undefined.

RANGE: This metric produces an integer greater than or equal to two, the number of operators in the most succinct implementation of an algorithm.

NECESSARY CONDITIONS: Every possible combination of characters that constitutes an operator, the criteria for establishing uniqueness, and the boundaries of a module must be defined for the context.

QUALIFICATIONS: The number of unique operators in a module is not necessarily equal to the sum of the values for each of its sub-modules. Thus, η_1 is not an additive quantity.

CRITICAL ANALYSIS: This measure is not necessarily strictly monotonic or repeatable. A program reader may increment the count for a word that is not an operator. In this case, an increase in the measure does not represent an increase in the number of operators. Furthermore, a different reader will likely define a different set of operators.

A software tool can also produce non-monotonic results. Although it has well-defined rules for identifying operators, the rules will likely misinterpret some uncommon realization. Whether the count produced by a tool is right or wrong, it will certainly produce a repeatable result.

Despite these problems, if the program reader or the software tool follows carefully defined rules, either can generate highly reliable counts.

SQM RELATIONSHIPS: Clarity, Complexity, Modifiability, Performance, Reliability, Simplicity, Understandability

TOOLS: LOGISCOPE, Verilog USA Inc., Alexandria, VA
PC-Metric, SET Laboratories, Inc., Mulino, OR

REFERENCES: Albrecht and Gaffney 1983; Arthur 1983; Basili, Selby, and Phillips 1983; Bulut 1974; Carver 1986; Coulter 1983; Curtis 1980; Elshoff 1976; Fitzsimmons 1978; Funami 1976; Gordon 1976, 1979; Halstead 1972, 1977, 1979; Halstead and Zislis 1973; Henry, Kafura, and Harris 1981; Jones 1978; Kafura and Reddy 1987; Lassez 1981; Li and Cheung 1987; Prather 1984; Shen, Conte, and Dunsmore 1983; Shneiderman 1980; Siyan 1989; Weyuker 1988; Zislis 1973; Zweben 1977

η_2 - Number of Unique Operands

This metric quantifies the number of unique operands that occur in a module of source code.

RULE: Count the number of unique operands in the defined module.

An operand is any word of source code that represents an argument upon which an operation is to be performed. (A word of source code is any sequence of characters set off as such by the delimiters defined for the particular language.) In computer program source code, most operands are the variables and constants of the program. The classification of each word of code must be based on an analysis of what that word of code does from a functional point of view.

Uniqueness is to be determined according to the uniqueness of function rather than syntactical form. A unique argument is rarely represented by more than one form. When this does occur, the various forms are not to be counted as unique operands. For example, if a variable name can be denoted by either VELOCITY or VEL, these two forms represent one argument. Conversely, two unique arguments may be represented by the same form. When this occurs, the form is to be counted as two distinct operands, as indicated by the context. For example, if the constant, 10, denotes 10 array elements in one part of a program, but it also denotes 10 volts in another part, this one operand represents two arguments.

A module is defined to be a set of code that is to have a measure assigned to it. It must be a well-defined entity in order to ensure that the count is repeatable.

DOMAIN: This measure only applies to that part of source code that implements an algorithm; the code must contain solely operators and operands. An algorithm with less than two operators and one operand is undefined.

RANGE: This metric produces an integer greater than or equal to one, the number of operands in the most succinct implementation of an algorithm.

NECESSARY CONDITIONS: Every possible combination of characters that constitutes an operand, the criteria for establishing uniqueness, and the boundaries of a module must be defined for the context.

QUALIFICATIONS: The number of unique operands in a module is not necessarily equal to the sum of the values for each of its sub-modules. Thus, η_2 is not an additive quantity.

CRITICAL ANALYSIS: This measure is not necessarily strictly monotonic or repeatable. A program reader may increment the count for a word that is not an operand. In this case, an increase in the measure does not represent an increase in the number of operands. Furthermore, a different reader will likely define a different set of operands. A software tool can also produce non-monotonic results. Although it has well-defined rules for identifying operands, the rules will likely misinterpret some uncommon realization. Whether the count produced by a tool is right or wrong, it will certainly produce a repeatable result.

Despite these problems, if the program reader or the software tool follows carefully defined rules, either can generate highly reliable counts.

SQM RELATIONSHIPS: Clarity, Complexity, Modifiability, Performance, Reliability, Simplicity, Understandability

TOOLS: LOGISCOPE, Verilog USA Inc., Alexandria, VA
PC-Metric, SET Laboratories, Inc., Mulino, OR

REFERENCES: Albrecht and Gaffney 1983; Arthur 1983; Basili, Selby, and Phillips 1983; Bulut 1974; Carver 1986; Coulter 1983; Curtis 1980; Elshoff 1976; Fitzsimmons 1978; Funami 1976; Gordon 1976, 1979; Halstead 1972, 1977, 1979; Halstead and Zislis 1973; Henry, Kafura, and Harris 1981; Jones 1978; Kafura and Reddy 1987; Lassez 1981; Li and Cheung 1987; Prather 1984; Shen, Conte, and Dunsmore 1983; Shneiderman 1980; Siyan 1989; Weyuker 1988; Zislis 1973; Zweben 1977.

N_1 - Total Number of Operator Occurrences

This metric quantifies the number of occurrences of operators in a module of source code.

RULE: Count the total number of occurrences of operators in the defined module.

An operator is any word of source code that represents an operation to be performed. (A word of source code is any sequence of characters set off as such by the delimiters defined for the particular language.) The operation is performed on the objects that are the arguments of the operator. In computer program source code, most operators are the keywords that define a program statement. Some keywords have multiple parts that must occur as a set. They constitute one compound operator. The classification of each word of code must be based on an analysis of what that word of code does from a functional point of view.

A module is defined to be a set of code that is to have a measure assigned to it. It must be a well-defined entity in order to ensure that the count is repeatable.

DOMAIN: This measure only applies to that part of source code that implements an algorithm; the code must contain solely operators and operands. An algorithm with less than two operators and one operand is undefined.

RANGE: This metric produces an integer greater than or equal to two, the number of operator occurrences in the most succinct implementation of an algorithm.

NECESSARY CONDITIONS: Every possible combination of characters that constitutes an operator must be defined for the context. The boundaries of a module must be defined for the context.

QUALIFICATIONS: None.

CRITICAL ANALYSIS: This measure is not necessarily strictly monotonic or repeatable. A program reader may increment the count for a word that is not an operator. In this case, an increase in the measure does not represent an increase in the number of operators. Furthermore, a different reader will likely define a different set of operators.

A software tool can also produce non-monotonic results. Although it has well-defined rules for identifying operators, the rules will likely misinterpret some uncommon realization. Whether the count produced by a tool is right or wrong, it will certainly produce a repeatable result.

Despite these problems, if the program reader or the software tool follows carefully defined rules, either can generate highly reliable counts.

SQM RELATIONSHIPS: Clarity, Complexity, Modifiability, Performance, Reliability, Simplicity, Understandability

TOOLS: LOGISCOPE, Verilog USA Inc., Alexandria, VA
PC-Metric, SET Laboratories, Inc., Mulino, OR

REFERENCES: Albrecht and Gaffney 1983; Arthur 1983; Basili, Selby, and Phillips 1983; Bulut 1974; Carver 1986; Coulter 1983; Curtis 1980; Elshoff 1976; Fitzsimmons 1978; Funami 1976; Gordon 1976, 1979; Halstead 1972, 1977, 1979; Halstead and Zislis 1973; Henry, Kafura, and Harris 1981; Jones 1978; Kafura and Reddy 1987; Lassez 1981; Li and Cheung 1987; Prather 1984; Shen, Conte, and Dunsmore 1983; Shneiderman 1980; Siyan 1989; Weyuker 1988; Zislis 1973; Zweben 1977.

N_2 - Total Number of Operand Occurrences

This metric quantifies the number of occurrences of operands in a module of source code.

RULE: Count the total number of occurrences of operands in the defined module.

An operand is any word of source code that represents an argument upon which an operation is to be performed. (A word of source code is any sequence of characters set off as such by the delimiters defined for the particular language.) In computer program source code, operands are the variables and constants of the program. The classification of each word of code must be based on an analysis of what that word of code does, from a functional point of view.

A module is defined to be a set of code that is to have a measure assigned to it. It must be a well-defined entity in order to ensure that the count is repeatable.

DOMAIN: This measure only applies to that part of source code that implements an algorithm; the code must contain solely operators and operands. An algorithm with less than two operators and one operand is undefined.

RANGE: This metric produces an integer greater than or equal to one, the number of operand occurrences in the most succinct implementation of an algorithm.

NECESSARY CONDITIONS: Every possible combination of characters that constitutes an operand must be defined for the context. The boundaries of a module must be defined for the context.

QUALIFICATIONS: None.

CRITICAL ANALYSIS: This measure is not necessarily strictly monotonic or repeatable. A program reader may increment the count for a word that is not an operand. In this case, an increase in the measure does not represent an increase in the number of operands. Furthermore, a different reader will likely define a different set of operands.

A software tool can also produce non-monotonic results. Although it has well-defined rules for identifying operands, the rules will likely misinterpret some uncommon realization. Whether the count produced by a tool is right or wrong, it will certainly produce a repeatable result.

Despite these problems, if the program reader or the software tool follows carefully defined rules, either can generate highly reliable counts.

SQM RELATIONSHIPS: Clarity, Complexity, Modifiability, Performance, Reliability, Simplicity, Understandability

TOOLS: LOGISCOPE, Verilog USA Inc., Alexandria, VA
PC-Metric, SET Laboratories, Inc., Mulino, OR

REFERENCES: Albrecht and Gaffney 1983; Arthur 1983; Basili, Selby, and Phillips 1983; Bulut 1974; Carver 1986; Coulter 1983; Curtis 1980; Elshoff 1976; Fitzsimmons 1978; Funami 1976; Gordon 1976, 1979; Halstead 1972, 1977, 1979; Halstead and Zislis 1973; Henry, Kafura, and Harris 1981; Jones 1978; Kafura and Reddy 1987; Lassez 1981; Li and Cheung 1987; Prather 1984; Shen, Conte, and Dunsmore 1983; Shneiderman 1980; Siyan 1989; Weyuker 1988; Zislis 1973; Zweben 1977.

η - Vocabulary

This metric calculates the total number of unique operators and operands that occur in a module of source code.

RULE: $\eta = \eta_1 + \eta_2$ words

where ç$_1$ is the number of unique operators and ç$_2$ is the number of unique operands. Each of these arguments is defined on a previous data sheet.

A module is defined to be a set of code that is to have a measure assigned to it. It must be a well-defined entity in order to ensure that the count is repeatable.

The Vocabulary represents the number of unique words in a module. Every word of executable code should be listed once in a vocabulary listing. However, when an operator and an operand are identical in form, they still constitute unique words since they serve unique functions. They are distinguished by their context.

DOMAIN: This measure only applies to that part of source code that implements an algorithm; the code must contain solely operators and operands. An algorithm with less than two operators and one operand is undefined.

RANGE: This metric produces an integer greater than or equal to three, the Vocabulary of the most succinct implementation of an algorithm.

NECESSARY CONDITIONS: Every possible combination of characters that constitutes an operator or operand, the criteria for establishing uniqueness, and the boundaries of a module must be defined for the context.

QUALIFICATIONS: The Vocabulary of a module is not necessarily equal to the sum of the Vocabularies of each of its sub-modules. Thus, Vocabulary is not an additive quantity.

CRITICAL ANALYSIS: This measure is not necessarily strictly monotonic or repeatable. The problems are due to the counts that compose it. See the respective software metric data sheets for details.

SQM RELATIONSHIPS: Clarity, Complexity, Modifiability, Performance, Reliability, Simplicity, Understandability

TOOLS: LOGISCOPE, Verilog USA Inc., Alexandria, VA
PC-Metric, SET Laboratories, Inc., Mulino, OR

REFERENCES: Albrecht and Gaffney 1983; Arthur 1983; Basili, Selby, and Phillips 1983; Bulut 1974; Carver 1986; Coulter 1983; Curtis 1980; Elshoff 1976; Fitzsimmons 1978; Funami 1976; Gordon 1976, 1979; Halstead 1972, 1977, 1979; Halstead and Zislis 1973; Henry, Kafura, and Harris 1981; Jones 1978; Kafura and Reddy 1987; Lassez 1981; Li and Cheung 1987; Prather 1984; Shen, Conte, and Dunsmore 1983; Shneiderman 1980; Siyan 1989; Weyuker 1988; Zislis 1973; Zweben 1977.

N - Implementation Length

This metric calculates the total number of occurrences of operators and operands in a module of source code.

RULE: $N = N_1 + N_2$ words

where N_1 is the total number of operator occurrences and N_2 is the total number of operand occurrences. Each of these arguments is defined on a previous data sheet.

A module is defined to be a set of code that is to have a measure assigned to it. It must be a well-defined entity in order to ensure that the count is repeatable.

The Implementation Length represents the count of every word in a module. No word of executable code should be left uncounted.

DOMAIN: This measure only applies to that part of source code that implements an algorithm; the code must contain solely operators and operands. An algorithm with less than two operators and one operand is undefined.

RANGE: This metric produces an integer greater than or equal to three, the Implementation Length of the most succinct implementation of an algorithm.

NECESSARY CONDITIONS: Every possible combination of characters that constitutes an operator or operand must be defined for the context. The boundaries of a module must be defined for the context.

QUALIFICATIONS: None.

CRITICAL ANALYSIS: This measure is not necessarily strictly monotonic or repeatable. The problems are due to the counts that compose it. See the respective software metric data sheets for details.

SQM RELATIONSHIPS: Clarity, Complexity, Modifiability, Performance, Reliability, Simplicity, Understandability

TOOLS: LOGISCOPE, Verilog USA Inc., Alexandria, VA
PC-Metric, SET Laboratories, Inc., Mulino, OR

REFERENCES: Albrecht and Gaffney 1983; Arthur 1983; Basili, Selby, and Phillips 1983; Bulut 1974; Carver 1986; Coulter 1983; Curtis 1980; Elshoff 1976; Fitzsimmons 1978; Funami 1976; Gordon 1976, 1979; Halstead 1972, 1977, 1979; Halstead and Zislis 1973; Henry, Kafura, and Harris 1981; Jones 1978; Kafura and Reddy 1987; Lassez 1981; Li and Cheung 1987; Prather 1984; Shen, Conte, and Dunsmore 1983; Shneiderman 1980; Siyan 1989; Weyuker 1988; Zislis 1973; Zweben 1977

$\hat{N}$ - Estimated Length

This metric estimates the total number of occurrences of operators and operands in a module of source code.

RULE: $\hat{N} = \eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2$ words

where η_1 is the number of unique operators and η_2 is the number of unique operands. Each of these arguments is defined on a previous data sheet.

A module is defined to be a set of code that is to have a measure assigned to it. It must be a well-defined entity in order to ensure that the count is repeatable.

DOMAIN: This measure only applies to that part of source code that implements an algorithm; the code must contain solely operators and operands. An algorithm with less than two operators and one operand is undefined.

RANGE: This metric produces a real number greater than or equal to two, the Estimated Length of the most succinct implementation of an algorithm.

NECESSARY CONDITIONS: Every possible combination of characters that constitutes an operator or operand, the criteria for establishing the uniqueness of operators and operands, and the boundaries of a module must be defined for the context.

QUALIFICATIONS: While the measure is given in terms of a real number, the number of words should be rounded up to the next highest integer.

This estimate assumes that every combination of operators and operands of length η occurs only once in the module. This means that repeated segments of code, as long as η words, are put into subroutines rather than in redundant code. If such redundancy is present in the code, the Implementation Length will be underestimated by the Estimated Length Equation.

The Estimated Length Equation also assumes that operators and operands alternate without variation. Any code that does not follow this convention will likely produce an underestimation of the Implementation Length.

Because the Estimated Length Equation is nonlinear, adding the Estimated Lengths of sub-modules would not be expected to give an accurate Estimated Length for the module that they constitute. Nevertheless, an experiment by Ingojo indicated that the relationship holds fairly well (Halstead 1977).

CRITICAL ANALYSIS: This measure is not necessarily strictly monotonic or repeatable. Some of the problems are due to problems with the counts that compose it. See the respective software metric data sheets for the details of the problems that they contribute to this metric.

Most of the problems with the Estimated Length, however, are due to the simplifying assumptions upon which the estimate is based. Whether or not these assumptions are reasonable, experiments show that yields a close estimate of the Implementation Length, N (Halstead 1977).

SQM RELATIONSHIPS: Clarity, Complexity, Modifiability, Performance, Reliability, Simplicity, Understandability

TOOLS: LOGISCOPE, Verilog USA Inc., Alexandria, VA
PC-Metric, SET Laboratories, Inc., Mulino, OR

REFERENCES: Albrecht and Gaffney 1983; Arthur 1983; Basili, Selby, and Phillips 1983; Bulut 1974; Carver 1986; Coulter 1983; Curtis 1980; Elshoff 1976; Fitzsimmons 1978; Funami 1976; Gordon 1976, 1979; Halstead 1972, 1977, 1979; Halstead and Zislis 1973; Henry, Kafura, and Harris 1981; Jones 1978; Kafura and Reddy 1987; Lassez 1981; Li and Cheung 1987; Prather 1984; Shen, Conte, and Dunsmore 1983; Shneiderman 1980; Siyan 1989; Weyuker 1988; Zislis 1973; Zweben 1977

V - Volume

This metric calculates the number of binary digits required to uniquely represent all of the operators and operands that occur in a module of length, N, and Vocabulary, η. It is not primarily a measure of the size or complexity of the function programmed, but a measure of the size of a particular implementation of a function. This metric should primarily be used to compare the size of the same program written in various languages or by various programmers.

RULE: $V = N\log_2 \eta$ bits

where N is the Implementation Length and η is the Vocabulary. Each of these arguments is defined on a previous data sheet.

A module is defined to be a set of code that is to have a measure assigned to it. It must be a well-defined entity in order to ensure that the count is repeatable.

DOMAIN: This measure only applies to that part of source code that implements an algorithm; the code must contain solely operators and operands. An algorithm with less than two operators and one operand is undefined.

RANGE: This metric produces a real number greater than or equal to about 4.755, the Volume of the most succinct implementation of an algorithm.

NECESSARY CONDITIONS: Every possible combination of characters that constitutes an operator or operand, the criteria for establishing the uniqueness of operators and operands, and the boundaries of a module must be defined for the context.

QUALIFICATIONS: While the measure is given in terms of a real number, the number of bits should be rounded up to the next highest integer.

The Volume of a module is not necessarily equal to the sum of the values of each of its sub-modules. Thus, Volume is not an additive quantity.

CRITICAL ANALYSIS: This measure is not necessarily strictly monotonic or repeatable. The problems are due to the counts that compose it. See the respective software metric data sheets for the details of the problems that they contribute to this metric.

This measure is related to the size of the function being implemented, but it is not a monotonic relationship. A larger algorithm could be implemented so tersely that its volume is less than a smaller algorithm. Conversely, a smaller algorithm could be implemented so verbosely that its volume is greater than a larger algorithm.

