IEEE Standard for a Software Quality Metrics Methodology

Circuits and Devices

Communications Technology

IEEE Computer Society

Sponsored by the Software Engineering Standards Subcommittee of the Technical Committee on Software Engineering

Electromagnetics and Radiation

Energy and Power

Industrial Applications

Signals and Applications

Standards Coordinating Committees

IEEE Std 1061-1992

IEEE

Published by the Institute of Electrical and Electronics Engineers, Inc., 345 East 47th Street, New York, NY 10017, USA.

March 12, 1993 *SH15842*

IEEE Standard for a Software Quality Metrics Methodology

Sponsor

Software Engineering Standards Subcommittee
of the
Technical Committee on Software Engineering
of the
IEEE Computer Society

Approved December 13, 1992

IEEE Standards Board

Abstract: A methodology for establishing quality requirements and identifying, implementing, analyzing, and validating the process and product of software quality metrics is defined. The methodology spans the entire software life cycle. Although this standard includes examples of metrics, this standard does not prescribe specific metrics.
Keywords: direct metric, factor, metrics framework, software quality metric, subfactor

---

The Institute of Electrical and Electronics Engineers, Inc.
345 East 47th Street, New York, NY 10017-2394, USA

ISBN 1-55937-277-X

IEEE Standards documents are developed within the Technical Committees of the IEEE Societies and the Standards Coordinating Committees of the IEEE Standards Board. Members of the committees serve voluntarily and without compensation. They are not necessarily members of the Institute. The standards developed within IEEE represent a consensus of the broad expertise on the subject within the Institute as well as those activities outside of IEEE that have expressed an interest in participating in the development of the standard.

Use of an IEEE Standard is wholly voluntary. The existence of an IEEE Standard does not imply that there are no other ways to produce, test, measure, purchase, market, or provide other goods and services related to the scope of the IEEE Standard. Furthermore, the viewpoint expressed at the time a standard is approved and issued is subject to change brought about through developments in the state of the art and comments received from users of the standard. Every IEEE Standard is subjected to review at least every five years for revision or reaffirmation. When a document is more than five years old and has not been reaffirmed, it is reasonable to conclude that its contents, although still of some value, do not wholly reflect the present state of the art. Users are cautioned to check to determine that they have the latest edition of any IEEE Standard.

Comments for revision of IEEE Standards are welcome from any interested party, regardless of membership affiliation with IEEE. Suggestions for changes in documents should be in the form of a proposed change of text, together with appropriate supporting comments.

Interpretations: Occasionally questions may arise regarding the meaning of portions of standards as they relate to specific applications. When the need for interpretations is brought to the attention of IEEE, the Institute will initiate action to prepare appropriate responses. Since IEEE Standards represent a consensus of all concerned interests, it is important to ensure that any interpretation has also received the concurrence of a balance of interests. For this reason IEEE and the members of its technical committees are not able to provide an instant response to interpretation requests except in those cases where the matter has previously received formal consideration.

Comments on standards and requests for interpretations should be addressed to:

Secretary, IEEE Standards Board
445 Hoes Lane
P.O. Box 1331
Piscataway, NJ 08855-1331
USA

IEEE Standards documents are adopted by the Institute of Electrical and Electronics Engineers without regard to whether their adoption may involve patents on articles, materials, or processes. Such adoption does not assume any liability to any patent owner, nor does it assume any obligation whatever to parties adopting the standards documents.

Introduction

(This introduction is not a part of IEEE Std 1061-1992, IEEE Standard for a Software Quality Metrics Methodology.)

At the time this standard was completed, the Software Quality Metrics Methodology Working Group had the following membership:

Norman Schneidewind, *Chair* **George Klammer,** *Vice Chair*
Chris Baldwin and Elizabeth Banks, *Editors* **Ron Braun and Craig Fuget,** *Secretaries*

Martha Amis
James Anderson
Nelson Andrews
John Bowen
Fletcher Buckley
S.C. Chang
David Classick
Taz Daughtrey
Raymond Day
Deborah De Toma
Rose Der
James Dinkey
Ruth Euler
Michael Evangelist
David Favor
Thomas Kurihara
Hal Larson
Violet Foldes
Al Freund
Oscar Garcia
John Girard
Randy Greene
Warren Harrison
Clark Hay
Dan Hocking
Meinhard Hoffmann
Robert Holibaugh
John Horch
Bud Jones
William Junk
Maria Kaufmann
Philip Marriott
Denis Meredith
Mary Mikhail
Celia Modell
Jai Navlakha
Wilma Osborne
Art Price
Horst Richter
Carl Seddio
Raghu Singh
Karen Snow
William Turner
John Walker
Gene Walters
Paul Wolfgang

The following persons were on the balloting committee that approved this standard for submission to the IEEE Standards Board:

Wolf Arfvidson
Richard L. Aurbach
Motoei Azuma
Bruce M. Bakken
Bakul Banerjee
David Barber
Boris Beizer
Leo Beltracchi
Mordechai Ben-Menachem
H.R. Berlack
William J. Boll
Kathleen L. Briggs
Bruce Brocka
A. Winsor Brown
William Bryan
Fletcher Buckley
Margaret K. Butler
Jung K. Chung
Won L. Chung
Antonio Cicu
Francois Coallier
Christopher Cooke
Stuart Corcoran
Stewart Crawford
Patricia W. Daggett
Taz Daughtrey
Claudia Dencker
Peter A. Denny
Einar Dragstedt
Peter Dyson
Mary L. Eads
Dr. Leo G. Egan
Joseph C. Evans
John W. Fendrich
Roger G. Fordham
Kirby Fortenberry
John Franklin
David Gelperin
Yair Gershkovitch
Shirley Gloss-Soler
John Garth Glynn
Victor M. Guarnera
David A. Gustafson
George B. Hawthorne
Mark Heinrich
Charles P. Hollocker
David Johnson III
S. Jones
Richard Karcich
S. Kasad
Peter Klopfenstein
Robert Kosinski
Joseph Krupinski
Thomas M. Kurihara
Lak Ming Lam
Rebecca Ann Lamb
Renee Lamb
John B. Lane
F.C. Lim
Bertil Lindberg
Ben Livson
Dieter Look
J. Maayan
Harold Mains
Henry A. Malec
David M. Marks
Philip C. Marriott
Roger Martin
Tomoo Matsubara
Scott D. Matthews
Ivano Mazza
John P. McArdle
Celia H. Modell
Celia H. Modell
Gerry Neidhart
Dennis E. Nickle
Paul G. Petersen
Shari Lawrence Pfleeger
John G. Phippen
Meir Razy
Frances A. Ruhlman
Julio Sanz
Stephen R. Schach
Hans Schaefer
Norman Schneidewind
Roger W. Scholten
Gregory D. Schumacher
Carl S. Seddio
Robert W. Shillato
David M. Siefert
Al Sorkowitz
Richard Staunton
Robert Thibodeau
Leonard L. Tripp
Mark-Rene Uchida
Margaret Updike
Udo Voges
Dolores Wallace
John W. Walz
Paul A. Willis
Andrew Wilson
Paul A. Wolfgang
Paul R. Work
Alfred W. Yonda
Janusz Zalewski

When the IEEE Standards Board approved this standard on Dec. 3, 1992, it had the following membership:

Marco W. Migliaro, *Chair* **Donald C. Loughry,** *Vice Chair*
Andrew G. Salem, *Secretary*

Dennis Bodson
Paul L. Borrill
Clyde Camp
Donald C. Fleckenstein
Jay Forster*
David F. Franklin
Ramiro Garcia
Thomas L. Hannan
Donald N. Heirman
Ben C. Johnson
Walter J. Karplus
Ivor N. Knight
Joseph Koepfinger*
Irving Kolodny
D. N. "Jim" Logothetis
Lawrence V. McCall
T. Don Michael*
John L. Rankine
Wallace S. Read
Ronald H. Reimer
Gary S. Robinson
Martin V. Schneider
Terrance R. Whittemore
Donald W. Zipse

*Member Emeritus

Also included are the following nonvoting IEEE Standards Board liaisons:

Satish K. Aggarwal
James Beall
Richard B. Engelman
David E. Soffrin
Stanley Warshaw

Rachel Auslander
IEEE Standards Project Editor

Contents

IEEE Standard for a Software Quality Metrics Methodology

1. Overview

This standard is divided into five clauses. Clause 1 provides the scope of this standard. Clause 2 provides a set of definitions that are defined more fully in the standard, but are provided here as a quick reference. Clause 3 states the purpose of software quality metrics. Clause 4 provides a framework for software quality metrics. Clause 5 provides a methodology for software quality metrics, which is mandatory when applying this standard. Also in this standard are four annexes that are included for illustrative and reference purposes only.

1.1 Scope

This standard provides a methodology for establishing quality requirements and identifying, implementing, analyzing, and validating the process and product of software quality metrics. This methodology applies to all software at all phases of any software life cycle structure.

This standard does not prescribe specific metrics. However, the annexes include examples of metrics together with two complete examples of the use of this standard.

1.2 Audience

This standard is intended for those associated with the acquisition, development, use, support, maintenance, and audit of software. The standard is particularly aimed at those measuring or assessing the quality of software.

This standard can be used by

- An acquisition/project manager to identify, define, and prioritize the quality requirements for a system
- A system developer to identify specific traits that should be built into the software in order to meet the quality requirements
- A quality assurance/control/audit organization and a system developer to evaluate whether the quality requirements are being met
- A system maintainer to assist in change management during product evolution
- A user to assist in specifying the quality requirements for a system

2. Definitions

2.1 critical range: Metric values used to classify software into the categories of acceptable, marginal, or unacceptable.

2.2 critical value: Metric value of a validated metric that is used to identify software that has unacceptable quality.

2.3 direct metric: A metric applied during development or during operations that represents a software quality factor (for example, mean time to software failure for the factor reliability).

2.4 factor: *See:* **quality factor**. For the sake of brevity, the term "factor" is used in place of the term "quality factor" in this standard.

2.5 factor sample: A set of factor values that is drawn from the metrics database and used in metrics validation.

2.6 factor value: A value of the direct metric that represents a factor. (*See:* **metric value.**)

2.7 measure: A way to ascertain or appraise value by comparing it to a standard. To apply a metric.

2.8 measurement: The act or process of measuring. A figure, extent, or amount obtained by measuring.

2.9 metric: *See:* **software quality metric**. For the sake of brevity, the term "metric" is used in place of the term "software quality metric" in this standard.

2.10 metrics framework: A tool used for organizing, selecting, communicating, and evaluating the required quality attributes for a software system. A hierarchical breakdown of factors, subfactors, and metrics for a software system.

2.11 metrics sample: A set of metric values that is drawn from the metrics database and used in metrics validation.

2.12 metric validation: The act or process of ensuring that a metric correctly predicts or assesses a quality factor.

2.13 metric value: A metric output or an element that is from the range of a metric.

2.14 predictive assessment: The process of using a predictive metric(s) to predict the value of another metric.

2.15 predictive metric: A metric applied during development and used to predict the values of a software quality factor.

2.16 process step: Any task performed in the development, implementation, or maintenance of software (for example, identifying the software components of a system as part of the design).

2.17 process metric: A metric used to measure characteristics of the methods, techniques, and tools employed in developing, implementing, and maintaining the software system.

2.18 product metric: A metric used to measure the characteristics of the documentation and code.

2.19 quality attribute: A characteristic of software, or a generic term applying to factors, subfactors, or metric values.

2.20 quality factor: A management-oriented attribute of software that contributes to its quality.