SQM RELATIONSHIPS: Clarity, Complexity, Modifiability, Performance, Reliability, Simplicity, Understandability

TOOLS: LOGISCOPE, Verilog USA Inc., Alexandria, VA
PC-Metric, SET Laboratories, Inc., Mulino, OR

REFERENCES: Albrecht and Gaffney 1983; Arthur 1983; Basili, Selby, and Phillips 1983; Bulut 1974; Carver 1986; Coulter 1983; Curtis 1980; Elshoff 1976; Fitzsimmons 1978; Funami 1976; Gordon 1976, 1979; Halstead 1972, 1977, 1979; Halstead and Zislis 1973; Henry, Kafura, and Harris 1981; Jones 1978; Kafura and Reddy 1987; Lassez 1981; Li and Cheung 1987; Prather 1984; Shen, Conte, and Dunsmore 1983; Shneiderman 1980; Siyan 1989; Weyuker 1988; Zislis 1973; Zweben 1977

V^* - Potential Volume

This metric calculates the number of binary digits required to implement the function of a module in its most efficient form, namely two operators (i.e., one that says "do the function" and another that groups the operands with the operator), and all the operands that it requires.

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

where η_2^* is the sum of the number of unique inputs and outputs for the module.

A module is defined to be a set of code that is to have a measure assigned to it. It must be a well-defined entity in order to ensure that the count is repeatable.

DOMAIN: This measure only applies to that part of source code that implements an algorithm; the code must contain solely operators and operands. An algorithm with less than two operators and one operand is undefined. The operand ensures that there is at least one input or output.

RANGE: This metric produces a real number greater than or equal to about 4.755, the Potential Volume of the most succinct implementation of an algorithm.

NECESSARY CONDITIONS: Every possible combination of characters that constitutes an operator or operand, the criteria for establishing the uniqueness of operators and operands, and the boundaries of a module must be defined for the context.

QUALIFICATIONS: While the measure is given in terms of a real number, the number of bits should be rounded up to the next highest integer.

The Potential Volume of a module is not necessarily equal to the sum of the values of each of its sub-modules. Thus, Potential Volume is not an additive quantity.

CRITICAL ANALYSIS: This measure is not necessarily strictly monotonic or repeatable. A program reader may increment the count for some part of the code that is not an input or output. In this case, an increase in the measure does not represent an increase in the size of the function. Furthermore, a different reader will likely define a different set of inputs and outputs.

A software tool can also produce non-monotonic results. Although it has well-defined rules for identifying inputs and outputs, the rules will likely misinterpret some uncommon realization. Whether the count produced by a tool is right or wrong, it will certainly produce a repeatable result. Despite these problems, if the program reader or the software tool follows carefully defined rules, either can generate highly reliable counts.

SQM RELATIONSHIPS: Conciseness, Efficiency

TOOLS: No listing.

REFERENCES: Albrecht and Gaffney 1983; Arthur 1983; Basili, Selby, and Phillips 1983; Bulut 1974; Carver 1986; Coulter 1983; Curtis 1980; Elshoff 1976; Fitzsimmons 1978; Funami 1976; Gordon 1976, 1979; Halstead 1972, 1977, 1979; Halstead and Zislis 1973; Henry, Kafura, and Harris 1981; Jones 1978; Kafura and Reddy 1987; Lassez 1981; Li and Cheung 1987; Prather 1984; Shen, Conte, and Dunsmore 1983; Shneiderman 1980; Siyan 1989; Weyuker 1988; Zislis 1973; Zweben 1977

L - Program Level

This metric calculates how efficiently a module of source code is implemented. It is the ratio of the minimum number of bits it takes to represent an algorithm to the number of bits in a particular implementation. As the Vocabulary and/or Implementation Length decrease, the Program Level increases, i.e., the module is written at a higher level because each word carries more weight. Thus, Program Level indicates the amount of function per word rather than the number of words per function. In this sense, it measures what a program hides rather than what it shows.

RULE: $$L = \frac{V*}{V} \times \frac{\text{bits of function}}{\text{bits of implementation}}$$

where V^* is the Potential Volume and V is the Volume. Each of these arguments is defined on a previous data sheet.

A module is defined to be a set of code that is to have a measure assigned to it. It must be a well-defined entity in order to ensure that the count is repeatable.

DOMAIN: This measure only applies to that part of source code that implements an algorithm; the code must contain solely operators and operands. An algorithm with less than two operators and one operand is undefined. The operand ensures that there is at least one input or output.

RANGE: This metric produces a positive real number less than or equal to one, the Program Level of the most succinct implementation of an algorithm.

NECESSARY CONDITIONS: Every possible combination of characters that constitutes an operator or operand, the criteria for establishing the uniqueness of operators and operands, and the boundaries of a module must be defined for the context.

QUALIFICATIONS: The Program Level of a module is not necessarily equal to the sum of the values for each of its sub-modules. Thus, Program Level is not an additive quantity.

CRITICAL ANALYSIS: This measure is not necessarily strictly monotonic or repeatable. The problems are due to the counts that compose it. See the respective software metric data sheets for the details of the problems that they contribute to this metric.

SQM RELATIONSHIPS: Clarity, Complexity, Modifiability, Performance, Reliability, Simplicity, Understandability

TOOLS: PC-Metric, SET Laboratories, Inc., Mulino, OR

REFERENCES: Albrecht and Gaffney 1983; Arthur 1983; Basili, Selby, and Phillips 1983; Bulut 1974; Carver 1986; Coulter 1983; Curtis 1980; Elshoff 1976; Fitzsimmons 1978; Funami 1976; Gordon 1976, 1979; Halstead 1972, 1977, 1979; Halstead and Zislis 1973; Henry, Kafura, and Harris 1981; Jones 1978; Kafura and Reddy 1987; Lassez 1981; Li and Cheung 1987; Prather 1984; Shen, Conte, and Dunsmore 1983; Shneiderman 1980; Siyan 1989; Weyuker 1988; Zislis 1973; Zweben 1977

$\hat{L}$ - Estimated Program Level

This metric calculates how efficiently a module of source code is implemented. It is an estimate of the Program Level metric that can be used when the number of unique inputs and outputs, ç$_2^*$, is not known. The Estimated Program Level increases as the module is written at a higher level. This occurs as the number of operators decreases to its minimum value of two and as the number of operand occurrences decreases to only one occurrence of each operand.

RULE: $\hat{L} = \frac{2\eta_2}{\eta_1 N_2}$ (unitless)

where η_2 is the number of unique operands, η_1 is the number of unique operators, and N_2 is the total number of operand occurrences. Each of these arguments is defined on a previous data sheet.

A module is defined to be a set of code that is to have a measure assigned to it. It must be a well-defined entity in order to ensure that the count is repeatable.

DOMAIN: This measure only applies to that part of source code that implements an algorithm; the code must contain solely operators and operands. An algorithm with less than two operators and one operand is undefined.

RANGE: This metric produces a positive real number less than or equal to one, the Estimated Program Level of the most succinct implementation of an algorithm.

NECESSARY CONDITIONS: Every possible combination of characters that constitutes an operator or operand, the criteria for establishing the uniqueness of operators and operands, and the boundaries of a module must be defined for the context.

QUALIFICATIONS: The Estimated Program Level of a module is not necessarily equal to the sum of the estimated values for each of its sub-modules. Thus, the Estimated Program Level is not an additive quantity.

CRITICAL ANALYSIS: This measure is not necessarily strictly monotonic or repeatable. Some of the problems are due to the counts that compose it. See the respective software metric data sheets for the details of the problems that they contribute to this metric.

Most of the problems are due to the simplifying assumptions upon which the estimate is based. Whether or not these assumptions are reasonable, experiments show that yields a close estimate of the Program Level, L (Bulut 1974, Elshoff 1976, Halstead 1977, and Ottenstein 1981). They can be used interchangeably.

SQM RELATIONSHIPS: Clarity, Complexity, Modifiability, Performance, Reliability, Simplicity, Understandability

TOOLS: LOGISCOPE, Verilog USA Inc., Alexandria, VA
PC-Metric, SET Laboratories, Inc., Mulino, OR

REFERENCES: Albrecht and Gaffney 1983; Arthur 1983; Basili, Selby, and Phillips 1983; Bulut 1974; Carver 1986; Coulter 1983; Curtis 1980; Elshoff 1976; Fitzsimmons 1978; Funami 1976; Gordon 1976, 1979; Halstead 1972, 1977, 1979; Halstead and Zislis 1973; Henry, Kafura, and Harris 1981; Jones 1978; Kafura and Reddy 1987; Lassez 1981; Li and Cheung 1987; Prather 1984; Shen, Conte, and Dunsmore 1983; Shneiderman 1980; Siyan 1989; Weyuker 1988; Zislis 1973; Zweben 1977

I - Intelligence Content

This metric simply calculates the product of the Estimated Program Level (how efficiently the module is implemented) and the Volume (the size of the implementation) of a module of source code. It represents that constant quantity of information that is present in any implementation of a particular function, in any language, at any level. This quantity is basically an estimated Potential Volume of the program; I and V^* can often be used interchangeably.

$$\text{RULE:} \qquad I = \frac{2\eta_2}{\eta_1 N_2} \times (N_1 + N_2)\log_2(\eta_1 + \eta_2) \quad \text{bits}$$

where η_2 is the number of unique operands, N_1 is the total number of operator occurrences, N_2 is the total number of operand occurrences, and η_1 is the number of unique operators. Each of these arguments is defined on a previous data sheet.

A module is defined to be a set of code that is to have a measure assigned to it. It must be a well-defined entity in order to ensure that the count is repeatable.

DOMAIN: This measure only applies to that part of source code that implements an algorithm; the code must contain solely operators and operands. An algorithm with less than two operators and one operand is undefined.

RANGE: This metric produces a real number greater than or equal to about 4.755, the Intelligence Content of the most succinct implementation of an algorithm.

NECESSARY CONDITIONS: Every possible combination of characters that constitutes an operator or operand, the criteria for establishing the uniqueness of operators and operands, and the boundaries of a module must be defined for the context.

QUALIFICATIONS: While the measure is given in terms of a real number, the number of bits should be rounded up to the next highest integer.

The Intelligence Content of a module is not necessarily equal to the sum of the values for each of its sub-modules. Thus, Intelligence Content is not an additive quantity.

CRITICAL ANALYSIS: This measure is not necessarily strictly monotonic or repeatable. The problems are due to the counts that compose it. See the respective software metric data sheets for the details of the problems that they contribute to this metric.

The Intelligence Content measures the size of an algorithm based on its implementation. The Potential Volume measures the size of an algorithm based on its functional aspects: inputs and outputs. Therefore, when the Intelligence Content is used as an estimate for the Potential Volume, the accuracy of the estimate depends upon the quality of the programming. Redundant and extraneous uses of operators and operands alter the estimate. Halstead has characterized improper usage and has categorized them into six groups that he calls impurity classes (Halstead 1977).

Although it is generally undesirable that the constancy of the Intelligence Content be affected by the presence of program impurities, this effect results in an additional use of this metric. It provides a measure of the amount of impurity in an algorithm implementation. The closer the estimate is to the Potential Volume, the purer the algorithm implementation.

SQM RELATIONSHIPS: Conciseness, Efficiency

TOOLS: LOGISCOPE, Verilog USA Inc., Alexandria, VA
PC-Metric, SET Laboratories, Inc., Mulino, OR

REFERENCES: Albrecht and Gaffney 1983; Arthur 1983; Basili, Selby, and Phillips 1983; Bulut 1974; Carver 1986; Coulter 1983; Curtis 1980; Elshoff 1976; Fitzsimmons 1978; Funami 1976; Gordon 1976, 1979; Halstead 1972, 1977, 1979; Halstead and Zislis 1973; Henry, Kafura, and Harris 1981; Jones 1978; Kafura and Reddy 1987; Lassez 1981; Li and Cheung 1987; Prather 1984; Shen, Conte, and Dunsmore 1983; Shneiderman 1980; Siyan 1989; Weyuker 1988; Zislis 1973; Zweben 1977

E - Programming Effort

This metric calculates the number of elementary mental discriminations done by a programmer to reduce a preconceived algorithm to a module of source code in a language in which the programmer is fluent. It is based on the assumption that, when writing a program, a programmer selects each word of the program by mentally searching a list of words from which to choose. Specifically, the programmer performs a mental binary search of the vocabulary of ç words in order to select the N words used in the implementation. Furthermore, each comparison (or mental discrimination) in the selection process requires an effort related to the difficulty of understanding the program. This program difficulty is supplied by the reciprocal of the Program Level.

RULE: $E = V/L$ discriminations

where V is the Volume and L is the Program Level. Each of these arguments is defined on a previous data sheet. A module is defined to be a set of code that is to have a measure assigned to it.
It must be a well-defined entity in order to ensure that the count is repeatable.

This relationship indicates that the greater the Volume, or the lower the Program Level, the greater the effort required to write a program.

DOMAIN: This measure only applies to that part of source code that implements an algorithm; the code must contain solely operators and operands. An algorithm with less than two operators and one operand is undefined.

RANGE: This metric produces a real number greater than or equal to about 4.755, the Programming Effort for the most succinct implementation of an algorithm.

NECESSARY CONDITIONS: Every possible combination of characters that constitutes an operator or operand, the criteria for establishing the uniqueness of operators and operands, and the boundaries of a module must be defined for the context.

QUALIFICATIONS: While the measure is given in terms of a real number, the number of bits should be rounded up to the next highest integer.

The Program Effort of a module is not necessarily equal to the sum of the values for each of its sub-modules. Thus, Program Effort is not an additive quantity.

CRITICAL ANALYSIS: This measure is not necessarily strictly monotonic or repeatable. The problems are due to the counts that compose it. See the respective software metric data sheets for the details of the problems that they contribute to this metric. Often, the Estimated Program Level is used in place of the Program Level. In this case, any lack of monotonicity in the Programming Effort metric would be produced primarily by the assumptions underlying the Estimated Program Level.

SQM RELATIONSHIPS: Clarity, Complexity, Maintainability, Reliability, Simplicity

TOOLS:TOOLS: LOGISCOPE, Verilog USA Inc., Alexandria, VA
PC-Metric, SET Laboratories, Inc., Mulino, OR

REFERENCES: Albrecht and Gaffney 1983; Arthur 1983; Basili, Selby, and Phillips 1983; Bulut 1974; Carver 1986; Coulter 1983; Curtis 1980; Elshoff 1976; Fitzsimmons 1978; Funami 1976; Gordon 1976, 1979; Halstead 1972, 1977, 1979; Halstead and Zislis 1973; Henry, Kafura, and Harris 1981; Jones 1978; Kafura and Reddy 1987; Lassez 1981; Li and Cheung 1987; Prather 1984; Shen, Conte, and Dunsmore 1983; Shneiderman 1980; Siyan 1989; Weyuker 1988; Zislis 1973; Zweben 1977

$\hat{T}$ - Estimated Programming Time

This metric calculates the time it took a programmer to make the number of elementary mental discriminations performed when a module of source code was programmed. The number of elementary mental discriminations is given by the Programming Effort.

RULE:

$$\hat{\mathbf{T}} = \frac{E}{3600S} = \frac{\eta_1 N_2 (\eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2) \log_2 \eta}{7{,}200 \eta_2 S}$$

where E is the Programming Effort and the Stroud Number, S, is the total number of elementary mental discriminations that a programmer makes per second. This calculation assumes that the programmer's attention is entirely undivided and that the range of values found from psychological experimentation, $5<S<20$, applies to programming activity.

Each of the remaining arguments of this rule is defined on a previous data sheet. A module is defined to be a set of code that is to have a measure assigned to it. It must be a well-defined entity in order to ensure that the count is repeatable.

DOMAIN: This measure only applies to that part of source code that implements an algorithm; the code must contain solely operators and operands. An algorithm with less than two operators and one operand is undefined.

RANGE: This metric produces a real number greater than or equal to about one quarter of a second, the Estimated Programming Time for the most succinct implementation of an algorithm (with S=20).

NECESSARY CONDITIONS: Every possible combination of characters that constitutes an operator or operand, the criteria for establishing the uniqueness of operators and operands, and the boundaries of a module must be defined for the context.

QUALIFICATIONS: The Estimated Programming Time of a module is not necessarily equal to the sum of the estimated values for each of its sub-modules. Thus, Estimated Programming Time is not an additive quantity.

CRITICAL ANALYSIS: This measure is not necessarily strictly monotonic or repeatable. Some of the problems are due to the counts that compose it. See the respective software metric data sheets for the details of the problems that they contribute to this metric.

Most of the problems are due to the variability associated with the Stroud Number. It is not necessarily monotonic or repeatable for all programming contexts. It is not necessarily monotonic because a programmer's knowledge in one discipline may result in a shorter programming time for a program with a greater value of Program

Effort than for a program with a lesser value of Program Effort. This could occur when the former program is in a discipline with which the programmer is familiar, but the latter program is not. It is not necessarily repeatable because on some days a programmer will concentrate better than on others.

SQM RELATIONSHIPS: Performance

TOOLS:TOOLS: LOGISCOPE, Verilog USA Inc., Alexandria, VA
PC-Metric, SET Laboratories, Inc., Mulino, OR

REFERENCES: Albrecht and Gaffney 1983; Arthur 1983; Basili, Selby, and Phillips 1983; Bulut 1974; Carver 1986; Coulter 1983; Curtis 1980; Elshoff 1976; Fitzsimmons 1978; Funami 1976; Gordon 1976, 1979; Halstead 1972, 1977, 1979; Halstead and Zislis 1973; Henry, Kafura, and Harris 1981; Jones 1978; Kafura and Reddy 1987; Lassez 1981; Li and Cheung 1987; Prather 1984; Shen, Conte, and Dunsmore 1983; Shneiderman 1980; Siyan 1989; Weyuker 1988; Zislis 1973; Zweben 1977

λ - Language Level

This metric calculates the efficiency with which algorithms can be implemented in a particular language. For any module of source code written in the particular language, the Language Level should be a constant, regardless of the Volume of the module or the Program Level at which it is written.

RULE: $\lambda = L^2V$

where L is the Program Level and V is the Volume. Each of these arguments is defined on a previous data sheet.

A module is defined to be a set of code that is to have a measure assigned to it. It must be a well-defined entity in order to ensure that the count is repeatable.

DOMAIN: This measure only applies to that part of source code that implements an algorithm; the code must contain solely operators and operands. An algorithm with less than two operators and one operand is undefined.

RANGE: This metric produces a positive real number less than or equal to the Volume of the module.

NECESSARY CONDITIONS: Every possible combination of characters that constitutes an operator or operand, the criteria for establishing the uniqueness of operators and operands, and the boundaries of a module must be defined for the context.

QUALIFICATIONS: The Language Level of a module is not necessarily equal to the sum of the values for each of its sub-modules. Thus, Language Level is not an additive quantity.

CRITICAL ANALYSIS: This measure is not necessarily strictly monotonic or repeatable. The problems are due to the counts that compose it. See the respective software metric data sheets for the details of the problems that they contribute to this metric.

Often, the Estimated Program Level is used in place of the Program Level. In this case, any lack of monotonicity in the Language Level would be produced primarily by the assumptions underlying the Estimated Program Level.

SQM RELATIONSHIPS: None.

TOOLS:TOOLS: LOGISCOPE, Verilog USA Inc., Alexandria, VA
PC-Metric, SET Laboratories, Inc., Mulino, OR

REFERENCES: Albrecht and Gaffney 1983; Arthur 1983; Basili, Selby, and Phillips 1983; Bulut 1974; Carver 1986; Coulter 1983; Curtis 1980; Elshoff 1976; Fitzsimmons 1978; Funami 1976; Gordon 1976, 1979; Halstead 1972, 1977, 1979; Halstead and Zislis 1973; Henry, Kafura, and Harris 1981; Jones 1978; Kafura and Reddy 1987; Lassez 1981; Li and Cheung 1987; Prather 1984; Shen, Conte, and Dunsmore 1983; Shneiderman 1980; Siyan 1989; Weyuker 1988; Zislis 1973; Zweben 1977

$\hat{B}$ - Number of Bugs

This metric calculates an estimate of the number of bugs that a module of source code contains at the time of measurement.