2.21 quality requirement: A requirement that a software attribute be present in software to satisfy a contract, standard, specification, or other formally imposed document.

2.22 quality subfactor: A decomposition of a quality factor or quality subfactor to its technical components.

2.23 sample software: Software selected from a current or completed project from which data can be obtained for use in preliminary testing of data collection and metric computation procedures.

2.24 software component: A general term used to refer to a software system or an element, such as module, unit, data, or document.

2.25 software quality metric: A function whose inputs are software data and whose output is a single numerical value that can be interpreted as the degree to which software possesses a given attribute that affects its quality.
NOTE—This definition differs from the definition of quality metric found in IEEE Std 610.12-1990.

2.26 subfactor: *See:* **quality subfactor**. For the sake of brevity, the term "subfactor" is used in place of the term "quality subfactor" in this standard.

2.27 validated metric: A metric whose values have been statistically associated with corresponding quality factor values.

3. Purpose of software quality metrics

Software quality is the degree to which software possesses a desired combination of attributes. This desired combination of attributes shall be clearly defined; otherwise, assessment of quality is left to intuition. For the purposes of this standard, defining software quality for a system is equivalent to defining a list of software quality attributes required for that system. In order to measure the software quality attributes an appropriate set of software metrics shall be identified.

The purpose of software metrics is to make assessments throughout the software life cycle as to whether the software quality requirements are being met. The use of software metrics reduces subjectivity in the assessment of software quality by providing a quantitative basis for making decisions about software quality. However, the use of software metrics does not eliminate the need for human judgment in software evaluations. The use of software metrics within an organization or project is expected to have a beneficial effect by making software quality more visible.

More specifically, the use of this standard's methodology for measuring quality allows an organization to

- Achieve quality goals
- Establish quality requirements for a system at its outset
- Establish acceptance criteria and standards
- Evaluate the level of quality achieved against the established requirements
- Detect anomalies or point to potential problems in the system
- Predict the level of quality that will be achieved in the future
- Monitor changes in quality when software is modified
- Assess the ease of change to the system during product evolution
- Normalize, scale, calibrate, or validate a metric

To accomplish these aims, both process and product metrics should be represented in the system metrics plan.

4. Software quality metrics framework

The software quality metrics framework shown in figure 1 is designed to be flexible. It permits additions, deletions, and modifications of factors, subfactors, and metrics. Each level may be expanded to several sublevels. The framework can thus be applied to all systems and can be adapted as appropriate without changing the basic concept.

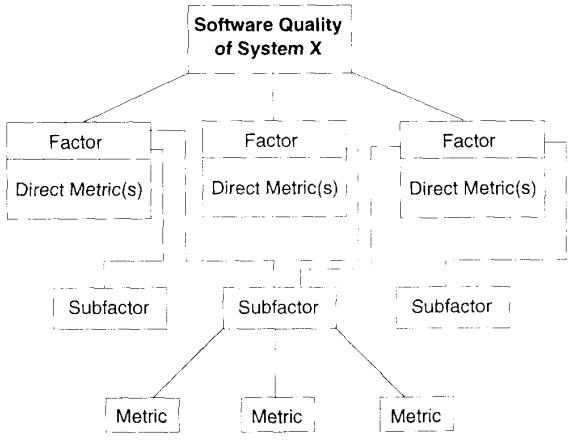

Figure 1—Software quality metrics framework

The first level of the software quality metrics framework hierarchy begins with the establishment of quality requirements by the assignment of various quality attributes. All attributes defining the quality requirements shall be agreed upon by the project team, and then the definitions should be established. Quality factors that represent management and user-oriented views are then assigned to the attributes. If necessary, subfactors are then assigned to each factor. Associated with each factor is a direct metric that serves as a quantitative representation of a quality factor. For example, a direct metric for the factor reliability could be mean time to failure. Each factor shall have one or more associated direct metrics and target values, such as one hour of execution time, that is set by project management. Otherwise, there is no way to determine whether the factor has been achieved. Some examples of factors are listed in annex A.

At the second level of the hierarchy are the quality subfactors that represent software-oriented attributes that indicate quality. These are obtained by decomposing each factor into measurable software attributes. Subfactors are independent attributes of software, and therefore may correspond to more than one factor (see annex A for further explanation). The subfactors are concrete attributes of software that are more meaningful than factors to technical personnel, such as analysts, designers, programmers, testers, and maintainers. The decomposition of factors into subfactors facilitates objective communication between the manager and the technical personnel regarding the quality objectives. Some examples of subfactors are listed in annex A.

At the third level of the hierarchy the subfactors are decomposed into metrics used to measure system products and processes during the development life cycle. Direct metric values, or factor values, are typically unavailable or expensive to collect early in the software life cycle. Therefore, metrics at the third level, that are validated against direct metrics, are used to estimate factor values early in the software life cycle.

From top to bottom the framework facilitates

— Establishment of quality requirements in terms of factors by managers early in a system's life cycle
— Communication of the established factors in terms of quality subfactors to the technical personnel
— Identification of metrics that are related to the established factors and subfactors

From bottom to top the framework enables the managerial and technical personnel to obtain feedback by

— Evaluating the software products and processes at the elementary metrics level
— Analyzing the metrics values to estimate and assess the quality factors

5. The software quality metrics methodology

The software quality metrics methodology is a systematic approach to establishing quality requirements and identifying, implementing, analyzing, and validating the process and product of software quality metrics for a software system. It spans the entire software life cycle and comprises five steps.

These steps shall be applied iteratively because insights gained from applying a step may show the need for further evaluation of the results of prior steps.

a) *Establish software quality requirements.* A list of quality factors is selected, prioritized, and quantified at the outset of system development or system change. These requirements shall be used to guide and control the development of the system and, on delivery of the system, to assess whether the system met the quality requirements specified in the contract.
b) *Identify software quality metrics.* The software quality metrics framework is applied in the selection of relevant metrics.
c) *Implement the software quality metrics.* Tools are either procured or developed, data is collected, and metrics are applied at each phase of the software life cycle.
d) *Analyze the software quality metrics results.* The metrics results are analyzed and reported to help control the development and assess the final product.
e) *Validate the software quality metrics.* Predictive metrics results are compared to the direct metrics results to determine whether the predictive metrics accurately measure their associated factors.