RULE: $\hat{B} = V/E_0$ bugs

where V is the Volume and E_0 is the number of discriminations per bug. The Volume is as defined on a previous data sheet. E_0 is determined by an evaluation of a programmer's previous work. Experimentation has found that 3,200 discriminations per bug is a typical value.

A module is defined to be a set of code that is to have a measure assigned to it. It must be a well-defined entity in order to ensure that the count is repeatable.

DOMAIN: This measure only applies to that part of source code that implements an algorithm; the code must contain solely operators and operands. An algorithm with less than two operators and one operand is undefined.

RANGE: This metric produces a positive real number.

NECESSARY CONDITIONS: Every possible combination of characters that constitutes an operator or operand, the criteria for establishing the uniqueness of operators and operands, and the boundaries of a module must be defined for the context.

QUALIFICATIONS: While the measure is given in terms of a real number, the number of bugs should be rounded up to the next highest integer.

The Number of Bugs in a module is not necessarily equal to the sum of the values for each of its sub-modules. Thus, Number of Bugs is not an additive quantity.

CRITICAL ANALYSIS: This measure is not necessarily strictly monotonic or repeatable. Some of the problems are due to the counts that compose it. See the respective software metric data sheets for the details of the problems that they contribute to this metric.

Most of the problems are due to the variability associated with E_0. It is not necessarily monotonic or repeatable for all programming contexts. It is not necessarily monotonic because a programmer's knowledge in one discipline may result in fewer bugs for a program with a greater Volume than for a program with a lesser Volume. This could occur when the former program is in a discipline with which the programmer is familiar, but the latter program is not. It is not necessarily repeatable because on some days a programmer will concentrate better than on others.

SQM RELATIONSHIPS: Maintainability

TOOLS:TOOLS: LOGISCOPE, Verilog USA Inc., Alexandria, VA
PC-Metric, SET Laboratories, Inc., Mulino, OR

REFERENCES: Albrecht and Gaffney 1983; Arthur 1983; Basili, Selby, and Phillips 1983; Bulut 1974; Carver 1986; Coulter 1983; Curtis 1980; Elshoff 1976; Fitzsimmons 1978; Funami 1976; Gordon 1976, 1979; Halstead 1972, 1977, 1979; Halstead and Zislis 1973; Henry, Kafura, and Harris 1981; Jones 1978; Kafura and Reddy 1987; Lassez 1981; Li and Cheung 1987; Prather 1984; Shen, Conte, and Dunsmore 1983; Shneiderman 1980; Siyan 1989; Weyuker 1988; Zislis 1973; Zweben 1977

Information Flow

This metric calculates the complexity of a module of source code based on the size of the module and the flow of information within it.

RULE: The Information Flow Complexity (IFC) of a module is the sum of the IFCs for each procedure that composes it. The IFC for each procedure is calculated as follows:

$$\text{IFC} = \text{Length x (Fan-in x Fan-out)}^2$$

where Length is given by the number of lines of source code in the procedure, Fan-in is the number of local data flows that terminate at the procedure, and Fan-out is the number of local data flows that emanate from the procedure.

A module is defined as the set of procedures that either directly update or directly retrieve information from the particular data structure with which the module is associated.

Local data flow is data that is passed from procedure to procedure without going through the data structure.

DOMAIN: This measure only applies to source code that implements an algorithm; the code must perform some function that has inputs and outputs.

RANGE: This metric produces an integer greater than or equal to one, the IFC of a one statement procedure that has one input and one output.

NECESSARY CONDITIONS: The criteria for determining the local inputs and outputs must be defined for the context. The criteria for delimiting procedures and modules must be defined for the context.

QUALIFICATIONS: The IFC of a module is not necessarily equal to the IFC of each of its submodules. Thus, IFC is not an additive quantity.

CRITICAL ANALYSIS: This measure is not necessarily strictly monotonic or repeatable. A program reader may erroneously count local inputs, local outputs, or lines of source code. In this case, an increase in the measure does not represent an increase in the amount of information flow. Furthermore, a different reader will likely judge the attributes differently.

A problem with repeatability can also result from the arbitrariness of establishing module boundaries. The larger the module, the greater will be the IFC. Furthermore, as the module boundaries shift, the fan-in and fan-out counts will change. It is important that all measurements be based on an unambiguous specification for establishing module boundaries.

A software tool can also produce non-monotonic results. Although it has well-defined rules for identifying the inputs and outputs and for counting the lines of code, the rules will likely misinterpret some uncommon realization. Whether the count produced by a tool is right or wrong, it will certainly produce a repeatable result.

Despite these problems, if the program reader or the software tool follows carefully defined rules, either can generate highly reliable counts.

The IFC metric uses the source code to deduce the characteristics of the data flow of the algorithm solved by the program. The rules used for this step are dependent on the programming language and technology used. But it is primarily the data flow that is measured, not the program. Although the process for deducing the data flow is implementation dependent, the resulting value of the IFC metric has the advantage of being highly language independent, as well as highly independent of programmer style or experience. It is even highly independent of any changes in technology. Once the data flow complexity of an algorithm is calculated, most any program in any language that performs the same algorithm, whether poorly or well written, will have the same IFC value.

This complexity measure has been shown to represent the decrease in complexity that sometimes can be made by adding lines of code (Henry and Kafura 1984). It appears to measure a dimension of complexity somewhat different from that measured by Halstead's and McCabe's metrics (Kafura and Reddy 1987).

SQM RELATIONSHIPS: Complexity, Maintainability, Modifiability, Performance, Reliability, Simplicity, Understandability

TOOLS: No listing.

REFERENCES: Henry and Kafura 1984; Henry, Kafura, and Harris 1981; Kafura and Reddy 1987

v(G) - Cyclomatic Complexity

This metric calculates the number of linearly independent program flow paths in the basis set of paths for a module of source code. Every possible path of program flow through a module can be represented as a linear combination of some subset of the basis set of paths.

RULE: $v(G) = e - n + 2p$

where e is the number of edges and n is the number of nodes in the program flow control graphs of all the modules. The variable p is the number of modules.

A module is defined to be a set of connected code that is to have a measure assigned to it. It must be a well-defined entity in order to ensure that the count is repeatable. To be connected, there must exist an unbroken chain of nodes and edges between any two nodes. A subroutine is a module, distinct from its calling module, unless the graph is drawn with the subroutine code substituted for the calling statement.

To draw the control graph, first, block the code into predicate nodes, collecting nodes, and function nodes. A predicate node is a block of code ending with a statement that branches to multiple locations. A collecting node begins with a labeled statement to which flow branches. When a predicate node is labeled, it is both a predicate node and a collecting node. Each strictly sequential block of statements between any two predicate or collecting nodes compose a function node.

Once the code is blocked, graphically represent each block with a circle and the flow of control between them with lines. In this graph, the circles are nodes and the lines are edges. The function nodes can be omitted from the graph without affecting the value of the Cyclomatic Complexity.

DOMAIN: This metric can be applied to any source code that can be blocked into nodes.

RANGE: This metric produces an integer greater than or equal to one, the Cyclomatic Complexity of code with purely sequential flow.

NECESSARY CONDITIONS: Every possible combination of characters that constitutes each type of node must be defined for the context. The boundaries of a module must be defined for the context.

QUALIFICATIONS: None.

CRITICAL ANALYSIS: This measure is not necessarily strictly monotonic or repeatable. A program reader may miss a node when drawing the graph. In this case, a net increase in the edge-node difference is not represented by an increase in the measure. Furthermore, a different reader will likely judge the nodes differently for the same code.

A software tool can also produce non-monotonic results. Although it has well-defined rules for identifying nodes, the rules will likely misinterpret some uncommon realization. Whether the count produced by a tool is right or wrong, it will certainly produce a repeatable result.

Despite these problems, if the program reader or the software tool follows carefully defined rules, either can generate highly reliable counts.

SQM RELATIONSHIPS: Complexity, Maintainability, Modifiability, Modularity, Performance, Reliability, Simplicity, Testability, Understandability

TOOLS:
Analysis of Complexity Tool (ACT), McCabe & Assocs., Columbia, MD
Battlemap Analysis Tool (BAT), McCabe & Assocs., Columbia, MD
LOGISCOPE, Verilog USA Inc., Alexandria, VA
PC-Metric, SET Laboratories, Inc., Mulino, OR

REFERENCES: Arthur 1983; Basili 1983; Basili and Hutchens 1983; Basili, Selby, and Phillips 1983; Carver 1986; Ct et al. 1988; Crawford, McIntosh, and Pregibon 1985; Curtis 1980; Elshoff 1982, 1983; Evangelist 1983; Gaffney 1981; Harrison 1984; Henry, Kafura, and Harris 1981; Kafura and Reddy 1987; Li and Cheung 1987; Lind and Vairavan 1989; Mannino, Stoddard, and Sudduth 1990; McCabe 1976, 1982, 1989; McCabe and Butler 1989; McClure 1978; Myers 1977; Prather 1983, 1984; Ramamurthy and Melton 1988; Schneidewind and Hoffmann 1979; Shneiderman 1980; Siyan 1989; Walsh 1979; Ward 1989; Weyuker 1988

v(G) - Cyclomatic Complexity
Alternate Rule 1

This metric quantifies the number of linearly independent program flow paths in the basis set of paths for a module of source code. Every possible path of program flow through a module can be represented as a linear combination of some subset of the basis set of paths.

ALTERNATIVE RULE 1: The Cyclomatic Complexity of a module is equal to the number of regions delineated by its control graph, with an additional edge from the last node to the first. The control graph must be drawn with no crossing edges: it must be a planar graph.

A module is defined to be a set of connected code that is to have a measure assigned to it. It must be a well-defined entity in order to ensure that the count is repeatable. To be connected, there must exist an unbroken chain of nodes and edges between any two nodes. A subroutine is a module, distinct from its calling module, unless the graph is drawn with the subroutine code substituted for the calling statement.

To draw the control graph, first, block the code into predicate nodes, collecting nodes, and function nodes. A predicate node is a block of code ending with a statement that branches to multiple locations. A collecting node begins with a labeled statement to which flow branches. When a predicate node is labeled, it is both a predicate node and a collecting node. Each strictly sequential block of statements between any two predicate or collecting nodes compose a function node.

Once the code is blocked, graphically represent each block with a circle and the flow of control between them with lines. In this graph, the circles are nodes and the lines are edges. Also draw an edge from the exit node to the entrance node. The function nodes can be omitted from the graph without affecting the value of the Cyclomatic Complexity. The region surrounding the graph must also be counted as a region.

DOMAIN: This metric can be applied to any source code that can be blocked into nodes and can be drawn in a planar graph.

RANGE: This metric produces an integer greater than or equal to one, the Cyclomatic Complexity of code with purely sequential flow.

NECESSARY CONDITIONS: Every possible combination of characters that constitutes each type of node must be defined for the context. The boundaries of a module must be defined for the context.

QUALIFICATIONS: The control graphs of all structured and many unstructured programs can be drawn as planar graphs. Sometimes the control graph of unstructured code cannot be drawn as a planar graph.

CRITICAL ANALYSIS: This measure is not necessarily strictly monotonic or repeatable. A program reader may miss a node when drawing the graph. In this case, a net increase in the edge-node difference is not represented by an increase in the measure. Furthermore, a different reader will likely judge the nodes differently for the same code.

A software tool can also produce non-monotonic results. Although it has well-defined rules for identifying nodes, the rules will likely misinterpret some uncommon realization. Whether the count produced by a tool is right or wrong, it will certainly produce a repeatable result.

Despite these problems, if the program reader or the software tool follows carefully defined rules, either can generate highly reliable counts.

SQM RELATIONSHIPS: Complexity, Maintainability, Modifiability, Modularity, Performance, Reliability, Simplicity, Testability, Understandability

TOOLS: Analysis of Complexity Tool (ACT), McCabe & Assocs., Columbia, MD
Battlemap Analysis Tool (BAT), McCabe & Assocs., Columbia, MD
LOGISCOPE, Verilog USA Inc., Alexandria, VA
PC-Metric, SET Laboratories, Inc., Mulino, OR

REFERENCES: Arthur 1983; Basili 1983; Basili and Hutchens 1983; Basili, Selby, and Phillips 1983; Carver 1986; Ct et al. 1988; Crawford, McIntosh, and Pregibon 1985; Curtis 1980; Elshoff 1982, 1983; Evangelist 1983; Gaffney 1981; Harrison 1984; Henry, Kafura, and Harris 1981; Kafura and Reddy 1987; Li and Cheung 1987; Lind and Vairavan 1989; Mannino, Stoddard, and Sudduth 1990; McCabe 1976, 1982, 1989; McCabe and Butler 1989; McClure 1978; Myers 1977; Prather 1983, 1984; Ramamurthy and Melton 1988; Schneidewind and Hoffmann 1979; Shneiderman 1980; Siyan 1989; Walsh 1979; Ward 1989; Weyuker 1988

v(G) - Cyclomatic Complexity

Alternate Rule 2

This metric quantifies the number of linearly independent program flow paths in the basis set of paths for a module of source code. Every possible path of program flow through a module can be represented as a linear combination of some subset of the basis set of paths.

ALTERNATIVE RULE 2: $v(G) = \pi + 1$

where ð is the number of control flow branch conditions in the module.

A module is defined to be a set of connected code that is to have a measure assigned to it. It must be a well-defined entity in order to ensure that the count is repeatable. To be connected, there must exist an unbroken chain of nodes and edges between any two nodes. A subroutine is a module, distinct from its calling module, unless the graph is drawn with the subroutine code substituted for the calling statement.

This formula for structured programs requires that for every predicate node there is exactly one collecting node, and that the program has unique entry and exit nodes. Exceptions to the rules always alter v(G). As long as there are relatively few exceptions to these rules, the Cyclomatic Complexity will be not be substantially affected. If there are a significant number of exceptions, the impact of multiple exits can be corrected by computing the following:

$$v(G) = \pi - s + 2$$

where s is the number of exits from the module (Harrison 1984). Of course, counting the number of exits requires additional analysis. If the code is primarily structured, the inaccuracy of the original equation is probably not significant enough to warrant the extra effort.

DOMAIN: This metric can be applied to any executable source code.

RANGE: This metric produces an integer greater than or equal to one, the Cyclomatic Complexity of code with purely sequential flow.

NECESSARY CONDITIONS: Every possible combination of characters that constitutes a branch condition must be defined for the context. The boundaries of a module must be defined for the context.

QUALIFICATIONS: This measure counts all conditions present, whether necessary or not. For example, code may contain a condition in a context where it will always be true. The Cyclomatic Complexity counts the number of conditions present in the algorithm, as implemented.

The Cyclomatic Complexity of a collection of independent modules, as in a main program and its subroutines, is simply the sum of the Cyclomatic Complexities for each module. For this rule, it is improper to calculate the total Cyclomatic Complexity by analyzing all the modules as if they were one continuous sequence of code. In this case, the value is deflated by the number of modules in the collection minus one.

CRITICAL ANALYSIS: This measure is not necessarily strictly monotonic or repeatable. A program reader may miss a condition when reading the code. In this case, a net increase in the number of branch conditions is not represented by an increase in the measure. Furthermore, a different reader will likely judge the conditions differently for the same code.

A software tool can also produce non-monotonic results. Although it has well-defined rules for identifying branch conditions, the rules will likely misinterpret some uncommon realization. Whether the count produced by a tool is right or wrong, it will certainly produce a repeatable result.

Despite these problems, if the program reader or the software tool follows carefully defined rules, either can generate highly reliable counts.

SQM RELATIONSHIPS: Complexity, Maintainability, Modifiability, Modularity, Performance, Reliability, Simplicity, Testability, Understandability

TOOLS:
Analysis of Complexity Tool (ACT), McCabe & Assocs., Columbia, MD
Battlemap Analysis Tool (BAT), McCabe & Assocs., Columbia, MD
LOGISCOPE, Verilog USA Inc., Alexandria, VA
PC-Metric, SET Laboratories, Inc., Mulino, OR

REFERENCES: Arthur 1983; Basili 1983; Basili and Hutchens 1983; Basili, Selby, and Phillips 1983; Carver 1986; Ct et al. 1988; Crawford, McIntosh, and Pregibon 1985; Curtis 1980; Elshoff 1982, 1983; Evangelist 1983; Gaffney 1981; Harrison 1984; Henry, Kafura, and Harris 1981; Kafura and Reddy 1987; Li and Cheung 1987; Lind and Vairavan 1989; Mannino, Stoddard, and Sudduth 1990; McCabe 1976, 1982, 1989; McCabe and Butler 1989; McClure 1978; Myers 1977; Prather 1983, 1984; Ramamurthy and Melton 1988; Schneidewind and Hoffmann 1979; Shneiderman 1980; Siyan 1989; Walsh 1979; Ward 1989; Weyuker 1988

ev(G) - Essential Complexity

This metric calculates the amount of unstructured code in a module.

RULE: ev(G) = v(G) - m

where m is the number of proper subgraphs with unique entry and exit nodes.

A proper subgraph is any part of the graph that consists of only a "SEQUENCE" construct, an "IF-THEN-ELSE" construct, a "DO-UNTIL" construct, a "DO-WHILE" construct, or a "CASE" construct.

A module is defined to be a set of connected code that is to have a measure assigned to it. It must be a well-defined entity in order to ensure that the count is repeatable. To be connected, there must exist an unbroken chain of nodes and edges between any two nodes. A subroutine is a module, distinct from its calling module, unless the graph is drawn with the subroutine code substituted for the calling statement.

The smaller the value of ev(G), the greater the proportion of structured code. The Essential Complexity of a structured module is one. Greater values for the Essential Complexity of a module indicate the size of the nonstructured portion of the code; that code that does not conform to the proper subgraph structure (McCabe 1976).

Like the Cyclomatic Complexity, this measure also reflects the state of the algorithm, as implemented. It does not indicate whether the conditions, as implemented, are necessary for the algorithm.

To draw the control graph, first block the code into predicate nodes, processing nodes, and collecting nodes. A predicate node is a block of code ending with a statement that branches to multiple locations. A collecting node begins with a labeled statement to which flow branches. When a predicate node is labeled, it is both a predicate node and a collecting node. Each strictly sequential block of statements between any two predicate or collecting nodes compose a function node.

Once the code is blocked, graphically represent each block with a circle and the flow of control between them with lines. In this graph, the circles are nodes and the lines are edges. The function nodes can be omitted from the graph without affecting the value of the Cyclomatic Complexity.

DOMAIN: This metric can be applied to any source code that can be blocked into nodes.

RANGE: This metric produces an integer greater than or equal to one, the Essential Complexity of a totally structured module.

NECESSARY CONDITIONS: Every possible combination of characters that constitutes each type of node must be defined for the context. The boundaries of a module must be defined for the context.

QUALIFICATIONS: This measure only applies to executable source code.

CRITICAL ANALYSIS: This measure is not necessarily strictly monotonic or repeatable. A program reader may miss a node when drawing the graph. In this case, a net increase in the edge-node difference is not represented by an increase in the measure. Furthermore, a different reader will likely judge the nodes differently for the same code.

A software tool can also produce non-monotonic results. Although it has well-defined rules for identifying nodes, the rules will likely misinterpret some uncommon realization. Whether the count produced by a tool is right or wrong, it will certainly produce a repeatable result.

Despite these problems, if the program reader or the software tool follows carefully defined rules, either can generate highly reliable counts.

SQM RELATIONSHIPS: Complexity, Conciseness, Efficiency, Modularity, Performance, Reliability, Simplicity, Understandability

TOOLS:
Analysis of Complexity Tool (ACT), McCabe & Assocs., Columbia, MD
Battlemap Analysis Tool (BAT), McCabe & Assocs., Columbia, MD
LOGISCOPE, Verilog USA Inc., Alexandria, VA

REFERENCES: Arthur 1983; Basili 1983; Basili and Hutchens 1983; Basili, Selby, and Phillips 1983; Carver 1986; Ct et al. 1988; Crawford, McIntosh, and Pregibon 1985; Curtis 1980; Elshoff 1982, 1983; Evangelist 1983; Gaffney 1981; Harrison 1984; Henry, Kafura, and Harris 1981; Kafura and Reddy 1987; Li and Cheung 1987; Lind and Vairavan 1989; Mannino, Stoddard, and Sudduth 1990; McCabe 1976, 1982, 1989; McCabe and Butler 1989; McClure 1978; Myers 1977; Prather 1983, 1984; Ramamurthy and Melton 1988; Schneidewind and Hoffmann 1979; Shneiderman 1980; Siyan 1989; Walsh 1979; Ward 1989; Weyuker 1988

AP.3(2) - Architecture Standardization

This metric calculates the ratio of assembly language lines of code to total lines of source code in the modules of a Computer Software Configuration Item (CSCI).

RULE: For each module of the CSCI, answer the question:

"How many non-HOL [sic] lines of code, excluding comments (e.g., assembly language)?" (Bowen, Wigle, and Tsai 1985)

"Divide the total number of non-HOL (High Order Language) [sic] lines of code by the total number of source lines of code." (Bowen, Wigle, and Tsai 1985)

A CSCI is a set of code as defined in MIL-STD-483 and as used in DOD-STD-2167.

A module (or unit) is defined as a set of code to which a measure is assigned. It must be a well-defined entity in order to ensure that the count is repeatable.

DOMAIN: This metric can be applied to any module of source code that performs a function.

RANGE: This metric produces a real number between zero and one.

NECESSARY CONDITIONS: The combinations of characters that constitute each type of code line must be defined for the context. The boundaries of a module must be defined for the context.

QUALIFICATIONS: None.

CRITICAL ANALYSIS: This measure is not necessarily strictly monotonic or repeatable. A program reader may erroneously judge that a line of code is not assembly code. In this case, an increase in the measure does not represent an increase in the number of assembly language lines of code. Furthermore, a different reader will likely judge the lines of code differently for the same code.

A software tool can also produce non-monotonic results. Although it has well-defined rules for identifying the types of source code, the rules will likely misinterpret some uncommon implementation. Whether the count produced by a tool is right or wrong, the tool will certainly produce a repeatable result.

Despite these problems, if the program reader or the software tool follows carefully defined rules, either can generate highly reliable counts.

SQM RELATIONSHIPS: Reusability

TOOLS:
***, METRIQS, San Juan Capistrano, CA
AdaMAT, Dynamics Research Corporation, Andover, MA
AMS, Rome Air Development Center, Griffiss AFB, NY
(*** This tool is used internally to support their consulting service)

REFERENCES: Boehm et al. 1978; Bowen, Wigle, and Tsai 1985; Lasky, Kaminsky, and Boaz 1989; McCall, Richards, and Walters 1977; Millman and Curtis 1980; Murine 1983, 1985a, 1985b, 1986, 1988; Pierce, Hartley, and Worrells 1987; Shneiderman 1980; Sunazuka, Azuma, and Yamagishi 1985; Warthman 1987

CP.1(2) - Completeness Checklist

This metric calculates the ratio of identified data references that are fully documented to total identified data references in the modules of a Computer Software Configuration Item (CSCI).

RULE: For each module of the CSCI, answer the question:

"How many identified data references are documented with regard to source, meaning, and format?" (Bowen, Wigle, and Tsai 1985)

"Divide the total number of identified and fully documented data references by the total number of identified data references." (Bowen, Wigle, and Tsai 1985)

A CSCI is a set of code as defined in MIL-STD-483 and as used in DOD-STD-2167.

A module (or unit) is defined as a set of code to which a measure is assigned. It must be a well-defined entity in order to ensure that the count is repeatable.

DOMAIN: This metric can be applied to any module of source code that contains data references.

RANGE: This metric produces a real number between zero and one.

NECESSARY CONDITIONS: The combinations of characters that constitute an identified data reference, the combinations of characters that satisfy the source, meaning, and format, and the boundaries of a module must be defined for the context.

QUALIFICATIONS: None.

CRITICAL ANALYSIS: This measure is not necessarily strictly monotonic or repeatable. A program reader may erroneously judge that a description in a comment satisfies the requirement. In this case, an increase in the measure does not represent an increase in the completeness. Furthermore, a different reader will likely judge the sufficiency of the documentation differently for the same code. At best, it only establishes that the data reference was documented, not how well it was documented.

A software tool can also produce non-monotonic results. Although it has well-defined rules for evaluating the descriptions, the rules will likely misinterpret some uncommon implementation. Whether the count produced by a tool is right or wrong, the tool will certainly produce a repeatable result.

Despite these problems, if the program reader or the software tool follows carefully defined rules, either can generate highly reliable counts.

SQM RELATIONSHIPS: Correctness

TOOLS: ***, METRIQS, San Juan Capistrano, CA
AdaMAT, Dynamics Research Corporation, Andover, MA
AMS, Rome Air Development Center, Griffiss AFB, NY
(*** This tool is used internally to support their consulting service)

REFERENCES: Boehm et al. 1978; Bowen, Wigle, and Tsai 1985; Lasky, Kaminsky, and Boaz 1989; McCall, Richards, and Walters 1977; Millman and Curtis 1980; Murine 1983, 1985a, 1985b, 1986, 1988; Pierce, Hartley, and Worrells 1987; Shneiderman 1980; Sunazuka, Azuma, and Yamagishi 1985; Warthman 1987

SOFTWARE METRIC DATA SHEET RADC Family

SD.2(1) - Effectiveness of Comments

This metric calculates, for a single Computer Software Configuration Item (CSCI), the ratio of modules in which header comments are complete to total applicable modules.

RULE: For each module of the CSCI, answer the question:

"Are there prologue comments which contain all information in accordance with the established standard?" (Bowen, Wigle, and Tsai 1985)

Divide the total number of affirmative answers by the number of modules in the set to which the question applied.

A CSCI is a set of code as defined in MIL-STD-483 and as used in DOD-STD-2167.

A module (or unit) is defined as a set of code to which a measure is assigned. It must be a well-defined entity in order to ensure that the count is repeatable.

DOMAIN: This metric can be applied to any source code for which a standard has been set for header comments.

RANGE: This metric produces a real number between zero and one.

NECESSARY CONDITIONS: The combinations of characters that identify a prologue comment, the combinations of characters that must constitute each required piece of information, and the boundaries of a module must be defined for the context.

QUALIFICATIONS: This question requires that the comment exist and that it contain the specified information. The only case in which the question does not apply is for a module for which no prologue comments are required.

CRITICAL ANALYSIS: This measure is not necessarily strictly monotonic or repeatable. A program reader may erroneously judge that a piece of information in a comment satisfies the requirement. In this case, an increase in the measure does not represent an increase in the comment effectiveness. Furthermore, a different reader will likely judge the sufficiency of the comments differently for the same code.

A software tool can also produce non-monotonic results. Although it has well-defined rules for evaluating the information, the rules will likely misinterpret some uncommon implementation. Whether the count produced by a tool is right or wrong, the tool will certainly produce a repeatable result.

Despite these problems, if the program reader or the software tool follows carefully defined rules, either can generate highly reliable counts.

SQM RELATIONSHIPS: Expandability, Flexibility, Maintainability, Portability, Reusability, Verifiability

TOOLS: ***, METRIQS, San Juan Capistrano, CA
AdaMAT, Dynamics Research Corporation, Andover, MA
AMS, Rome Air Development Center, Griffiss AFB, NY
(*** This tool is used internally to support their consulting service.)

REFERENCES: Boehm et al. 1978; Bowen, Wigle, and Tsai 1985; Lasky, Kaminsky, and Boaz 1990; McCall, Richards, and Walters 1977; Millman and Curtis 1980; Murine 1983, 1985a, 1985b, 1986, 1988; Pierce, Hartley, and Worrells 1987; Shneiderman 1980; Sunazuka, Azuma, and Yamagishi 1985; Warthman 1987

APPENDIX B - SOFTWARE QUALITY FACTOR DATA SHEETS

This appendix contains individual data sheets on each of the most prevalent software quality factors encountered in this study. The data sheets are a summary of the concepts, constraints, criticisms, and other data needed to understand and critically evaluate the application of Software Quality Metrics (SQM) that measure these quality factors.

Although the data sheets are not primarily intended as a guide for applying SQM, if some of these data sheets are combined with the Software Metric Data Sheets that they reference, there will be sufficient information to fully calculate an SQM. On the other hand, some of the SQM are so extensive that the data sheets could not encompass all the necessary parts. See the references listed on these data sheets for complete coverage of a particular quality factor. See the references listed on the Software Metric Data Sheets for complete coverage of a particular software metric that an SQM uses to measure a quality factor.

Each data sheet covers only one factor, and they all contain the same fields of information. The fields are defined only as they apply to measuring source code. Many of the metrics can be applied more broadly, but the other possibilities are not addressed here. The data sheets are arranged alphabetically by factor name.

The "CLAIM" field is filled by the particular quality factor description or definition that is the most comprehensive, without being too general, and the most independent. The goal of identifying quality factors is to come up with a set of factors that are independent of one another and that comprehensively cover the full range of quality factors for software.

The "SQM RELATIONSHIPS" field lists those software metrics that are known to be related to the particular quality factor. This serves as a cross-reference to the applicable Software Metric Data Sheet of appendix A. However, not all of the referenced software metrics are addressed in appendix A. See the articles listed in the bibliography for further information.

The "TOOLS" information is supplied on an as-available basis. It is not to be taken as a comprehensive list of tools available. Nor is the information to be taken as a recommendation.

SOFTWARE QUALITY FACTOR DATA SHEET

Accuracy

This quality factor addresses the concern that programs provide the precision required for each output. Accuracy is important because most computer manipulations are not exact, but are limited approximations.

CLAIM: Some researchers claim that

"A software product possesses accuracy to the extent that its outputs are sufficiently precise to satisfy their intended use." (Boehm et al. 1978).

DOMAIN: This factor only applies to source code for which the required precision has been explicitly stated for each output.

NECESSARY CONDITIONS: The required precision must be defined for the context. The outputs must be defined for the context.

QUALIFICATIONS: None.

CRITICAL ANALYSIS: This factor does not address whether a calculation or output should or should not be present. The former is the concern of completeness, the latter of conciseness. Nor does it address whether the proper quantity is being calculated or output. That is the concern of correctness.

SQM RELATIONSHIPS: RADC's AC.1(7)

TOOLS: ***, METRIQS, San Juan Capistrano, CA
AdaMAT, Dynamics Research Corporation, Andover, MA
AMS, Rome Air Development Center, Griffiss AFB, NY
(*** This tool is used internally to support their consulting service.)

REFERENCES: Boehm et al. 1978; Bowen, Wigle, and Tsai 1985; Lasky, Kaminsky, and Boaz 1989; McCall, Richards, and Walters 1977; Millman and Curtis 1980; Murine 1983, 1985a, 1985b, 1986, 1988; Pierce, Hartley, and Worrells 1987; Shneiderman 1980; Sunazuka, Azuma, and Yamagishi 1985; Warthman 1987

SOFTWARE QUALITY FACTOR DATA SHEET

Clarity

This quality factor addresses the concern that programs be easily understood by people. Programmers are concerned particularly that source listings be easily understood. Users are concerned that the human interface of the program is easily understood.

CLAIM: One researcher claims that software clarity is a

"Measure of how clear a program is, i.e., how easy it is to read, understand, and use" (Kosarajo and Ledgard, as quoted in McCall, Richards, and Walters 1977).

Although users are concerned about the clarity of the program dialogue, it is best to let the users' concern for clarity be handled by the quality factor, usability. Then clarity only addresses the ease of reading and understanding the program.

DOMAIN: This factor applies to any source code. Programming is such an iterative process that the code should be programmed with clarity at least to aid the programmer who writes some code, reviews it, and then writes some more.

NECESSARY CONDITIONS: The audience to which the code must be clear must be defined for the context. The criteria for determining that the program is clear must be determined for the context.

QUALIFICATIONS: This is a programmer-oriented quality factor, not a user-oriented one.

CRITICAL ANALYSIS: Program clarity can suffer either because the necessary information is not provided, or because the information provided is not well presented. Either problem impacts program clarity, since this quality factor is concerned about the net impact on the ease of understanding the program, regardless of the cause. In this case, clarity can be equated with self-descriptiveness.

SQM RELATIONSHIPS: Halstead's Effort, RADC's SD.2(1), SD.2(3)

TOOLS: ***, METRIQS, San Juan Capistrano, CA
AdaMAT, Dynamics Research Corporation, Andover, MA
AMS, Rome Air Development Center, Griffiss AFB, NY
LOGISCOPE, Verilog USA Inc., Alexandria, VA
PC-Metric, SET Laboratories, Inc., Mulino, OR
(*** This tool is used internally to support their consulting service.)

REFERENCES: Albrecht and Gaffney 1983; Arthur 1983; Basili, Selby, and Phillips 1983; Boehm et al. 1978; Bowen, Wigle, and Tsai 1985; Bulut 1974; Carver 1986; Coulter 1983; Curtis 1980; Elshoff 1976; Fitzsimmons 1978; Funami 1976; Gordon

1976, 1979; Halstead 1972, 1973, 1977, 1979; Halstead and Zislis 1973; Henry, Kafura, and Harris 1981; Jones 1978; Kafura and Reddy 1987; Lasky, Kaminsky, and Boaz 1989; Lassez 1981; Li and Cheung 1987; McCall, Richards, and Walters 1977; Millman and Curtis 1980; Murine 1983, 1985a, 1985b, 1986, 1988; Pierce, Hartley, and Worrells 1987; Prather 1984; Shen, Conte, and Dunsmore 1983; Shneiderman 1980; Siyan 1989; Sunazuka, Azuma, and Yamagishi 1985; Warthman 1987; Weyuker 1988; Zislis 1973; Zweben 1977

SOFTWARE QUALITY FACTOR DATA SHEET

Completeness

This quality factor addresses the concern that program functions be implemented completely.

CLAIM: Some researchers claim that software completeness is determined by

"Those characteristics of software which provide full implementation of the functions required" (Bowen, Wigle, and Tsai 1985).

DOMAIN: This factor only applies to source code for which the required functions have been explicitly stated.

NECESSARY CONDITIONS: The required functions and their extent must be defined for the context.

QUALIFICATIONS: None.

CRITICAL ANALYSIS: This definition of completeness distinguishes functional completeness from the completeness of the source code. The latter encompasses additional attributes, such as program comments, that are desired by programmers rather than the user. Furthermore, these attributes are already addressed by the understandability and clarity quality factors.

SQM RELATIONSHIPS: RADC's CP.1(2)

TOOLS: ***, METRIQS, San Juan Capistrano, CA
AdaMAT, Dynamics Research Corporation, Andover, MA
AMS, Rome Air Development Center, Griffiss AFB, NY
(*** This tool is used internally to support their consulting service.)

REFERENCES: Boehm et al. 1978; Bowen, Wigle, and Tsai 1985; Lasky, Kaminsky, and Boaz 1989; McCall, Richards, and Walters 1977; Millman and Curtis 1980; Murine 1983, 1985a, 1985b, 1986, 1988; Pierce, Hartley, and Worrells 1987; Shneiderman 1980; Sunazuka, Azuma, and Yamagishi 1985; Wartham 1987

SOFTWARE QUALITY FACTOR DATA SHEET

Complexity

This quality factor addresses the concern that programs not be complex. There are many types of program complexity, but the complexity of concern is the complexity that is constructed in the mind of a programmer, the psychological complexity.

CLAIM: The authors claim that

"The complexity of a program is the extent to which it is involved or intricate, composed of many interwoven parts.".

DOMAIN: This factor applies to any source code.

NECESSARY CONDITIONS: The criteria for determining complexity must be defined for the context.

QUALIFICATIONS: Although the psychological complexity is the important one, a given SQM may claim that psychological complexity is totally encompassed by the complexity of a particular structure of a program. In this case, the two complexities can be used interchangeably. It is important that the type of complexity measure be qualified. Different types must not be directly compared.

This is a programmer-oriented quality factor, not a user-oriented one.

CRITICAL ANALYSIS: This quality factor addresses the complexity of program code, not the complexity of the user interface. Programmers are concerned that programs not be complex so that they can easily and assuredly satisfy programmer and user needs. Any complexity that may exist for the user is addressed by the quality factor, usability.

SQM RELATIONSHIPS: Albrecht's Function Points; Ejiogu's Structural Complexity; Halstead's Length, Volume, and Effort; Henry's Information Flow; McCabe's Cyclomatic Complexity and Essential Complexity; RADC's SI.3(1)

TOOLS: ***, METRIQS, San Juan Capistrano, CA
AdaMAT, Dynamics Research Corporation, Andover, MA
AMS, Rome Air Development Center, Griffiss AFB, NY
Analysis of Complexity Tool (ACT), McCabe & Assocs., Columbia, MD
Battlemap Analysis Tool (BAT), McCabe & Assocs., Columbia, MD
Checkpoint, Software Productivity Research, Inc., Burlington, MA
COMPLEXIMETER, Softmetrix, Inc., Chicago, IL
LOGISCOPE, Verilog USA Inc., Alexandria, VA
PC-Metric, SET Laboratories, Inc., Mulino, OR

SIZE PLANNER, Quantitative Software Management, Inc., McClean, VA
(*** This tool is used internally to support their consulting service.)

REFERENCES: Albrecht 1979, 1985; Albrecht and Gaffney 1983; Arthur 1983; Basili 1983; Basili and Hutchens 1983; Basili, Selby, and Phillips 1983; Behrens 1983; Boehm et al. 1978; Bowen, Wigle, and Tsai 1985; Bulut 1974; Carver 1986; Ct et al. 1988; Coulter 1983; Crawford, McIntosh, and Pregibon 1985; Curtis 1980; Drummond 1985; Ejiogu 1984a, 1984b, 1987, 1988, 1990; Elshoff 1976, 1982, 1983; Evangelist 1983; Fitzsimmons 1978; Funami 1976; Gaffney 1981; Gordon 1976, 1979; Halstead 1972, 1973, 1977, 1979; Halstead and Zislis 1973; Harrison 1984; Henry 1979; Henry and Kafura 1984; Henry, Kafura, and Harris 1981; Jones 1978, 1988; Kafura and Henry 1982; Kafura and Reddy 1987; Lasky, Kaminsky, and Boaz 1989; Lassez 1981; Li and Cheung 1987; Lind and Vairavan 1989; Low and Jeffrey 1990; Mannino, Stoddard, and Sudduth 1990; McCabe 1976, 1982, 1989; McCabe and Butler 1989; McCall, Richards, and Walters 1977; McClure 1978; Millman and Curtis 1980; Murine 1983, 1985a, 1985b, 1986, 1988; Myers 1977; Pierce, Hartley, and Worrells 1987; Prather 1983, 1984; Ramamurthy and Melton 1988; Schneidewind and Hoffmann 1979; Shen, Conte, and Dunsmore 1983; Shneiderman 1980; Siyan 1989; Sunazuka, Azuma, and Yamagishi 1985; Walsh 1979; Ward 1989; Warthman 1987; Weyuker 1988; Zislis 1973; Zweben 1977

SOFTWARE QUALITY FACTOR DATA SHEET

Conciseness

This quality factor addresses the concern that programs not contain any extraneous information.