The documentation or output produced as a result of these steps is shown in table 1.

Table 1—Outputs of metrics methodology steps

Metric Methodology Step	Output
Establish software quality requirements	— Quality requirements
Identify software quality metrics	— Approved quality metrics framework — Metrics set — Cost-benefit analysis
Implement the software quality metrics	— Description of data items — Metrics/data item — Traceability matrix — Training plan and schedule
Analyze the software quality metrics results	— Organization and development process changes
Validate the software quality metrics	— Validation results

5.1 Establish software quality requirements

Quality requirements shall be represented in either of the following forms:

— *Direct metric value.* A numerical target for a factor to be met in the final product. For example, mean time to failure (MTTF) is a direct metric of final system reliability.
— *Predictive metric value.* A numerical target related to a factor to be met during system development. This is an intermediate requirement that is an early indicator of final system performance. For example, design or code errors may be early predictors of final system reliability.

5.1.1 Identify a list of possible quality requirements

Identify quality requirements that may be applicable to the software system. Use organizational experience, required standards, regulations, or laws to create this list. Annex A contains sample lists of factors and subfactors. In addition, list other system requirements that may affect the feasibility of the quality requirements. Consider acquisition concerns, such as cost or schedule constraints, warranties, and organizational self-interest. Do not rule out mutually exclusive requirements at this point. Focus on factor/direct metric combinations instead of predictive metrics.

All parties involved in the creation and use of the system shall participate in the quality requirements identification process.

5.1.2 Determine the actual list of quality requirements

Rate each of the listed quality requirements by importance. Importance is a function of the system characteristics and the viewpoints of the people involved. To determine the actual list of the possible quality requirements, follow the two steps below:

— *Survey all involved parties.* Discuss the relative priorities of the requirements with all involved parties. Have each group weigh the quality requirements against the other system requirements and constraints. Ensure that all viewpoints are considered.
— *Create the actual list of quality requirements.* Resolve the results of the survey into a single list of quality requirements. This shall involve a technical feasibility analysis of the quality requirements. The proposed factors for this list may have cooperative or conflicting relationships. Conflicts between requirements shall be resolved at this point. In addition, if the choice of quality requirements is in conflict with cost, schedule, or system functionality, one or the other shall be altered. Care shall be exercised in choosing the desired list to ensure that the requirements are technically feasible, reasonable, complementary, achievable, and verifiable. All involved parties shall agree to this final list.

5.1.3 Quantify each factor

For each factor, assign one or more direct metrics to represent the factor, and assign direct metric values to serve as quantitative requirements for that factor. For example, if "high efficiency" was one of the quality requirements from the previous item, the direct metric "actual resource utilization/allocated resource utilization" with a value of 90% could represent that factor. This direct metric value is used to verify the achievement of the quality requirement. Without it, there is no way to tell whether or not the delivered system meets its quality requirements.

The quantified list of quality requirements and their definitions again shall be approved by all involved parties.

5.2 Identify software quality metrics

When identifying software quality metrics, apply the software quality metrics framework, and perform a cost-benefit analysis. Then gain commitment to the metrics from all involved parties.

5.2.1 Apply the software quality metrics framework

Create a chart of the quality requirements based on the hierarchical tree structure found in figure 1. At this point, only the factor level must be complete. Next decompose each factor into subfactors as indicated in clause 4. The decomposition into subfactors must continue for as many levels as needed until the subfactor level is complete.

Using the software quality metrics framework, decompose the subfactors into measurable metrics. For each validated metric on the metric level, assign a target value and a critical value and range that should be achieved during development. The target values constitute additional quality requirements for the system.

The framework and the target values for the metrics shall be reviewed and approved by all involved parties.

To help ensure that metrics are used appropriately, only validated metrics (that is, either direct metrics or metrics validated with respect to direct metrics) shall be used to assess current and future product and process quality (see 5.5 for a description of the validation methodology and annex C for examples of applying the validation methodology). Nonvalidated metrics may be included for future analysis, but shall not be included as part of the system requirements. Furthermore, the metrics that are used shall be those that are associated with the quality requirements of the software project. However, given that the above conditions are satisfied, the selection of specific metrics as candidates for validation and the selection of specific validated metrics for application is at the discretion of the user of this standard. Examples of metrics and experiences with their use are given in annex B.

Document each metric using the format shown in table 2.

Table 2—Metrics set

Item	Description
Name	Name given to the metric.
Costs	Costs of using the metric (see 5.2.2.1).
Benefits	Benefits of using the metric (see 5.2.2.2).
Impact	Indication of whether a metric may be used to alter or halt the project (ask, *Can the metric be used to indicate deficient software quality?*).
Target value	Numerical value of the metric that is to be achieved in order to meet quality requirements. Include the critical value and the range of the metric.
Factors	Factors that are related to this metric.
Tools	Software or hardware tools that are used to gather and store data, compute the metric and analyze the results.
Application	Description of how the metric is used and what its area of application is.
Data items	Input values that are necessary for computing the metric values.
Computation	Explanation of the steps involved in the metrics computation.
Interpretation	Interpretation of the results of the metrics computation (see 5.4.1).
Considerations	Considerations of the appropriateness of the metric (for example, *Can data be collected for this metric? Is the metric appropriate for this application?*).
Training required	Training required to implement or use the metric.
Example	An example of applying the metric (see annex C).
Validation history	Names of projects that have used the metric, and the validity criteria the metric has satisfied.
References	References, such as list of projects, project details, etc., giving further details on understanding or implementing the metric.

5.2.2 Perform a cost-benefit analysis

Perform a cost-benefit analysis by identifying the costs of implementing the metrics, identifying the benefits of applying the metrics, and then applying the metrics set.

5.2.2.1 Identify the costs of implementing the metrics

Identify and document (see table 2) all the costs associated with the metrics in the metrics set. For each metric, estimate and document the following impacts and costs.

- *Metrics utilization costs.* The costs of collecting data; automating the metric value calculation (when possible); and analyzing, interpreting, and reporting the results.
- *Software development process change costs.* The set of metrics may imply a change in the development process.
- *Organizational structure change costs.* The set of metrics may imply a change in the organizational structure used to produce the software system.
- *Special equipment.* The hardware or software tools may have to be located, purchased, adapted, or developed to implement the metrics.
- *Training.* The quality assurance/control organization or the entire development team may need training in the use of the metrics and data collection procedures. If the introduction of metrics has caused changes in the development process, the development team may need to be educated about the changes.

5.2.2.2 Identify the benefits of applying the metrics

Identify and document the benefits that are associated with each metric in the metrics set (see table 2).

Some benefits to consider include the following:

- Identify quality goals and increase awareness of the goals in the software organization
- Provide timely feedback useful in developing higher quality software
- Increase customer satisfaction by quantifying the quality of the software before it is delivered to the customer
- Provide a quantitative basis for making decisions about software quality
- Reduce software life cycle costs by improving process efficiency based on metric data

5.2.2.3 Adjust the metrics set

Weigh the benefits, tangible and intangible, against the costs of each metric. If the costs exceed the benefits of a given metric, alter or delete it from the metrics set. On the other hand, for metrics that remain, make plans for any necessary changes to the software development process, organizational structure, tools, and training. In most cases it will not be feasible to quantify benefits. In these cases judgment shall be exercised in weighing qualitative benefits against quantitative costs.

5.2.3 Gain commitment to the metrics set

All involved parties shall review the revised metrics set to which the cost-benefit analysis has been added. The metrics set shall be formally adopted and supported by this group.

5.3 Implement the software quality metrics

To implement the software quality metrics, define the data collection procedures, use selected software to prototype the measurement process, then collect the data and compute the metrics values.

5.3.1 Define the data collection procedures

For each metric in the metric set, determine the data that will be collected and determine the assumptions that will be made about the data (for example, random sample, subjective or objective measure). The flow of data shall be shown from point of collection to evaluation of metrics. Describe or reference when and how tools are to be used and data storage procedures. Also, identify candidate tools. Select tools for use with the prototyping process. A traceability matrix shall be established between metrics and data items.

Identify the organizational entities that will directly participate in data collection including those responsible for monitoring data collection. Describe the training and experience required for data collection and the training process for personnel involved.

Describe each data item thoroughly, using the format shown in table 3.

Table 3—Description of a data item

Item	Description
Name	Name given to the data item.
Metrics	Metrics that are associated with the data item.
Definition	Straightforward description of the data item.
Source	Location of where the data originates.
Collector	Entity responsible for collecting the data.
Timing	Time(s) in life cycle at which the data is to be collected. (Some data items are collected more than once.)
Procedures	Methodology (for example, automated or manual) used to collect the data.
Storage	Location of where the data is stored.
Representation	Manner in which the data is represented, for example, its precision and format (e.g., Boolean, dimensionless, etc.).
Sample	Method used to select the data to be collected and the percentage of the available data that is to be collected.
Verification	Manner in which the collected data is to be checked for errors.
Alternatives	Methods that may be used to collect the data other than the preferred method.
Integrity	Who is authorized to alter this data item and under what conditions.

5.3.2 Prototype the measurement process

Test the data collection and metric computation procedures on selected software that will act a a prototype. If possible, the samples selected should be similar to the project(s) on which the metrics will later be used. An analysis shall be made to determine if the data is collected uniformly and if instructions have always been interpreted in the same manner. In particular, data requiring subjective judgments shall be checked to determine if the descriptions and instructions are clear enough to ensure uniform results.

In addition, the cost of the measurement process for the prototype shall be examined to verify or improve the cost analysis.

Use the results collected from the prototype to improve the metric descriptions (see table 2) and descriptions of data items (see table 3).

5.3.3 Collect the data and compute the metrics values

Using the formats in table 2 and table 3, collect and store data at the appropriate time in the life cycle. The data shall be checked for accuracy and proper unit of measure.

Data collection shall be monitored. If a sample of data is used, requirements such as randomness, minimum size of sample, and homogeneous sampling shall be verified. If more than one person is collecting the data, it shall be checked for uniformity.

Compute the metrics values from the collected data.

5.4 Analyze the software metrics results

Analyzing the results of the software metrics includes interpreting the results, identifying software quality, making software quality predictions, and ensuring compliance with requirements.

5.4.1 Interpret the results

The results shall be interpreted and recorded against the broad context of the project as well as for a particular product or process of the project. The differences between the collected metric data and the target values for the metrics shall be analyzed against the quality requirements. Substantive differences shall be investigated.

5.4.2 Identify software quality

Quality metric values for software components shall be determined and reviewed. Quality metric values that are outside the anticipated tolerance intervals (low or unexpected high quality) shall be identified for further study. Unacceptable quality may be manifested as excessive complexity, inadequate documentation, lack of traceability, or other undesirable attributes. The existence of such conditions is an indication that the software may not satisfy quality requirements when it becomes operational. Since many of the direct metrics that are usually of interest cannot be measured during software development (for example, reliability metrics), validated metrics shall be used when direct metrics are not available. Direct or validated metrics shall be measured for software components and process steps. The measurements shall be compared with critical values of the metrics. Software components whose measurements deviate from the critical values shall be analyzed in detail. The fact that a measurement deviates from a critical value does not necessarily mean that the software component will exhibit unacceptable quality during operation. Deviation from a critical value may occur because metrics are not infallible; they are only indicators of quality. What metrics indicate about quality during development may not be the quality achieved in operation. Depending on the results of the analysis, software components shall be redesigned (acceptable quality is achieved by redesign), scrapped (quality is so poor that redesign is not feasible), or not changed (deviations for critical metric values are judged to be insignificant).