CLAIM: Some researchers claim that software conciseness is determined by

"The ability to satisfy functional requirements with minimum amount of software" (McCall, Richards, and Walters 1977).

DOMAIN: This factor only applies to source code for which the functional requirements have been explicitly stated.

NECESSARY CONDITIONS: The functional requirements must be defined for the context.

QUALIFICATIONS: This is a programmer-oriented quality factor, not a user-oriented one.

CRITICAL ANALYSIS: This quality factor basically addresses how efficiently the source code was used. Any efficiency quality factors may already encompass this quality factor. Another word used for efficiency is effectiveness.

SQM RELATIONSHIPS: Halstead's Potential Volume, McCabe's Essential Complexity,

TOOLS: Analysis of Complexity Tool (ACT), McCabe & Assocs., Columbia, MD
Battlemap Analysis Tool (BAT), McCabe & Assocs., Columbia, MD
LOGISCOPE, Verilog USA Inc., Alexandria, VA

REFERENCES: Halstead 1977; McCabe 1976, 1982

SOFTWARE QUALITY FACTOR DATA SHEET

Consistency

This quality factor addresses the concern that the source code syntax and constructs in programs be implemented uniformly.

CLAIM: Some researchers claim that software consistency is determined by

> "Those characteristics of software which provide for uniform design and implementation techniques and notation" (Bowen, Wigle, and Tsai 1985).

DOMAIN: This factor applies to any source code.

NECESSARY CONDITIONS: The standard notation must be defined for the context.

QUALIFICATIONS: This is a programmer-oriented quality factor, not a user-oriented one.

CRITICAL ANALYSIS: This quality factor addresses self-consistency. External consistency is more properly addressed by the quality factors, completeness, correctness, and performance, since they all compare the implementation with an external standard.

SQM RELATIONSHIPS: RADC's software metrics

TOOLS: ***, METRIQS, San Juan Capistrano, CA
AdaMAT, Dynamics Research Corporation, Andover, MA
AMS, Rome Air Development Center, Griffiss AFB, NY
(*** This tool is used internally to support their consulting service.)

REFERENCES: Boehm et al. 1978; Bowen, Wigle, and Tsai 1985; Lasky, Kaminsky, and Boaz 1989; McCall, Richards, and Walters 1977; Millman and Curtis 1980; Murine 1983, 1985a, 1985b, 1986, 1988; Pierce, Hartley, and Worrells 1987; Shneiderman 1980; Sunazuka, Azuma, and Yamagishi 1985; Wartham 1987

SOFTWARE QUALITY FACTOR DATA SHEET

Correctness

This quality factor addresses the concern that software design and documentation formats conform to the specifications and standards set for them. It is not concerned with any content affecting software operation or performance.

CLAIM: Some researchers claim that software correctness is determined by the

"Extent to which the software conforms to its specifications and standards" (Bowen, Wigle, and Tsai 1985).

DOMAIN: This factor only applies to source code for which specifications and standards have been explicitly stated.

NECESSARY CONDITIONS: The specifications and standards must be defined for the context.

QUALIFICATIONS: This is a programmer-oriented quality factor, not a user-oriented one.

CRITICAL ANALYSIS: This quality factor must not be confused with completeness. Although each specification may have been completely addressed, it may have been addressed incorrectly. Completeness of a function must be satisfied before its correctness can be established.

SQM RELATIONSHIPS: RADC's CP.1(2)

TOOLS: ***, METRIQS, San Juan Capistrano, CA
AdaMAT, Dynamics Research Corporation, Andover, MA
AMS, Rome Air Development Center, Griffiss AFB, NY
(*** This tool is used internally to support their consulting service.)

REFERENCES: Boehm et al. 1978; Bowen, Wigle, and Tsai 1985; Lasky, Kaminsky, and Boaz 1989; McCall, Richards, and Walters 1977; Millman and Curtis 1980; Murine 1983, 1985a, 1985b, 1986, 1988; Pierce, Hartley, and Worrells 1987; Shneiderman 1980; Sunazuka, Azuma, and Yamagishi 1985; Wartham 1987

SOFTWARE QUALITY FACTOR DATA SHEET

Efficiency

This quality factor addresses the concern that programs optimally use any computer resources.

CLAIM: Some researchers claim that

"A software product possesses the characteristic Efficiency to the extent that it fulfills its purpose without waste of resources." (Boehm et al. 1978)

In practice, one is not usually concerned with general efficiency, but with the efficiency of utilization of a particular resource, for example, efficient use of processing time.

DOMAIN: This factor only applies to source code for which the purpose has been explicitly stated.

NECESSARY CONDITIONS: The criteria for establishing that a resource is wasted must be defined for the context.

QUALIFICATIONS: None.

CRITICAL ANALYSIS: This quality factor must not be confused with how the software rates for a particular performance measure. Performance is concerned with how close the measure comes to a standard. Efficiency requires that the standard be met, and then addresses how effectively it was met.

SQM RELATIONSHIPS: Halstead's Potential Volume, McCabe's Essential Complexity, RADC's EP.2(4)

TOOLS: ***, METRIQS, San Juan Capistrano, CA
AdaMAT, Dynamics Research Corporation, Andover, MA
AMS, Rome Air Development Center, Griffiss AFB, NY
Analysis of Complexity Tool (ACT), McCabe & Assocs., Columbia, MD
Battlemap Analysis Tool (BAT), McCabe & Assocs., Columbia, MD
(*** This tool is used internally to support their consulting service.)

REFERENCES: Boehm et al. 1978; Bowen, Wigle, and Tsai 1985; Halstead 1977; Lasky, Kaminsky, and Boaz 1989; McCabe 1976, 1982; McCall, Richards, and Walters 1977; Millman and Curtis 1980; Murine 1983, 1985a, 1985b, 1986, 1988; Pierce, Hartley, and Worrells 1987; Shneiderman 1980; Sunazuka, Azuma, and Yamagishi 1985; Wartham 1987

SOFTWARE QUALITY FACTOR DATA SHEET

Expandability

This quality factor addresses the concern that program limitations be easy to extend.

CLAIM: Some researchers claim that software expandability is determined by the

> "Relative effort [required] to increase the software capability or performance by enhancing current functions or by adding new functions or data" (Bowen, Wigle, and Tsai 1985).

DOMAIN: This factor applies to any source code.

NECESSARY CONDITIONS: The criteria for distinguishing a change from an expansion must be defined for the context.

QUALIFICATIONS: None.

CRITICAL ANALYSIS: This quality factor must not be confused with flexibility, maintainability, or reusability. Expandability assumes that all existing functions are still performed, but possibly added to or enhanced. Expandability also assumes that there is no change in the context in which the program is used. Another word used to describe this quality factor is augmentability.

SQM RELATIONSHIPS: RADC's GE.1(1), GE.2(2), MO.1(3), MO.1(5), MO.1(7), SD.2(1), SD.2(3), SI.3(1)

TOOLS: ***, METRIQS, San Juan Capistrano, CA
AdaMAT, Dynamics Research Corporation, Andover, MA
AMS, Rome Air Development Center, Griffiss AFB, NY
(*** This tool is used internally to support their consulting service.)

REFERENCES: Boehm et al. 1978; Bowen, Wigle, and Tsai 1985; Lasky, Kaminsky, and Boaz 1989; McCall, Richards, and Walters 1977; Millman and Curtis 1980; Murine 1983, 1985a, 1985b, 1986, 1988; Pierce, Hartley, and Worrells 1987; Shneiderman 1980; Sunazuka, Azuma, and Yamagishi 1985; Wartham 1987

SOFTWARE QUALITY FACTOR DATA SHEET

Flexibility

This quality factor addresses the concern that programs be easy to change to meet different requirements, with no change in the context.

CLAIM: Some researchers claim that software flexibility is determined by the

> "Ease of effort for changing the software missions, functions, or data to satisfy other requirements" (Bowen, Wigle, and Tsai 1985).

DOMAIN: This factor applies to any source code.

NECESSARY CONDITIONS: The criteria for distinguishing different requirements from a change in context or an expansion must be defined for the context.

QUALIFICATIONS: None.

CRITICAL ANALYSIS: This quality factor must not be confused with expandability, maintainability, or reusability. Expandability addresses the simple case of satisfying extended requirements, rather than different requirements. In order to keep flexibility independent of expandability, flexibility should assess the ability to accommodate different requirements. Maintainability assumes no change in the functions of a program. Reusability requires a change of context.

SQM RELATIONSHIPS: RADC's GE.1(1), GE.2(2), MO.1(3), MO.1(5), MO.1(7), SD.2(1), SD.2(3), SI.3(1)

TOOLS: ***, METRIQS, San Juan Capistrano, CA
AdaMAT, Dynamics Research Corporation, Andover, MA
AMS, Rome Air Development Center, Griffiss AFB, NY
(*** This tool is used internally to support their consulting service.)

REFERENCES: Boehm et al. 1978; Bowen, Wigle, and Tsai 1985; Lasky, Kaminsky, and Boaz 1989; McCall, Richards, and Walters 1977; Millman and Curtis 1980; Murine 1983, 1985a, 1985b, 1986, 1988; Pierce, Hartley, and Worrells 1987; Shneiderman 1980; Sunazuka, Azuma, and Yamagishi 1985; Wartham 1987

SOFTWARE QUALITY FACTOR DATA SHEET

Integrity

This quality factor addresses the concern that programs must continue to perform their function even under adverse conditions: inputs that are unexpected, improper, or harmful.

CLAIM: Some researchers claim that software integrity is determined by the

"Ability of software to prevent purposeful or accidental damage to the data or software" (United States Army Computer Systems Command as quoted in McCall, Richards, and Walters 1977).

This definition assumes that damage to the data or software will result in the loss of function.

DOMAIN: This factor applies to any source code.

NECESSARY CONDITIONS: The adverse conditions must be defined for the context.

QUALIFICATIONS: This definition assumes that program integrity is an issue only when the program is operating. When it is not operating, the operating system must guard the program and its data from damage; the program cannot.

CRITICAL ANALYSIS: This quality factor must not be confused with security or reliability. Security contributes directly to integrity, but security is only concerned with unauthorized access to information. When a program is not secure, the confidentiality of the data is compromised, whether or not the program is subjected to adverse conditions. Similarly, reliability is determined by program failures that occur during execution under the specified conditions, not during adverse conditions.

SQM RELATIONSHIPS: RADC's software metrics

TOOLS: ***, METRIQS, San Juan Capistrano, CA
AdaMAT, Dynamics Research Corporation, Andover, MA
AMS, Rome Air Development Center, Griffiss AFB, NY
(*** This tool is used internally to support their consulting service.)

REFERENCES: Boehm et al. 1978; Bowen, Wigle, and Tsai 1985; Lasky, Kaminsky, and Boaz 1989; McCall, Richards, and Walters 1977; Millman and Curtis 1980; Murine 1983, 1985a, 1985b, 1986, 1988; Pierce, Hartley, and Worrells 1987; Shneiderman 1980; Sunazuka, Azuma, and Yamagishi 1985; Wartham 1987

SOFTWARE QUALITY FACTOR DATA SHEET

Maintainability

This quality factor addresses the concern that programs be easy to fix, once a failure is identified.

CLAIM: Some researchers claim that software maintainability is determined by the

"Ease of effort for locating and fixing a software failure within a specified time period" (Bowen, Wigle, and Tsai 1985).

DOMAIN: This factor only applies to source code for which operational failure and restoration is defined.

NECESSARY CONDITIONS: The criteria for program failures must be defined for the context.

QUALIFICATIONS: In general, maintainability encompasses the ease of identifying and making any changes to programs. But when used as one of a set of quality factors, maintainability is usually restricted to changes that are made to fix software failures as in the definition above. Often, maintainability is further restricted to the ease of making the fix, once the cause for the failure is found. This restriction keeps it highly independent. In particular, it makes maintainability independent of complexity and understandability.

CRITICAL ANALYSIS: This quality factor must not be confused with expandability, flexibility, modifiability, portability, or reusability. None of these quality factors apply to changes made in response to software failures. They address the concerns related to making changes for other reasons.

SQM RELATIONSHIPS: Halstead's Effort; Henry's Information Flow; McCabe's Cyclomatic Complexity; RADC's MO.1(3), MO.1(5), MO.1(7), SD.2(1), SD.2(3), SI.3(1)

TOOLS: ***, METRIQS, San Juan Capistrano, CA
AdaMAT, Dynamics Research Corporation, Andover, MA
AMS, Rome Air Development Center, Griffiss AFB, NY
Analysis of Complexity Tool (ACT), McCabe & Assocs., Columbia, MD
Battlemap Analysis Tool (BAT), McCabe & Assocs., Columbia, MD
PC-Metric, SET Laboratories, Inc., Mulino, OR
(*** This tool is used internally to support their consulting service.)

REFERENCES: Albrecht and Gaffney 1983; Arthur 1983; Basili 1983; Basili and Hutchens 1983; Basili, Selby, and Phillips 1983; Boehm et al. 1978; Bowen, Wigle, and Tsai 1985; Bulut 1974; Carver 1986; Ct et al. 1988; Coulter 1983; Crawford, McIntosh, and Pregibon 1985; Curtis 1980; Elshoff 1976, 1982, 1983; Evangelist 1983; Fitzsimmons 1978; Funami 1976; Gaffney 1981; Gordon 1976, 1979; Halstead

1972, 1973, 1977, 1979; Halstead and Zislis 1973; Harrison 1984; Henry 1979; Henry and Kafura 1984; Henry, Kafura, and Harris 1981; Jones 1978; Kafura and Henry 1982; Kafura and Reddy 1987; Lasky, Kaminsky, and Boaz 1989; Lassez 1981; Li and Cheung 1987; Lind and Vairavan 1989; Mannino, Stoddard, and Sudduth 1990; McCabe 1976, 1982, 1989; McCabe and Butler 1989; McCall, Richards, and Walters 1977; McClure 1978; Millman and Curtis 1980; Murine 1983, 1985a, 1985b, 1986, 1988; Myers 1977; Pierce, Hartley, and Worrells 1987; Prather 1983, 1984; Ramamurthy and Melton 1988; Schneidewind and Hoffmann 1979; Shen, Conte, and Dunsmore 1983; Shneiderman 1980; Siyan 1989; Sunazuka, Azuma, and Yamagishi 1985; Walsh 1979; Ward 1989; Warthman 1987; Weyuker 1988; Zislis 1973; Zweben 1977

SOFTWARE QUALITY FACTOR DATA SHEET

Modifiability

This quality factor addresses the concern that programs be easy to change, regardless of the reason for the change.

CLAIM: Some researchers claim that

"A software product possesses modifiability to the extent that it facilitates the incorporation of changes, once the nature of the desired change has been determined." (Boehm et al. 1978)

DOMAIN: This factor applies to any source code.

NECESSARY CONDITIONS: The criteria for a program change must be defined for the context.

QUALIFICATIONS: The effort expended to understand how to change a program is not addressed by this quality factor. Only the next step of implementing the change is addressed.

CRITICAL ANALYSIS: This quality factor encompasses all the concerns addressed by expandability, flexibility, maintainability, portability, and reusability. This does not mean that they are identical, but rather that modifiability is fundamental to them all. The ease of making all types of changes must be addressed for this quality factor. The other quality factors emphasize a particular type of modifiability.

SQM RELATIONSHIPS: Halstead's Effort; Henry's Information Flow; McCabe's Cyclomatic Complexity

TOOLS: Analysis of Complexity Tool (ACT), McCabe & Assocs., Columbia, MD
Battlemap Analysis Tool (BAT), McCabe & Assocs., Columbia, MD
LOGISCOPE, Verilog USA Inc., Alexandria, VA
PC-Metric, SET Laboratories, Inc., Mulino, OR

REFERENCES: Albrecht and Gaffney 1983; Arthur 1983; Basili 1983; Basili and Hutchens 1983; Basili, Selby, and Phillips 1983; Bulut 1974; Carver 1986; Ct et al. 1988; Coulter 1983; Crawford, McIntosh, and Pregibon 1985; Curtis 1980; Elshoff 1976, 1982, 1983; Evangelist 1983; Fitzsimmons 1978; Funami 1976; Gaffney 1981; Gordon 1976, 1979; Halstead 1972, 1973, 1977, 1979; Halstead and Zislis 1973; Harrison 1984; Henry 1979; Henry and Kafura 1984; Henry, Kafura, and Harris 1981; Jones 1978; Kafura and Henry 1982; Kafura and Reddy 1987; Lassez 1981; Li and Cheung 1987; Lind and Vairavan 1989; Mannino, Stoddard, and Sudduth 1990; McCabe 1976, 1982, 1989; McCabe and Butler 1989; McClure 1978; Myers 1977; Prather 1983, 1984; Ramamurthy and Melton 1988; Schneidewind and

Hoffmann 1979; Shen, Conte, and Dunsmore 1983; Shneiderman 1980; Siyan 1989; Walsh 1979; Ward 1989; Weyuker 1988; Zislis 1973; Zweben 1977

SOFTWARE QUALITY FACTOR DATA SHEET

Modularity

This quality factor addresses the concern that programs be composed of many small, simple, independent steps that are clearly delineated by the code.

CLAIM: One researcher claims that software modularity is the

> "Formal way of dividing a program into a number of sub-units each having a well defined function and relationship to the rest of the program" (Mealy as quoted in McCall, Richards, and Walters 1977).

These sub-units are called modules. Thus, the quality factor is modularity. The program itself may be a sub-unit of some larger program.

DOMAIN: This factor applies to any source code.

NECESSARY CONDITIONS: The criteria for distinguishing modules and intermodule relationships must be defined for the context.

QUALIFICATIONS: This is a programmer-oriented quality factor, not a user-oriented one.

CRITICAL ANALYSIS: This quality factor is usually used to indicate the presence of one of the following quality factors: expandability, flexibility, maintainability, modifiability, portability, reusability, and understandability. Depending on which of these quality factors is to be indicated by modularity, different modules and intermodule relationships are identified. For example, a program that is broken into one set of modules to enhance expandability may be broken into a different set of modules to enhance understandability.

SQM RELATIONSHIPS: McCabe's Cyclomatic Complexity and Essential Complexity; RADC's MO.1(3), MO.1(5), MO.1(7)

TOOLS:
***, METRIQS, San Juan Capistrano, CA
AdaMAT, Dynamics Research Corporation, Andover, MA
AMS, Rome Air Development Center, Griffiss AFB, NY
Analysis of Complexity Tool (ACT), McCabe & Assocs., Columbia, MD
Battlemap Analysis Tool (BAT), McCabe & Assocs., Columbia, MD
LOGISCOPE, Verilog USA Inc., Alexandria, VA
PC-Metric, SET Laboratories, Inc., Mulino, OR
(*** This tool is used internally to support their consulting service.)