Unexpected high quality metric values shall cause a review of the software development process as the expected tolerance levels may need to be modified or the process for identifying quality metric values may need to be improved.

5.4.3 Make software quality predictions

During development validated metrics shall be used to make predictions of direct metric values. Predicted values of direct metrics shall be compared with target values to determine whether to flag software components for detailed analysis. Predictions shall be made for software components and process steps. Software components and process steps whose predicted direct metric values deviate from the target values shall be analyzed in detail.

Potentially, prediction is very valuable because it estimates the metric of ultimate interest—the direct metric. However, prediction is difficult because it involves using validated metrics from an earlier phase of the life cycle (for example, development) to make a prediction about a different but related metric (direct metric) in a much later phase (for example, operations).

5.4.4 Ensure compliance with requirements

Direct metrics shall be used to ensure compliance of software products with quality requirements during system and acceptance testing (see 5.1). Direct metrics shall be measured for software components and process steps. These values shall be compared with target values of the direct metrics that represent quality requirements. Software components and process steps whose measurements deviate from the target values are noncompliant.

5.5 Validate the software quality metrics

For information on the statistical techniques used, see [B114], [B115], [B117],[1] or similar references.

5.5.1 Purpose of metrics validation

The purpose of metrics validation is to identify both product and process metrics that can predict specified quality factor values, which are quantitative representations of quality requirements. If metrics are to be useful, they shall indicate accurately whether quality requirements have been achieved or are likely to be achieved in the future. When it is possible to measure factor values at the desired point in the life cycle, these direct metrics are used to evaluate software quality. At some points in the life cycle, certain quality factor values (for example, reliability) are not available. They are obtained after delivery or late in the project. In these cases, other metrics are used early on in a project to predict quality factor values.

The history of the application of metrics indicates that predictive metrics were seldom validated (that is, it was not demonstrated through statistical analysis that the metrics measured software characteristics that they purported to measure.) However, it is important that predictive metrics be validated before they are used to evaluate software quality. Otherwise, metrics might be misapplied (that is, metrics might be used that have little or no relationship to the desired quality characteristics).

Although quality subfactors are useful when identifying and establishing factors and metrics, they need not be used in metrics validation, because the focus in validation is on determining whether a statistically significant relationship exists between predictive metric values and factor values.

Quality factors may be affected by multiple variables. A single metric, therefore, may not sufficiently represent any one factor if it ignores these other variables.

[1]The numbers in brackets correspond to those of the bibliographical references in annex D.

5.5.2 Validity criteria

To be considered valid, a predictive metric shall demonstrate a high degree of association with the quality factors it represents. This is equivalent to accurately portraying the quality condition(s) of a product or process. A metric may be valid with respect to certain validity criteria and invalid with respect to other criteria.

The person in the organization who understands the consequence of the values selected shall designate threshold values for the following:

V – square of the linear correlation coefficient
B – rank correlation coefficient
A – prediction error
P – success rate

A short numerical example follows the definition of each validity criterion. Detailed examples of the application of metrics validation are contained in annex C.

a) *Correlation.* The variation in the quality factor values explained by the variation in the metric values, which is given by the square of the linear correlation coefficient (R) between the metric and the corresponding factor, shall exceed V, where $R^2 > V$.
This criterion assesses whether there is a sufficiently strong linear association between a factor and a metric to warrant using the metric as a substitute for the factor, when it is infeasible to use the latter. For example, the correlation coefficient between a complexity metric and the factor reliability may be 0.8. The square of this is 0.64. Only 64% of the variation in the factor is explained by the variation in the metric. If V has been established as 0.7, the conclusion would be drawn that the metric is invalid (that is, there is insufficient association, or correlation between the metric and reliability). If this relationship is demonstrated over a representative sample of software components, the conclusion could be drawn that the metric is invalid.

b) *Tracking.* If a metric M is directly related to a quality factor F, for a given product or process, then a change in a quality factor value from F_{T1} to F_{T2}, at times T1 and T2, shall be accompanied by a change in metric value from M_{T1} to M_{T2}, which is the same direction (for example, if F increases, M increases). If M is inversely related to F, then a change in F shall be accompanied by a change in M in the opposite direction (for example, if F increases, M decreases). To perform this test, compute the coefficient of rank correlation (r) from n paired values of the factor and the metric. Each of the factor/metric pairs is measured at the same point in time, and the n pairs of values are measured at n points in time. The absolute value of (r) shall exceed B.
This criterion assesses whether a metric is capable of tracking changes in product or process quality over the life cycle.
For example, if a complexity metric is claimed to be a measure of reliability, then it is reasonable to expect a change in the reliability of a software component to be accompanied by an appropriate change in metric value (for instance, if the product increases in reliability, the metric value should also change in a direction that indicates the product has improved). That is, if Mean Time to Failure (MTTF) is used to measure reliability and is equal to 1000 hours during testing (T1) and 1500 hours during operation (T2), a complexity metric whose value is 8 in T1 and 6 in T2, where 6 less complex than 8, is said to track reliability for this software component. If this relationship is demonstrated over a representative sample of software components, the conclusion could be drawn that the metric can track reliability (that is, indicate changes in product reliability) over the software life cycle.

c) *Consistency.* If factor values F1, F2,..., Fn, corresponding to products or processes 1, 2, n, have the relationship F1 > F2> ..., Fn, then the corresponding metric values shall have the relationship M1 > M2 > ..., Mn. To perform this test, compute the coefficient of rank correlation (r) between paired values (from the same software components) of the factor and the metric; $|r|$ shall exceed B.
This criterion assesses whether there is consistency between the ranks of the factor values of a set of software components and the ranks of the metric values for the same set of software components.