REFERENCES: Arthur 1983; Basili 1983; Basili and Hutchens 1983; Basili, Selby, and Phillips 1983; Boehm et al. 1978; Bowen, Wigle, and Tsai 1985; Carver 1986; Ct et al. 1988; Crawford, McIntosh, and Pregibon 1985; Curtis 1980; Elshoff 1982, 1983; Evangelist 1983; Gaffney 1981; Harrison 1984; Henry, Kafura, and Harris 1981; Kafura and Reddy 1987; Lasky, Kaminsky, and Boaz 1989; Li and Cheung 1987; Lind and Vairavan 1989; Mannino, Stoddard, and Sudduth 1990; McCabe 1976, 1982, 1989; McCabe and Butler 1989; McCall, Richards, and Walters 1977; McClure 1978; Millman and Curtis 1980; Murine 1983, 1985a, 1985b, 1986, 1988; Myers 1977; Pierce, Hartley, and Worrells 1987; Prather 1983, 1984; Ramamurthy and Melton 1988; Schneidewind and Hoffmann 1979; Shneiderman 1980; Siyan 1989; Sunazuka, Azuma, and Yamagishi 1985; Walsh 1979; Ward 1989; Warthman 1987; Weyuker 1988

SOFTWARE QUALITY FACTOR DATA SHEET

Performance

This quality factor addresses the concern of how well a program attribute or function is implemented with respect to some standard. Often, this is related to the utilization of resources.

CLAIM: One researcher claims that software performance is determined by

> "The effectiveness with which resources of the host system are utilized toward meeting the objective of the software system" (Dennis as quoted in McCall, Richards, and Walters 1977).

DOMAIN: This factor only applies to source code for which performance standards have been set.

NECESSARY CONDITIONS: The standard against which the implementation is to be compared must be defined for the context.

QUALIFICATIONS: None.

CRITICAL ANALYSIS: This quality factor must not be confused with completeness or correctness. Performance is concerned with how well the job is done, given that a program completely and correctly meets its specifications. The performance quality factor is used to determine which satisfactory program performs better.

SQM RELATIONSHIPS: Albrecht's Function Points; Ejiogu's Structural Complexity; Halstead's Length, Volume, and Effort; Henry's Information Flow; McCabe's Cyclomatic Complexity and Essential Complexity

TOOLS: Analysis of Complexity Tool (ACT), McCabe & Assocs., Columbia, MD
Battlemap Analysis Tool (BAT), McCabe & Assocs., Columbia, MD
Checkpoint, Software Productivity Research, Inc., Burlington, MA
COMPLEXIMETER, Softmetrix, Inc., Chicago, IL
LOGISCOPE, Verilog USA Inc., Alexandria, VA
PC-Metric, SET Laboratories, Inc., Mulino, OR
SIZE PLANNER, Quantitative Software Management, Inc., McClean, VA

REFERENCES: Albrecht 1979, 1985; Albrecht and Gaffney 1983; Arthur 1983; Basili 1983; Basili and Hutchens 1983; Basili, Selby, and Phillips 1983; Behrens 1983; Bulut 1974; Carver 1986; Ct et al. 1988; Coulter 1983; Crawford, McIntosh, and Pregibon 1985; Curtis 1980; Drummond 1985; Ejiogu 1984a, 1984b, 1987, 1988, 1990; Elshoff 1976, 1982, 1983; Evangelist 1983; Fitzsimmons 1978; Funami 1976; Gaffney 1981; Gordon 1976, 1979; Halstead 1972, 1973, 1977, 1979; Halstead and Zislis 1973; Harrison 1984; Henry 1979; Henry and Kafura 1984;

Henry, Kafura, and Harris 1981; Jones 1978, 1988; Kafura and Henry 1982; Kafura and Reddy 1987; Lassez 1981; Li and Cheung 1987; Lind and Vairavan 1989; Low and Jeffrey 1990; Mannino, Stoddard, and Sudduth 1990; McCabe 1976, 1982, 1989; McCabe and Butler 1989; McClure 1978; Myers 1977; Prather 1983, 1984; Ramamurthy and Melton 1988; Schneidewind and Hoffmann 1979; Shen, Conte, and Dunsmore 1983; Shneiderman 1980; Siyan 1989; Walsh 1979; Ward 1989; Weyuker 1988; Zislis 1973; Zweben 1977

SOFTWARE QUALITY FACTOR DATA SHEET

Portability

This quality factor addresses the concern that programs be changed easily to operate on a different set of equipment.

CLAIM: One researcher claims that software portability is determined by

> "How quickly and cheaply the software system can be converted to perform the same functions using different equipment" (Kosy as quoted in McCall, Richards, and Walters 1977).

DOMAIN: This factor only applies to source code that performs functions that are supported on other equipment.

NECESSARY CONDITIONS: The criteria for distinguishing lack of modifiability from lack of portability must be defined for the context.

QUALIFICATIONS: The portability of the code, not the function that it performs, is addressed by this quality factor. If the hardware on which it operates is unique, there is no machine to which the code can be transported to perform the same function. This is not the fault of the code.

Every change required to operate a program on different equipment is weighted by the modifiability of the code. If the modifiability is poor, a very machine-independent program may still appear to be not very portable.

CRITICAL ANALYSIS: This quality factor must not be confused with reusability. Portability addresses only those changes required to make the program work on a different machine; everything else remains the same.

SQM RELATIONSHIPS: RADC's MO.1(3), MO.1(5), MO.1(7), SD.2(1), SD.2(3)

TOOLS: ***, METRIQS, San Juan Capistrano, CA
AdaMAT, Dynamics Research Corporation, Andover, MA
AMS, Rome Air Development Center, Griffiss AFB, NY
(*** This tool is used internally to support their consulting service.)

REFERENCES: Boehm et al. 1978; Bowen, Wigle, and Tsai 1985; Lasky, Kaminsky, and Boaz 1989; McCall, Richards, and Walters 1977; Millman and Curtis 1980; Murine 1983, 1985a, 1985b, 1986, 1988; Pierce, Hartley, and Worrells 1987; Shneiderman 1980; Sunazuka, Azuma, and Yamagishi 1985; Wartham 1987

SOFTWARE QUALITY FACTOR DATA SHEET

Reliability

This quality factor addresses the concern that programs continue to perform properly over time.

CLAIM: One researcher claims that software reliability is determined by

"The probability that a software system will operate without failure for at least a given period of time when used under stated conditions" (Kosy as quoted in McCall, Richards, and Walters 1977).

DOMAIN: This factor only applies to source code for which the proper operating conditions are specified.

NECESSARY CONDITIONS: The proper operating conditions and the criteria for establishing a failure must be defined for the context.

QUALIFICATIONS: For code-based SQM, this quality factor must be given by probabilities based solely on program code. Most reliability estimates are based on experimental data.

CRITICAL ANALYSIS: This quality factor must not be confused with accuracy or correctness. Reliability addresses the continuance of functions already presumed to be both accurate and correct.

SQM RELATIONSHIPS: Halstead's Effort; Henry's Information Flow; McCabe's Cyclomatic Complexity and Essential Complexity, RADC's SI.3(1)

TOOLS:
***, METRIQS, San Juan Capistrano, CA
AdaMAT, Dynamics Research Corporation, Andover, MA
AMS, Rome Air Development Center, Griffiss AFB, NY
Analysis of Complexity Tool (ACT), McCabe & Assocs., Columbia, MD
Battlemap Analysis Tool (BAT), McCabe & Assocs., Columbia, MD
LOGISCOPE, Verilog USA Inc., Alexandria, VA
PC-Metric, SET Laboratories, Inc., Mulino, OR
(*** This tool is used internally to support their consulting service.)

REFERENCES: Albrecht and Gaffney 1983; Arthur 1983; Basili 1983; Basili and Hutchens 1983; Basili, Selby, and Phillips 1983; Boehm et al. 1978; Bowen, Wigle, and Tsai 1985; Bulut 1974; Carver 1986; Ct et al. 1988; Coulter 1983; Crawford, McIntosh, and Pregibon 1985; Curtis 1980; Elshoff 1976, 1982, 1983; Evangelist 1983; Fitzsimmons 1978; Funami 1976; Gaffney 1981; Gordon 1976, 1979; Halstead 1972, 1973, 1977, 1979; Halstead and Zislis 1973; Harrison 1984; Henry 1979; Henry and Kafura 1984; Henry, Kafura, and Harris 1981; Jones 1978; Kafura and

Henry 1982; Kafura and Reddy 1987; Lasky, Kaminsky, and Boaz 1989; Lassez 1981; Li and Cheung 1987; Lind and Vairavan 1989; Mannino, Stoddard, and Sudduth 1990; McCabe 1976, 1982, 1989; McCabe and Butler 1989; McCall, Richards, and Walters 1977; McClure 1978; Millman and Curtis 1980; Murine 1983, 1985a, 1985b, 1986, 1988; Myers 1977; Pierce, Hartley, and Worrells 1987; Prather 1983, 1984; Ramamurthy and Melton 1988; Schneidewind and Hoffmann 1979; Shen, Conte, and Dunsmore 1983; Shneiderman 1980; Siyan 1989; Sunazuka, Azuma, and Yamagishi 1985; Walsh 1979; Ward 1989; Warthman 1987; Weyuker 1988; Zislis 1973; Zweben 1977

SOFTWARE QUALITY FACTOR DATA SHEET

Reusability

This quality factor addresses the concern that programs be easy to reuse in a different application.

CLAIM: Some researchers claim that software reusability is determined by the

"Relative effort to convert a software component for use in a different application" (Bowen, Wigle, and Tsai 1985).

DOMAIN: This factor applies to any source code.

NECESSARY CONDITIONS: The proper operating conditions and the criteria for establishing a failure must be defined for the context.

QUALIFICATIONS: Every change required to use a program in a different application is weighted by the modifiability of the code. If the modifiability is poor, a very application-independent program may still appear to be not very reusable.

This is a programmer-oriented quality factor, not a user-oriented one.

CRITICAL ANALYSIS: This quality factor must not be confused with expandability, flexibility, or portability. All three address ease of changing a program, but for reasons other than using the program to perform the same function in a different application.

SQM RELATIONSHIPS: RADC's AP.3(2), FS.1(2), GE.1(1), GE.2(2), MO.1(3), MO.1(5), MO.1(7), SD.2(1), SD.2(3), SI.3(1), ST.5(2)

TOOLS: ***, METRIQS, San Juan Capistrano, CA
AdaMAT, Dynamics Research Corporation, Andover, MA
AMS, Rome Air Development Center, Griffiss AFB, NY
(*** This tool is used internally to support their consulting service.)

REFERENCES: Boehm et al. 1978; Bowen, Wigle, and Tsai 1985; Lasky, Kaminsky, and Boaz 1989; McCall, Richards, and Walters 1977; Millman and Curtis 1980; Murine 1983, 1985a, 1985b, 1986, 1988; Pierce, Hartley, and Worrells 1987; Shneiderman 1980; Sunazuka, Azuma, and Yamagishi 1985; Wartham 1987

SOFTWARE QUALITY FACTOR DATA SHEET

Simplicity

This quality factor addresses the concern that, as much as possible, programs be implemented in strictly sequential steps that depend only on the step before it. Comments should exhibit a similar straight forwardness.

CLAIM: Some researchers claim that software simplicity is determined by

> "Those characteristics of software which provide for definition and implementation of functions in the most noncomplex and understandable manner" (Bowen, Wigle, and Tsai 1985).

Simplicity is the opposite of the quality factor, complexity.

DOMAIN: This factor applies to any source code.

NECESSARY CONDITIONS: The criteria for determining simplicity must be defined for the context.

QUALIFICATIONS: Simplicity of implementation must be evaluated independently of the simplicity or complexity of the algorithm. Simplicity addresses how simply an algorithm of a certain complexity is implemented.

This is a programmer-oriented quality factor, not a user-oriented one.

CRITICAL ANALYSIS: This quality factor is usually represented by other quality factors. Since simplicity is desired because it enhances understanding, clarity and understandability are more commonly used.

SQM RELATIONSHIPS: Albrecht's Function Points; Ejiogu's Structural Complexity; Halstead's Length, Volume, and Effort; Henry's Information Flow; McCabe's Cyclomatic Complexity and Essential Complexity; RADC's SI.3(1)

TOOLS: ***, METRIQS, San Juan Capistrano, CA
AdaMAT, Dynamics Research Corporation, Andover, MA
Analysis of Complexity Tool (ACT), McCabe & Assocs., Columbia, MD
Battlemap Analysis Tool (BAT), McCabe & Assocs., Columbia, MD
Checkpoint, Software Productivity Research, Inc., Burlington, MA
COMPLEXIMETER, Softmetrix, Inc., Chicago, IL
LOGISCOPE, Verilog USA Inc., Alexandria, VA
PC-Metric, SET Laboratories, Inc., Mulino, OR
SIZE PLANNER, Quantitative Software Management, Inc., McClean, VA
(*** This tool is used internally to support their consulting service.)

REFERENCES: Albrecht 1979, 1985; Albrecht and Gaffney 1983; Arthur 1983; Basili 1983; Basili and Hutchens 1983; Basili, Selby, and Phillips 1983; Behrens 1983; Boehm et al. 1978; Bowen, Wigle, and Tsai 1985; Bulut 1974; Carver 1986; Ct et al. 1988; Coulter 1983; Crawford, McIntosh, and Pregibon 1985; Curtis 1980; Drummond 1985; Ejiogu 1984a, 1984b, 1987, 1988, 1990; Elshoff 1976, 1982, 1983; Evangelist 1983; Fitzsimmons 1978; Funami 1976; Gaffney 1981; Gordon 1976, 1979; Halstead 1972, 1973, 1977, 1979; Halstead and Zislis 1973; Harrison 1984; Henry 1979; Henry and Kafura 1984; Henry, Kafura, and Harris 1981; Jones 1978, 1988; Kafura and Henry 1982; Kafura and Reddy 1987; Lasky, Kaminsky, and Boaz 1989; Lassez 1981; Li and Cheung 1987; Lind and Vairavan 1989; Low and Jeffrey 1990; Mannino, Stoddard, and Sudduth 1990; McCabe 1976, 1982, 1989; McCabe and Butler 1989; McCall, Richards, and Walters 1977; McClure 1978; Millman and Curtis 1980; Murine 1983, 1985a, 1985b, 1986, 1988; Myers 1977; Pierce, Hartley, and Worrells 1987; Prather 1983, 1984; Ramamurthy and Melton 1988; Schneidewind and Hoffmann 1979; Shen, Conte, and Dunsmore 1983; Shneiderman 1980; Siyan 1989; Sunazuka, Azuma, and Yamagishi 1985; Walsh 1979; Ward 1989; Warthman 1987; Weyuker 1988; Zislis 1973; Zweben 1977;

SOFTWARE QUALITY FACTOR DATA SHEET

Testability

This quality factor addresses the concern that programs be easy to test.

CLAIM: Some researchers claim that

> "A software product possesses the characteristic Testability to the extent that it facilitates the establishment of acceptance criteria and supports evaluation of its performance." (Boehm et al. 1978).

DOMAIN: This factor only applies to source code for which acceptance criteria can been established.

NECESSARY CONDITIONS: The test plan must be defined for the context.

QUALIFICATIONS: This is a programmer-oriented quality factor, not a user-oriented one.

CRITICAL ANALYSIS: This quality factor must not be confused with maintainability. Testability is only concerned with finding problems, not fixing them.

SQM RELATIONSHIPS: McCabe's Cyclomatic Complexity

TOOLS: Analysis of Complexity Tool (ACT), McCabe & Assocs., Columbia, MD
Battlemap Analysis Tool (BAT), McCabe & Assocs., Columbia, MD
LOGISCOPE, Verilog USA Inc., Alexandria, VA
PC-Metric, SET Laboratories, Inc., Mulino, OR

REFERENCES: Arthur 1983; Basili 1983; Basili and Hutchens 1983; Basili, Selby and Phillips 1983; Carver 1986; Ct et al. 1988; Crawford, McIntosh, and Pregibon 1985; Curtis 1980; Elshoff 1982, 1983; Evangelist 1983; Gaffney 1981; Harrison 1984; Henry, Kafura, and Harris 1981; Kafura and Reddy 1987; Li and Cheung 1987; Lind and Vairavan 1989; Mannino, Stoddard, and Sudduth 1990; McCabe 1976, 1982, 1989; McCabe and Butler 1989; McClure 1978; Myers 1977; Prather 1983, 1984; Ramamurthy and Melton 1988; Schneidewind and Hoffmann 1979; Shneiderman 1980; Siyan 1989; Walsh 1979; Ward 1989; Weyuker 1988

SOFTWARE QUALITY FACTOR DATA SHEET

Understandability

This quality factor addresses the concern that programs be easy to understand.

CLAIM: One researcher claims that software understandability is determined by the

> "Ease with which the implementation can be understood" (Richards as quoted in McCall, Richards, and Walters 1977).

DOMAIN: This factor applies to any source code.

NECESSARY CONDITIONS: The criteria for establishing whether a program is understood by a program reader must be defined for the context.

QUALIFICATIONS: This is a programmer-oriented quality factor, not a user-oriented one.

CRITICAL ANALYSIS: This quality factor must not be confused with modifiability. Before any change can be made to a program, the code must be understood as implemented. The next step may be to modify it.

SQM RELATIONSHIPS: Ejiogu's Structural Complexity; Halstead's Effort; Henry's Information Flow; McCabe's Cyclomatic Complexity and Essential Complexity

TOOLS: Analysis of Complexity Tool (ACT), McCabe & Assocs., Columbia, MD
Battlemap Analysis Tool (BAT), McCabe & Assocs., Columbia, MD
COMPLEXIMETER, Softmetrix, Inc., Chicago, IL
PC-Metric, SET Laboratories, Inc., Mulino, OR

REFERENCES: Albrecht and Gaffney 1983; Arthur 1983; Basili 1983; Basili and Hutchens 1983; Basili, Selby, and Phillips 1983; Bulut 1974; Carver 1986; Ct et al. 1988; Coulter 1983; Crawford, McIntosh, and Pregibon 1985; Curtis 1980; Ejiogu 1984a, 1984b, 1987, 1988, 1990; Elshoff 1976, 1982, 1983; Evangelist 1983; Fitzsimmons 1978; Funami 1976; Gaffney 1981; Gordon 1976, 1979; Halstead 1972, 1973, 1977, 1979; Halstead and Zislis 1973; Harrison 1984; Henry 1979; Henry and Kafura 1984; Henry, Kafura, and Harris 1981; Jones 1978; Kafura and Henry 1982; Kafura and Reddy 1987; Lassez 1981; Li and Cheung 1987; Lind and Vairavan 1989; Mannino, Stoddard, and Sudduth 1990; McCabe 1976, 1982, 1989; McCabe and Butler 1989; McClure 1978; Myers 1977; Prather 1983, 1984; Ramamurthy and Melton 1988; Schneidewind and Hoffmann 1979; Shen, Conte, and Dunsmore 1983; Shneiderman 1980; Siyan 1989; Walsh 1979; Ward 1989; Weyuker 1988; Zislis 1973; Zweben 1977

SOFTWARE QUALITY FACTOR DATA SHEET

Usability

This quality factor addresses the concern that programs be easy to use.

CLAIM: Some researchers claim that

"A software product possesses the characteristic Usability to the extent that it is convenient and practicable to use." (Boehm et al. 1978).

DOMAIN: This factor applies to any source code.

NECESSARY CONDITIONS: The criteria for establishing that a program is easy to use must be defined for the context.

QUALIFICATIONS: To keep this quality factor independent of maintainability, portability, and reusability, this quality factor must address the usability of an error-free program that is run on the intended machine, in the intended context or application.