Thus, this criterion is used to determine whether a metric can accurately rank, by quality, a set of products or processes.

For example, if the reliability of software components *X*, *Y*, and *Z*, as measured by MTTF, is 1000, 1500, and 800 hours respectively, and the corresponding complexity metric values are 5, 3, and 7, where low metric values are better than high values, the ranks for reliability and metric values, with 1 representing the highest rank, are as follows:

Software Component	Reliability Rank	Complexity Metric Rank
Y	1	1
X	2	2
Z	3	3

If this relationship is demonstrated over a representative sample of software components, the conclusion could be drawn that the metric is consistent and can be used to rank the quality of software components. For example, the ranks could be used to establish priority of testing and allocation of budget and effort to testing (that is, the worst software component would receive the most attention, largest budget, and most staff).

d) *Predictability.* If a metric is used at time T1 to predict a quality factor for a given product or process, it shall predict a related quality factor Fp_{T2} with an accuracy of

$$\left| \frac{Fa_{T2} - Fp_{T2}}{Fa_{T2}} \right| < A$$

where Fa_{T2} is the actual value of F at time T2.

This criterion assesses whether a metric is capable of predicting a factor value with the required accuracy.

For example, if a complexity metric is used during development to predict the reliability of a software component during operation (T2) to be 1200 hours MTTF (Fp_{T2}) and the actual MTTF that is measured during operation is 1000 hours (Fa_{T2}), then the error in prediction is 200 hours, or 20%. If the acceptable prediction error (A) is 25%, prediction accuracy is acceptable. If the ability to predict is demonstrated over a representative sample of software components, the conclusion could be drawn that the metric can be used as a predictor of reliability. For example, prediction could be used during development to identify those software components that need to be improved.

e) *Discriminative Power.* A metric shall be able to discriminate between high quality software components (for example, high MTTF) and low quality software components (for example, low MTTF). For instance, the set of metric values associated with the former should be significantly higher (or lower) than those associated with the latter.

This criterion assesses whether a metric is capable of separating a set of high quality software components from a set of low quality components. This capability identifies critical values for metrics that can be used to identify software components that may have unacceptable quality. The Mann-Whitney Test and the Chi-square test for differences in probabilities (Contingency Tables) can be used for this validation test.

For example, if all software components with a complexity metric value of greater than 10 (critical value) have a MTTF of 1000 hours and all components with a complexity metric value equal to or less than 10 have a MTTF of 2000 hours, and this difference is sufficient to pass the statistical tests, then the metric separates low from high quality software components. If the ability to discriminate is demonstrated over a representative sample of software components, the conclusion could be drawn that the metric can discriminate between low and high reliability components for quality assurance and other quality functions.

f) *Reliability.* A metric shall demonstrate the above correlation, tracking, consistency, predictability, and discriminative power properties for P percent of the applications of the metric.

This criterion is used to ensure that a metric has passed a validity test over a sufficient number or percentage of applications so that there will be confidence that the metric can perform its intended function consistently.
For example, if the required success rate (P) for validating a complexity metric against the predictability criterion has been established as 80%, and there are 100 software components, the metric shall predict the factor with the required accuracy for at least 80 of the components.

5.5.3 Validation procedure

Metrics validation shall include identifying the quality factors sample, identifying the metrics sample, performing a statistical analysis, and documenting the results.

5.5.3.1 Identify the quality factors sample

These factor values (for example, measurements of reliability), which represent the quality requirements of a project, were previously identified (see 5.1.3) and collected and stored (see 5.3.3). For validation purposes, draw a sample from the metrics database.

5.5.3.2 Identify the metrics sample

These metrics (for example, design structure) are used to predict or to represent quality factor values, when the factor values cannot be measured. The metrics were previously identified (see 5.2.1) and their data collected and stored, and values computed from the collected data (see 5.3.3). For validation purposes, draw a sample from the same domain (for example, same software components) of the metrics database as used in 5.5.3.1.

5.5.3.3 Perform a statistical analysis

The tests described in 5.5.2 shall be performed.

Before a metric is used to evaluate the quality of a product or process, it shall be validated against the criteria described in 5.5.2. If a metric does not pass all of the validity tests, it shall only be used according to the criteria prescribed by those tests (for example, if it only passes the tracking validity test, it shall be used only for tracking the quality of a product or process).

5.5.3.4 Document the results

Documenting results shall include the direct metric, predictive metric, validation criteria, and numerical results, as a minimum.

5.5.4 Additional requirements

Additional requirements, those being the need for revalidation, confidence in analysis results, and the stability of environment, are described in the following subclauses.

5.5.4.1 The need for revalidation

It is important to revalidate a predictive metric before it is used for another environment or application. As the software engineering process changes, the validity of metrics changes. Cumulative metric validation values may be misleading because a metric that has been valid for several uses may become invalid. It is wise to compare the one-time validation of a metric with its validation history to avoid being misled.

The following statements of caution should be noted:

— A validated metric may not necessarily be valid in other environments or future applications.
— A metric that has been invalidated may be valid in other environments or future applications.

5.5.4.2 Confidence in analysis results

Metrics validation is a continuous process. Confidence in metrics is increased over time as metrics are validated on a variety of projects and as the metrics database and sample size increases. Confidence is not a static, one-time property. If a metric is valid, confidence will increase with increased use (that is, the correlation coefficient will be significant at decreasing values of the significance level). Greatest confidence occurs when metrics have been tentatively validated based on data collected from previous projects. Even when this is the case, the validation analysis will continue into future projects as the metrics database and sample size grow.