CRITICAL ANALYSIS: This quality factor must not be confused with programmer concerns, like understandability. Usability is a strictly user-oriented quality factor.

SQM RELATIONSHIPS: RADC's software metrics

TOOLS: ***, METRIQS, San Juan Capistrano, CA
AdaMAT, Dynamics Research Corporation, Andover, MA
AMS, Rome Air Development Center, Griffiss AFB, NY
(*** This tool is used internally to support their consulting service.)

REFERENCES: Boehm et al. 1978; Bowen, Wigle, and Tsai 1985; Lasky, Kaminsky, and Boaz 1989; McCall, Richards, and Walters 1977; Millman and Curtis 1980; Murine 1983, 1985a, 1985b, 1986, 1988; Pierce, Hartley, and Worrells 1987; Shneiderman 1980; Sunazuka, Azuma, and Yamagishi 1985; Wartham 1987

BIBLIOGRAPHY

Akiyama, Fumio, "An Example of Software System Debugging", Proceedings IFIP Congress, 1971, pp. 353-358.

Albrecht, A. J., "Measuring Application Development Productivity", Proceedings of the IBM Application Development Symposium, 1979.

Albrecht, Allan J., "Function Points Help Managers Assess Applications", Computerworld, August 26, 1985.

Albrecht, Allan J. and John E. Gaffney, Jr., "Software Function, Source Lines of Code, and Development Effort Prediction: A Software Science Validation", IEEE Transactions on Software Engineering, Volume SE-9, No. 6, November 1983, pp. 639-648.

Amster, S. J. et al., "An Experiment in Automatic Quality Evaluation of Software", Proceedings of the Symposium on Computer Software Engineering, New York Polytechnic, New York, NY, 1976.

Arthur, J., "Metrics: Tools to Aid Software Code Quality", Computerworld, Volume 17, No. 26, June 27, 1983.

Atwood, M. E. et al., "Annotated Bibliography on Human Factors in Software Development", ARI Technical Report, P-79-1, Science Applications, Inc., Englewood, CO, June 1979.

Bailey, C. T. and W. L. Dingee, "A Software Study Using Halstead Metrics", ACM SIGMETRICS (1981 ACM Workshop/Symposium on Measurement and Evaluation of Software Quality), Volume 10, March 1981, pp. 189-197.

Baker, Albert L. et al., "A Philosophy for Software Measurement", The Journal of Systems and Software, Volume 12, No. 3, July 1990, pp. 277-281.

Baker, Albert L. and Stuart H. Zweben, "A Comparison of Measures of Control Flow Complexity", Proceedings of COMPSAC 1979, IEEE, 1979.

Baker, Albert L. and Stuart H. Zweben, "The Use of Software Science in Evaluating Modularity Concepts", IEEE Transactions on Software Engineering, Volume SE-5, No. 2, March 1979, pp. 110-120.

Basili, Victor R. and Tsai-Yun Phillips, "Evaluating and Comparing Software Metrics in the Software Engineering Laboratory", ACM SIGMETRICS (1981 ACM Workshop/Symposium on Measurement and Evaluation of Software Quality), Volume 10, March 1981, pp. 95-106.

Basili, Victor R., Richard W. Selby, Jr., and Tsai-Yun Phillips, "Metric Analysis and Data Validation Across FORTRAN Projects", IEEE Transactions on Software Engineering, Volume SE-9, No. 6, November 1983.

Basili, Victor R. and H. Dieter Rombach, "Implementing Quantitative SQA: A Practical Model", IEEE Software, Volume 4, No. 5, September 1987, pp. 6-9.

Basili, Victor R. and David H. Hutchens, "An Empirical Study of a Syntactic Complexity Family", IEEE Transactions on Software Engineering, Volume SE-9, No. 6, November 1983, pp. 664-672.

Basili, Victor R. and Robert W. Reiter, Jr., "Evaluating Automatable Measures of Software Development", Department of Computer Science, University of Maryland, College Park, MD, 1979.

Behrens, C., "Measuring the Productivity of Computer Systems Development Activities with Function Points", IEEE Transactions on Software Engineering, Volume SE-9, No. 6, November 1983, pp. 648-652.

Bell, D. E. and J. E. Sullivan, "Further Investigations into the Complexity of Software", MTR-2874, MITRE, Bedford, MA, 1974.

Boehm, B. W. et al., Proceedings of the TRW Symposium on Reliable, Cost-Effective, Secure Software, Los Angeles, CA, March 20-21, 1974.

Boehm, B. W. et al., Characteristics of Software Quality, North-Holland Publishing Company, New York, NY, 1978.

Bowen, Thomas P., Gary B. Wigle, and Jay T. Tsai, "Specification of Software Quality Attributes", RADC-TR-85-37, RADC, Griffiss Air Force Base, NY, Volumes I, II, and III, February 1985.

Brainerd, Walter S., Charles H. Goldberg, and Jonathan L. Gross, FORTRAN 77 Fundamentals and Style, Boyd & Fraser Publishing Co., Boston, MA, 1985.

Browne, Jim C., "A Proposal for Structural Models of Software Systems", Computer Science and Statistics: Proceedings of the 13th Symposium on the Interface, Springer-Verlag, Inc., New York, NY, 1981, pp. 208-210.

Buckley, Fletcher J., "Standard Set of Useful Software Metrics is Urgently Needed", Computer, Volume 22, No. 7, July 1989, pp. 88-89.

Bulut, N. and M. H. Halstead, "Impurities Found in Algorithm Implementations", SIGPLAN Notices, Volume 9, No. 3, March 1974, pp. 9-11.

Campbell, D. T. and J. C. Stanley, Experimental and Quasi-Experimental Designs for Research, Houghton Mifflin Company, Boston, MA, 1963.

Card, David N., "Major Obstacles Hinder Successful Measurement", IEEE Software, Volume 5, No. 6, November 1988, pp. 82 and 86.

Carver, D. L., "Criteria for Estimating Module Complexity", Journal of Systems Management, August 1986.

Chen, Edward T., "Program Complexity and Programmer Productivity", IEEE Transactions on Software Engineering, Volume SE-4, No. 3, May 1978, pp. 187-194.

Ct, V. et al., "Software Metrics: An Overview of Recent Results", The Journal of Systems and Software, Volume 8, No. 2, March 1988, pp. 121-131.

Coulter, Neal S., "Software Science and Cognitive Psychology", IEEE Transactions on Software Engineering, Volume SE-9, No. 2, March 1983, pp. 166-171.

Coupal, Daniel and Pierre Robillard, "Factor Analysis of Source Code Metrics", The Journal of Systems and Software, Volume 12, No. 3, July 1990, pp. 263-270.

Crawford, S. G., A. A. McIntosh, and D. Pregibon, "An Analysis of Static Metrics and Faults in C Software", The Journal of Systems and Software, Volume 5, No. 1, February 1985, pp. 37-48.

Currans, Nancy, "A Comparison of Counting Methods for Software Science and Cyclomatic Complexity", Pacific Northwest Software Quality Conference.

Curtis, B., "Measurement and Experimentation in Software Engineering", Proceedings of the IEEE, IEEE, New York, NY, Volume 68, No. 9, September 1980, pp. 1144-1157.

Curtis, B. et al., "Measuring the Psychological Complexity of Software Maintenance Tasks with the Halstead and McCabe Metrics", IEEE Transactions on Software Engineering, Volume SE-5, No. 2, March 1979.

Curtis, Bill, Sylvia B. Sheppard, and Phil Millman, "Third Time Charm: Stronger Prediction of Programmer Performance by Software Complexity Metrics", IEEE, 1979.

Denicoff, Marvin, "Software Metrics: Paradigms and Processes", Computer Science and Statistics: Proceedings of the 13th Symposium on the Interface, Springer-Verlag, Inc., New York, NY, 1981, pp. 205-207.

Deutsch, Michael S., "Application of an Automated Verification System", Software Verification and Validation Realistic Project Approaches, Prentice Hall, Englewood Cliffs, NJ, 1982.

Dickson, et al., "Quantitative Analysis of Software Reliability", Proceedings of 1972 Reliability and Maintainability Symposium, Annals of Assurance Science, IEEE, Catalog No. 72CHO577-7R, pp. 148-157.

Drummond, S., "Measuring Applications Development Performance", Datamation, February 1985, pp. 103-108.

Duncan, Otis Dudley, Introduction to Structural Equation Models, Academic Press, Inc., New York, NY, 1975, pp. 1-168.

Dunsmore, H. E., "Software Metrics: An Overview of an Evolving Methodology", Information Processing and Management, Volume 20, No. 1-2, 1984, pp. 183-192.

Ejiogu[1], Lem O., "A Simple Measure of Software Complexity", Computerworld, Volume 18, No. 14, April 2, 1984.

Ejiogu[2], Lem O., "Software Structure: Its Characteristic Polynomials", SIGPLAN Notices, Volume 19, No. 12, December 1984, pp. 1-7.

Ejiogu, Lem O., "The Critical Issues of Software Metrics", SIGPLAN Notices, Volume 22, No. 3, March 1987, pp. 1-6.

Ejiogu, Lem O., "A Unified Theory of Software Metrics", Softmetrix, Inc., Chicago, IL, 1988, pp. 232-238.

Ejiogu, Lem O., "Beyond Structured Programming: An Introduction to the Principles of Applied Software Metrics", Structured Programming, Springer-Verlag, Inc., New York, NY, 1990.

Elshoff, James L., "A Numerical Profile of Commercial PL/I Programs", GMR-1927, General Motors Corporation Research Laboratories, Warren, MI, September 1975.

Elshoff, James L., "Measuring Commercial PL/I Programs Using Halstead's Criteria", SIGPLAN Notices, Volume 11, No. 5, May 1976, pp. 38-46.

Elshoff, James L., "The PEEK Measurement Program", GMR-4208, General Motors Corporation Research Laboratories, Warren, MI, November 15, 1982.

Elshoff, James L., "Characteristic Program Complexity Measures", GMR-4446R, General Motors Corporation Research Laboratories, Warren, MI, December 5, 1983.

Elwell, D. and N. VanSuetendael, "Software Quality Metrics", Digital Systems Validation Handbook-Volume II, DOT/FAA/CT-88/10, U.S. Department of Transportation, Federal Aviation Administration, February 1989.

Evangelist, W. M., "Software Complexity Metric Sensitivity to Program Structuring Rules", The Journal of Systems and Software, Volume 3, No. 3, September 1983.

Farr, Leonard and Henry J. Zagorski, "Quantitative Analysis of Programming Cost Factors: A Progress Report", Economics of Automatic Data Processing. ICC Symposium Proceedings 1965 Rome, North-Holland, Amsterdam, Holland, 1965, pp. 167-180.

Federal Aviation Regulations, Part 25, Airworthiness Standards: Transport Category Airplanes, Subpart F, Section 25.1309.

Ferdinand, A. E., "A Theory of System Complexity", International Journal of General Systems, Volume 1, No. 1, 1974.

Feuche, Mike, "Attention is Being Generated by Complexity Metrics Tools", MIS Week, Volume 9, No. 9, February 29, 1988.

Feuer, Alan, R. and Edward B. Fowlkes, "Some Results from an Empirical Study on Computer Software", Bell Telephone Laboratories, 1979.

Fisher, R. A., The Design of Experiments, Oliver and Boyd, London, United Kingdom, 1935.

Fitzsimmons, Ann B., "Relating the Presence of Software Errors to the Theory of Software Science", Proceedings of the Eleventh Hawaii International Conference on Systems Sciences, Western Periodicals, 1978.

Fitzsimmons, Ann B. and Tom L. Love, "A Review and Evaluation of Software Science", ACM Computing Surveys, Volume 10, No. 1, March 1978, pp. 1-18.

Funami, Y. and M. H. Halstead, "A Software Physics Analysis of Akiyama's Debugging Data", Proceedings of the MRI 24th International Symposium: Software Engineering, Polytechnic Press, New York, NY, 1976.

Gaffney, Jr., J. E., "Metrics in Software Quality Assurance", ACM '81 - Tutorial Abstract, November 9, 1981.

Gaffney, Jr., John E., "Software Metrics: A Key to Improved Software Development Management", Computer Science and Statistics: Proceedings of the 13th Symposium on the Interface, Pittsburgh, PA, March 12 and 13, 1981, pp. 211-220.

Gaffney, Jr., John E. and Richard R. Reynolds, "Specifying Performance Requirements for a Degradable System - The Case of an Unmanned Weather Station", CMG XIV International Conference on Computer Performance Evaluation, Crystal City, VA, December 1983.

Gaffney, Jr., John E., "Estimating the Number of Faults in Code", IEEE Transactions on Software Engineering, Volume SE-10, No. 4, July 1984, pp. 459-464.

Gaffney, Jr., John E., "The Impact on Software Development Costs of Using HOL's", IEEE Transactions on Software Engineering, Volume SE-12, No. 3, March 1986.

Gaffney, Jr., John E. and Charles F. Davis, "An Approach to Estimating Software Errors and Availability", SPC-TR-88-007, Software Productivity Consortium, Herndon, VA, March 1988.

Gibson, Virginia R. and James A. Senn, "System Structure and Software Maintenance Performance," Communications of the ACM, March 1989, Volume 32, No. 3, pp. 347-358.

Gilb, T., Software Metrics, Winthrop, Inc., Cambridge, MA, 1977.

Gordon, R. D. and M. H. Halstead, "An Experiment Comparing FORTRAN Programming Times with the Software Physics Hypothesis", TR 167, Purdue University, October 17, 1975.

Gordon, R. D. and M. H. Halstead, "An Experiment Comparing FORTRAN Programming Times with the Software Physics Hypothesis", AFIPS Conference Proceedings - National Computer Conference, Volume 45, 1976, pp. 935-937.

Gordon[1], Ronald D., "Measuring Improvements in Program Clarity", IEEE Transactions on Software Engineering, Volume SE-5, No. 2, March 1979, pp. 79-90.

Gordon[2], Ronald D., "A Qualitative Justification for a Measure of Program Clarity", IEEE Transactions on Software Engineering, Volume SE-5, No. 2, March 1979, pp. 121-128.

Gorla, Narasimhaiah, Alan C. Benander, and Barbara A. Benander, "Debugging Effort Estimation Using Software Metrics", IEEE Transactions on Software Engineering, Volume 16, No. 2, February 1990, pp. 223-231.

Gould, John D., "Some Psychological Evidence on How People Debug Computer Programs", International Journal of Man-Machine Studies, Volume 7, pp. 151-181.

Gremillion, L. L., "Determinants of Program Repair Maintenance Requirements", Communications of the ACM, Volume 27, No. 8, pp. 826ff.

Halstead, Maurice H., "Natural Laws Controlling Algorithm Structure", SIGPLAN Notices, Volume 7, No. 2, 1972.

Halstead, Maurice H. and P. M. Zislis, "Experimental Verification of Two Theorems of Software Physics", CSD-TR-97, Purdue University, Lafayette, IN, June 12, 1973.

Halstead, Maurice H., Elements of Software Science, Elsevier North Holland Publishing Company, Inc., New York, NY, 1977, pp. 19-26.

Halstead, Maurice H., "Advances in Software Science", Advances in Computers, Volume 18, NY Academic, 1979.

Harrison[1], W., "MAE: A Syntactic Metric Analysis Environment", The Journal of Systems and Software, Volume 8, 1988, pp. 57-62.

Harrison[2], W., "Using Software Metrics to Allocate Testing Resources", Journal of Management Information Systems, Volume 4, No. 4, Spring 1988.

Harrison, W. A., "Applying McCabe's Complexity Measure to Multiple-Exit Programs", Software - Practice & Experience, Volume 14, No. 10, October 1984.

Harrison, W. and C. Cook, "Are Deeply Nested Conditionals Less Readable?", The Journal of Systems and Software, Volume 6, 1986, pp.335-341.

Harrison, W. and C. Cook, "Insights On Improving the Maintenance Process Through Software Measures", Oregon State University, Computer Science Department, Corvallis, OR, March 1990.

Harrison, Warren and Curtis Cook, "A Reduced Form For Sharing Software Complexity Data", Oregon State University, Computer Science Department, Corvallis, OR, 1983.

Harrison, W. and K. Magel, "A Complexity Measure Based On Nesting Level", The Journal of Systems and Software, Volume 6, 1986, pp. 335-341.

Harrison, W. et al., "Applying Software Complexity Metrics To Program Maintenance", Computer, Volume 15, No. 9, September 1988.

Hays, William L. and Robert L. Winkler, Statistics, Holt, Rinehart, and Winston, Inc., New York, NY, 1970.

Henry, S. and D. Kafura, "The Evaluation of Software Systems' Structure Using Quantitative Software Metrics", Software - Practice and Experience, Volume 14, No. 6, June 1984.

Henry, S., D. Kafura, and K. Harris, "On the Relationships Among Three Software Metrics", 1981 ACM Workshop/Symposium on Measurement and Evaluation of Software Quality, March 1981.

IEEE Computer Society, IEEE Guide for the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software, IEEE Std 982.1-1988, April 1989.

IEEE Computer Society, IEEE Guide for the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software, IEEE Std 982.2-1988, June 1989.

Jones, Capers, "Building a Better Metric", Computerworld, Volume 22, No. 25, June 20, 1988, pp. 38-39.

Jones, T. C., "Measuring Programming Quality and Productivity", IBM Systems Journal, Volume 17, No. 1, 1978, pp. 39-63.

Kafura, Dennis and Geereddy R. Reddy, "The Use of Software Complexity Metrics in Software Maintenance", IEEE Transactions on Software Engineering, Volume SE-13, No. 3, March 1987, pp 335-343.

Keller, Steven E., "Using Metrics to Specify Software Quality", Ada Quality Quarterly, Systems Division of Dynamics Research Corporation, Andover, MA, 1989.

Kelvin, W. T., "Popular Lectures and Addresses, 1981-1984", Workshop on Quantitative Software Models for Reliability, Complexity, and Cost, IEEE, New York, NY, 1980.

Kernighan, B. W. and P. J. Plauger, The Elements of Programming Style, McGraw-Hill, New York, NY, 1974.

Kitchenham, B. A., "Measures of Programming Complexity", ICL Technical Journal, Volume 2, No. 3, 1981, pp. 298ff.

Krause, K. W., R. W. Smith, and M. A. Goodwin, "Optimal Software Test Planning through Automated Network Analysis", Proceedings IEEE Computer Software Reliability Symposium, 1973.

Krause, K. W., R. W. Smith, and M. A. Goodwin, "Optimal Software Test Planning through Automated Network Analysis", TRW-SS-73-01, TRW Engineering and Integration Division, Redondo Beach, CA, April 1973, pp. 1-5.

Kreitzberg, Charles B. and Ben Shneiderman, FORTRAN Programming: A Spiral Approach, Harcourt Brace Jovanovich, Inc., 1982.

Kruger, Gregory A., "Project Management Using Software Reliability Growth Models", Hewlett-Packard Journal, Volume 39, No. 3, June 1988, pp. 30-35.

Lasky, Jeffrey A., Alan R. Kaminsky, and Wade Boaz, "Software Quality Measurement Methodology Enhancements Study Results", RADC-TR-89-317, RADC, Griffiss Air Force Base, NY, January 1990.

Lassez, J.-L., D. Van Der Knijff, J. Shepherd, and C. Lassez, "A Critical Examination of Software Science", The Journal of Systems and Software, Volume 2, December 1981, pp. 105-112.

Lennselius, Bo, Claes Wohlin, and Ctirad Vrana, "Software Metrics: Fault Content Estimation and Software Process Control", Microprocessors and Microsystems, Volume 11, No. 7, September 1987, pp. 365-375.

Lew, Ken S., Tharam S. Dillon, and Kevin E. Forward, "Software Complexity and Its Impact on Software Reliability", IEEE Transactions on Software Engineering, Volume 14, No. 11, November 1988, pp. 1645-1655.

Li, H. F. and W. K. Cheung, "An Empirical Study of Software Metrics", IEEE Transactions on Software Engineering, Volume SE-13, No. 6, June 1987.

Lind, Randy K. and K. Vairavan, "An Experimental Investigation of Software Metrics and Their Relationship to Software Development Effort", IEEE Transactions on Software Engineering, Volume 15, No. 5, May 1989, pp. 649-653.

Littlewood, B. (edited by), Software Reliability - Achievement and Assessment, Blackwell Scientific Publications, Oxford, United Kingdom, 1987.

Love, L. T. and A. B. Bowman, "An Independent Test of the Theory of Software Physics", SIGPLAN Notices, Volume 11, November 1976, pp. 42-49.

Low, Graham C. and D. Ross Jeffery, "Function Points in the Estimation and Evaluation of the Software Process", IEEE Transactions on Software Engineering, Volume 16, No. 1, January 1990, pp. 64-71.

Mannino, Phoebe, Bob Stoddard, and Tammy Sudduth, "The McCabe Software Complexity Analysis as a Design and Test Tool", Texas Instruments Technical Journal, Volume 7, No. 2, March-April 1990, pp. 41-53.

McAuliffe, Daniel, "Measuring Program Complexity", Computer, Volume 21, No. 6, June 1988, pp. 97-98.

McCabe, T. J., "A Complexity Measure", IEEE Transactions on Software Engineering, Volume SE-2, No. 4, 1976.

McCabe, Thomas J., Structured Testing: A Software Testing Methodology Using the Cyclomatic Complexity Metric, U.S. Department of Commerce, National Bureau of Standards, Special Publication #500-99, Washington, DC, 1982.

McCabe, Thomas J., "The Cyclomatic Complexity Metric", Hewlett Packard Journal, April 1989.

McCabe, Thomas J. and Charles W. Butler, "Design Complexity Measurement and Testing", Communications of the ACM, December 1989, pp. 1415-1425.

McCall, Jim A., Paul K. Richards, and Gene F. Walters, "Factors in Software Quality", RADC-TR-77-369, Volumes I, II, and III, RADC, Griffiss Air Force Base, NY, November 1977.

McClure, Carma L., "A Model for Program Complexity Analysis", Proceedings of the Third International Conference on Software Engineering, IEEE, New York, NY, 1978, p. 149-157.

Millman, P. and B. Curtis, "A Matched Project Evaluation of Modern Programming Practices (Management Report)", RADC-TR-80-6, Volume 1 of 2, RADC, Griffiss Air Force Base, NY, 1980.

Millman, P. and B. Curtis, "A Matched Project Evaluation of Modern Programming Practices (Scientific Report)", RADC-TR-80-6, Volume 2 of 2, RADC, Griffiss Air Force Base, NY, 1980.

Mohanty, Siba N., "Models and Measurements for Quality Assessment of Software", Computing Surveys, Volume 11, No. 3, September 1972, pp. 251-275.

Moranda, P. B., "Is Software Science Hard?", ACM Computer Surveys, Volume 10, No. 4, December 1978, pp. 503-504.

Munson, John C. and Khoshgoftaar, "Applications of a Relative Complexity Metric for Software Project Management", The Journal of Systems and Software, Volume 12, No. 3, July 1990, pp. 283-291.

Murine^{1}, Gerald E., "The Application of Software Quality Metrics", Phoenix Conference on Computers and Communications, IEEE, Carlsbad, CA, 1983.

Murine^{2}, Gerald E., "Improving Management Visibility Through the Use of Software Quality Metrics", Proceedings, Seventh International Computer Software and Applications Conference, Chicago, IL, November 7-11, 1983.

Murine^{3}, Gerald E., "On Validating Software Quality Metrics", 4th Annual Phoenix Conference, Phoenix, AZ, March 1985.

Murine[4], Gerald E., "The Role of Software Quality Metrics in the Software Quality Evaluation Process", Proceedings, Ninth International Computer Software and Applications Conference, Chicago, IL, October 1985.

Murine, Gerald E., "Software Measurement Techniques", Proceedings, Measurement Science Conference, Long Beach, CA, January 1988.

Murine, Gerald E. and Rita Cerv, "The New Science of Software Quality Metrology", Proceedings, Measurement Science Conference, Irvine, CA, January 1986.

Musa, John D., "Faults, Failures, and a Metrics Revolution", IEEE Software, Volume 6, No. 2, March 1989, pp. 85 and 91.

Myers, G. J., Software Reliability, John Wiley and Sons, Inc., New York, NY, 1976.

Myers, G. J., "An Extension to the Cyclomatic Measure of Program Complexity", SIGPLAN Notices, Volume 12, No. 10, 1977.

Myers, G. J., Composite/Structured Design, Van Nostrand Reinhold Company, New York, NY, 1978.

Myers, G. J., The Art of Software Testing, John Wiley and Sons, Inc. New York, NY, 1979.

Ntafos, Simeon C., "A Comparison of Some Structural Testing Strategies", IEEE Transactions on Software Engineering, Volume 14, No. 6, June 1988, pp. 868-874.

Ohba, M., "Software Reliability Analysis Models", IBM Journal of Research and Development, Volume 28, No. 4, July 1984, pp. 428-443.

Oldehoeft, R. R., "A Contrast Between Language Level Measures", IEEE Transactions on Software Engineering, Volume SE-3, No. 6, November 1977, pp. 476-478.

Ottenstein, Linda M., "Quantitative Estimates of Debugging Requirements", IEEE Transactions on Software Engineering, Volume SE-5, No. 5, September 1979, pp. 504-514.

Ottenstein, Linda M., "Predicting Numbers of Errors Using Software Science", ACM SIGMETRICS (1981 ACM Workshop/Symposium on Measurement and Evaluation of Software Quality), Volume 10, March 1981, pp. 157-167.

Parnas, D. L., "On the Criteria to be Used in Decomposing Systems Into Modules", Communications of the ACM, Volume 15, No. 12, December 1972.

Perlis, A. J., F. G. Sayward, and M. Shaw (Editors), Software Metrics: An Analysis and Evaluation, MIT Press, Cambridge, MA, 1981.

Perrone, Giovanni, "Program Measures Complexity of Software", PC Week, Volume 5, No. 13, March 29, 1988.

Perry, W. E., "Lack of Measurements Plague IS Assessments", Information Systems News, May 2, 1983.

Pierce, Patricia, Richard Hartley, and Suellen Worrells, "Software Quality Measurement Demonstration Project II", RADC-TR-87-164, RADC, Griffiss Air Force Base, NY, October 1987.

Pippenger, Nicholas, "Complexity Theory", Scientific American, June 1978, pp. 114-124.

Prather, R. E., "Theory of Program Testing - An Overview", Bell System Technical Journal, Volume 62, No. 10, Part 2, December 1983.

Prather, R. E., "An Axiomatic Theory of Software Complexity Measure", The Computer Journal, Volume 27, No. 4, November 1984.

Quantitative Software Management Inc., "Hanscom AFB gets a Handle on Vendor Performance", QSM Perspectives, McLean, VA, Fall 1990.

Radio Technical Commission for Aeronautics, Software Considerations in Airborne Systems and Equipment Certification, Document No. RTCA/DO-178A, March 22, 1985.

Ramamurthy, Bina and Austin Melton, "A Synthesis of Software Science Measures and the Cyclomatic Number", IEEE Transactions on Software Engineering, Volume 14, No. 8, August 1988, pp. 1116-1121.

Reed, Fred, "Lab Picks Program to Monitor Ada Code", Federal Computer Week, Volume 2, No. 32, August 8, 1988.

Reynolds, R. G., "Metrics to Measure the Complexity of Partial Programs", The Journal of Systems and Software, Volume 4, No. 1, April 1984.

Robillard, Pierre N. and Germinal Boloix, "The Interconnectivity Metrics: A New Metric Showing How a Program is Organized", The Journal of Systems and Software, Volume 10, No. 1, July 1989, pp. 29-39.

Schneidewind, N. F. and H. M. Hoffman, "An Experiment in Software Error Data Collection and Analysis", IEEE Transactions on Software Engineering, Volume SE-5, No. 3, May 1979, pp. 276-286.

Selby, Richard W. and Adam A. Porter, "Learning from Examples: Generation and Evaluation of Decision Trees for Software Resource Analysis", IEEE Transactions on Software Engineering, Volume 14, No. 12, December 1988, pp. 1743-1757.

Shatz, Sol M., "Towards Complexity Metrics for Ada Tasking", IEEE Transactions on Software Engineering, Volume 14, No. 8, August 1988, pp. 1122-1127.

Shen, Vincent Y., Samuel D. Conte, and H. E. Dunsmore, "Software Science Revisited: A Critical Analysis of the Theory and its Empirical Support", IEEE Transactions on Software Engineering, Volume SE-9, No. 2, March 1983, pp. 155-165.

Shneiderman, B., Software Psychology: Human Factors in Computer and Information Systems, Winthrop Publishing Co., Cambridge, MA, 1980, pp. 94-120.

Shumskas, Anthony F., "A Layered Software Test and Evaluation Strategy for a Layered Defense System", ITEA Journal, Volume XI, No. 3, 1990.

Siyan, Karanjit S., "Coping with Complex Programs", Dr. Dobb's Journal of Software Tools, Volume 14, No. 3, March 1989.

Society of Automotive Engineers, Fault/Failure Analysis for Digital Systems and Equipment, Aerospace Recommended Practice 1834, August 7, 1986.

Software Engineering Research Review - Quantitative Software Models, Data and Analysis Center for Software (DACS), Griffiss Air Force Base, NY, March 1979.

Sullivan, J. E., "Measuring the Complexity of Computer Software", MTR-2648, MITRE Corporation, Bedford, MA, 1973.

Sunazuka, Toshihiko, Motoei Azuma, and Noriko Yamagishi, "Software Quality Assessment Technology", Proceedings, 8th International Conference on Software Engineering, August 28-30, 1985, pp. 142-148.

Szulewski, Paul A., Mark H. Whitworth, Philip Buchan, and J. Barton DeWolf, "The Measurement of Software Science Parameters in Software Designs", ACM SIGMETRICS (1981 ACM Workshop/Symposium on Measurement and Evaluation of Software Quality), Volume 10, March 1981, pp. 89-94.

Taft, Darryl K., "Quality Analysis Aids Ada Programming", Government Computer News, Volume 7, No. 7, April 1, 1988.

Takahashi, Muneo and Yuji Kamayachi, "An Empirical Study of a Model for Program Error Prediction", IEEE Transactions on Software Engineering, Volume 15, No. 1, January 1989, pp. 82-86.

Thayer, T. A., M. Lipow, and E. C. Nelson, "Software Reliability Study", TRW Software Series, TRW-SS-76-03, March 1976.

U.S. Department of the Air Force, Air Force Systems Command Software Management Indicators, AFSCP 800-14, January 20, 1986.

U.S. Department of the Air Force, Air Force Systems Command Software Quality Indicators, AFSCP 800-43, January 31, 1986.

U.S. Department of Defense, Defense System Software Quality Program, Military Standard DOD-STD-2168, August 1, 1979.

U.S. Department of Defense, Defense System Software Development, Military Standard DOD-STD-2167A, June 4, 1985.

U.S. Department of Transportation, Federal Aviation Administration, Digital Systems Validation Handbook-Volume II, DOT/FAA/CT-88/10, February 1989.

U.S. Department of Transportation, Federal Aviation Administration, "System Design and Analysis", Advisory Circular, No. 25.1309-1A, June 1988.

Van Der Knijff, D. J. J., "Software Physics and Program Analysis", The Austra-lian Computer Journal, Volume 10, No. 3, August 1978, pp. 82-86.

Van Der Poel, K. G. and S. R. Schach, "A Software Metric for Cost Estimation and Efficiency Measurement in Data Processing System Development", The Journal of Systems and Software, Volume 3, No. 3, September 1983.

Verilog Products Catalog, Verilog USA Inc., Alexandria, VA, January 1990.

Verilog, "Logiscope Automated Code Analyzer, Technical Presentation", December 1989.

Voldman, J. et al., "Fractal Nature of Software - Cache Interaction", IBM Journal of Research and Development, Volume 27, No. 2, March 1983, pp. 164-170.

Walsh, T. J., "A Software Reliability Study Using a Complexity Measure", Proceedings of the National Computer Conference, AFIPS, 1979.

Walters, G. F., "Applications of Metrics to a Software Quality Management (QM) Program", Software Quality Management, Petrocelli, New York, NY, 1979.

Ward, W. T., "Software Defect Prevention Using McCabe's Complexity Metric", Hewlett-Packard Journal, April 1989.

Warthman, James L., "Software Quality Measurement Demonstration Project I", RADC-TR-87-247, RADC, Griffiss Air Force Base, NY, December 1987.

Weyuker, Elaine J., "Evaluating Software Complexity Measures", IEEE Transactions on Software Engineering, Volume 14, No. 9, September 1988, pp. 1357-1365.

Woodfield, S. N., V. Y. Shen, and H. E. Dunsmore, "A Study of Several Metrics for Programming Effort", Journal of Systems and Software, Volume 2, December 1981, pp. 97-103.

Woodward, Martin R., Michael A. Hennell, and David Hedley, "A Measure of Control Flow Complexity in Program Text", IEEE Transactions on Software Engineering, Volume SE-5, No. 1, January 1979, pp. 45-50.

Yu, Tze-Jie, Vincent Y. Shen, and Hubert E. Dunsmore, "An Analysis of Several Software Defect Models", IEEE Transactions on Software Engineering, Volume 14, No. 9, September 1988, pp. 1261-1270.

Zislis, Paul M., "An Experiment in Algorithm Implementation", CSD-TR-96, Department of Computer Science, Purdue University, Lafayette, IN, June 1973.

Zweben, S. H., "A Study of the Physical Structure of Algorithms", IEEE Transactions on Software Engineering, Volume SE-3, No. 3, May 1977, pp. 250-258.

Zweben, Stuart H., "Software Physics: Resolution of an Ambiguity in the Counting Procedure", TR 93, Purdue University, Lafayette, IN.

GLOSSARY

ANTAGONISTIC QUALITY FACTORS. Quality Factors with conflicting attributes.

BINARY SEARCH. A searching algorithm in which the search population is repeatedly divided into two equal or nearly equal sections.

BITS. Binary digits.

CODE. The subset of software which exists for the sole purpose of being loaded into a computer to control it.

COMPLEMENTARY QUALITY FACTORS. Quality Factors with interrelated attributes.

CONTROL STRUCTURES. Programming constructs which direct the flow of control.

EQUIVALENCE STATEMENT. A FORTRAN statement which equates two variable names.

HARDWARE. The physical components of a computer.

METRIC. A measure.

MODULE. A unit of code which implements a function.

MONOTONIC FUNCTION. A function in which a certain change in the measure always represents a certain change in the property being measured, where either change is simply an increase or decrease in magnitude.

OBJECT CODE. The translation of source code that is loaded into a computer.

OPERANDS. The variables or constants on which the operators act.

OPERATORS. Symbols which affect the value or ordering of operands.

OPTIMIZING COMPILER. A computer program which, while translating source code into object code, removes inefficiencies from the code.

PROGRAM. A detailed set of instructions for accomplishing some purpose.

QUALITY MEASURE. A repeatable, monotonic relationship relating measures of objects (a set of numbers) to subjective qualities.

SOFTWARE METRIC. A measure of software objects.

SOFTWARE QUALITY FACTOR. Any software attribute that contributes either directly or indirectly, positively or negatively, toward the objectives for the system in which the software resides.

SOFTWARE QUALITY METRIC. (1) A measure that relates measures of the software objects (the symbols) to the software qualities (quality factors). (2) The measure of a software quality factor.

SOFTWARE. Computer programs and the documentation associated with the programs.

SOURCE CODE. Code that can be read by people.

STROUD NUMBER. The total number of elementary mental discriminations that a person makes per second.

SUBROUTINE. A self-contained body of code which can be called by other routines to perform a function.

WELL-BEHAVED FUNCTION. A smooth mathematical relationship.

ACRONYMS AND ABBREVIATIONS

λ	Language Level
$1/E_0$	Average number of discriminations a person is likely to make for each bug introduced into the code.
AC	Advisory Circular
ACT	Analysis of Complexity Tool
AFSC	Air Force Systems Command
AFSCP	AFSC Pamphlet
AM	Anomaly Management
AMC	Army Materiel Command
AP	Application Independence
ARP	Aerospace Recommended Practice
ASTROS	Advanced Systematic Techniques for Reliable Operational Software
$\hat{B}$	Number of Bugs (estimated)
BAT	Battlemap Analysis Tool
CAD	Computer Automated Design
CAM	Computer Automated Manufacturing
CE	Certification Engineer
CFR	Code of Federal Regulations
cgs	Centimeters/Grams/Seconds
CP	Completeness
CPU	Central Processing Unit
CR/LF	Carriage Return/Line Feed
CSCI	Computer Software Configuration Item
D	Program Difficulty
DAP	Data Analysis Processor
DR	Direct Ratio (Average)
DS	Direct Score
E	Programming Effort
EOF	End of File
EP	Effectiveness-Processing
ESD	Electronic Systems Division
FAA	Federal Aviation Administration
FAR	Federal Aviation Regulation
FC	Function Count
ff.	and the following pages
GE	Generality
FP	Function Point
h	Height of an Individual Node
H	Height of a Tree
HOL	High Order Language
I	Intelligence Content
IEEE	Institute of Electrical and Electronics Engineers
IFC	Information Flow Complexity
I/O	Input/Output
ISO	International Standards Organization
L	Program Level
$\hat{L}$	Estimated Program Level

$\hat{N}$	Estimated Length
LSDB	Launch Support Data Base
mks	Meters/Kilograms/Seconds
η	Vocabulary of a Program
η_1	Number of Unique Operators
η_2	Number of Unique Operands
η_1^*	Minimum Number of Unique Operators
η_2^*	Number of Different Input and Output Parameters
N	Implementation Length
N_1^*	Minimum number of operator occurrences
N_2^*	Minimum number of operand occurrences
N_1	Total Number of Operator Occurrences
N_2	Total Number of Operand Occurrences
PC	Personal Computer
PCA	Processing Complexity Adjustment
PROM	Programmable Read-Only Memory
RADC	Rome Air Development Center
RAM	Random-Access Memory
rpm	Revolutions Per Minute
R_t	Twin Number of the Root
RTCA	Radio Technical Commission for Aeronautics
RWP	Real-Time Weather Processor
S	Stroud Number
SAE	Society of Automotive Engineers
SAMTEC	Space and Missile Test Center
S_c	Structural Complexity
SD	Self-Descriptiveness
SI	Simplicity
SO	Second Order (Average)
SQA	Software Quality Assurance
SQF	Software Quality Factor
SQM	Software Quality Metrics
SQPP	Software Quality Program Plan
SRS	Software Requirement Specification
$\hat{T}$	Estimated Programming Time
SSS	System/Segment Specification
V	Volume
V^*	Potential Volume
WSO	Weighted Second Order (Average)