5.5.4.3 Stability of environment

Practicable metrics validation shall be undertaken in a stable development environment (that is, where the design language, implementation language, or program development tools do not change over the life of the project in which validation is performed). In addition, there shall be at least one project in which metrics data have been collected and validated prior to application of the predictive metrics. This project shall be similar to the one in which the metrics are applied with respect to software engineering skills, application, size, and software engineering environment.

Validation and application of metrics shall be performed during the same life cycle phases on different projects. For example: if metric X is collected during the development phase of project A and the saved values are later validated with respect to direct metric Y, whose values are collected during the operations phase of project A, the metric X shall be used during the development phase of project B to assess direct metric Y with respect to the operations phase of project B.

Annexes

(These informative annexes are not a part of IEEE Std 1061-1992, IEEE Standard for a Software Quality Metrics Methodology, but are included for information only.)

Annex A
Examples of factors, subfactors and metrics, and the relationships among them

(informative)

Tables A1 and A2 illustrate descriptions of factors and subfactors, respectively. This selection was made to provide illustrative and usable definitions of factors and subfactors. Figure A1 illustrates a possible set of metrics for each one of the factors and its associated subfactors.

Note that these tables are not exhaustive. There are many possible sets of factors, subfactors, and metrics that can be used. Consult annex D for additional sources of this information.

Table A1—Sample factors

Factor	Description
Efficiency	An attribute that bears on the relationship of the level of performance to the amount of resources used under stated conditions.
Functionality	An attribute that bears on the existence of certain properties and functions that satisfy stated or implied needs of users.
Maintainability	An attribute that bears on the effort needed for specific modifications.
Portability	An attribute that bears on the ability of software to be transferred from one environment to another.
Reliability	An attribute that bears on the capability of software to maintain its level of performance under stated conditions for a stated period of time.
Usability	An attribute that bears on the effort needed for use (including preparation for used and evaluation of results) and on the individual assessment of such use by users.

Table A2—Sample subfactors

Factor	Subfactor	Description
Efficiency	Time economy	Capability of software to perform specified functions under stated or implied conditions within appropriate time frames.
	Resource economy	Capability of software to perform specified functions under stated or implied conditions, using appropriate amounts of resources.
Functionality	Completeness	The degree to which software possesses necessary and sufficient functions to satisfy user needs.
	Correctness	The degree to which all software functions are specified.
	Security	The degree to which software can detect and prevent information leak, information loss, illegal use, and system resource destruction.
	Compatibility	The degree to which new software can be installed without changing environments and conditions that were prepared for the replaced software.
	Interoperability	The degree to which software can be connected easily with other systems and operated.
Maintainability	Correctability	The degree of effort required to correct errors in software and cope with user complaints.
	Expandability	The degree of effort required to improve or modify the efficiency or functions of software.
	Testability	The effort required to test software.
Portability	Hardware independence	The degree to which software does not depend on specific hardware environments.
	Software independence	The degree to which software does not depend on specific software environments.
	Installability	The effort required to adjust software to a new environment.
	Reusability	The degree to which software can be reused in applications other than the original application.
Reliability	Nondeficiency	The degree to which software does not contain undetected errors.
	Error tolerance	The degree to which software will continue to work without a system failure that would cause damage to the users. Also, the degree to which software includes degraded operation and recovery functions.
	Availability	The degree to which software remains operable in the presence of system failures.

Table A2—Sample subfactors (*Continued*)

Factor	Subfactor	Description
Usability	Understandability	The amount of user effort required to understand software.
	Ease of learning	The degree to which user effort required to understand software is minimized.
	Operability	The degree to which the operation of software matches the purpose, environment, and physiological characteristics of users, including ergonomic factors such as color, shape, sound, etc.
	Communicativeness	The degree to which software is designed in accordance with the psychological characteristics of users.

Figure A1 maps a factor to subfactors to metrics. This is for illustrative purposes only. Note that, in general, mappings between factors, subfactors, and metrics may be one-to-one, one-to-many, or many-to-one.

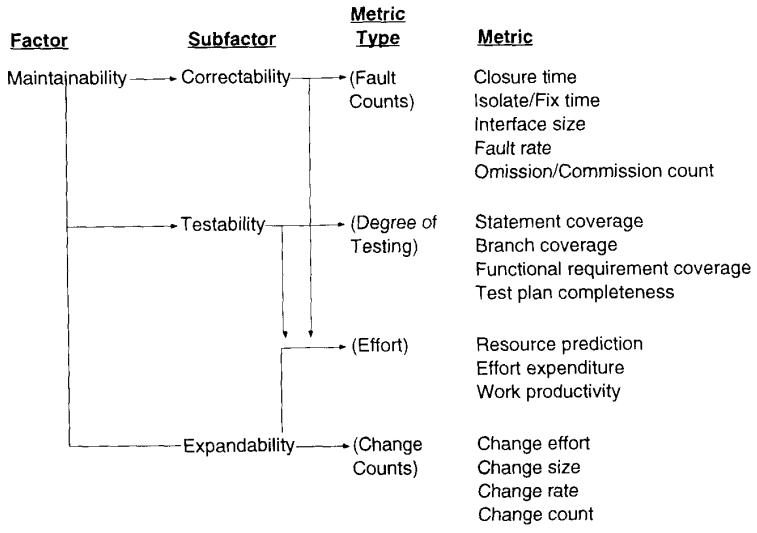

Figure A1—Sample diagram to show the relationships between factors, subfactors, and some metrics

Annex B
Sample metrics descriptions

(informative)

This annex provides the following:

a) Examples of how to compute software metrics
b) Detailed descriptions of some major metrics
c) Sample metric results

The examples are not intended to be all inclusive. Additional examples exist but, due to space limitations, all of them cannot be documented here.

B.1 Detailed metrics descriptions

The following summarizes some major metrics. The references listed below should be consulted for a complete understanding of the metrics.

Table B1—Example metrics computations

Factor/Subfactor	Computation
Completeness	Ratio of number of completed documents or software components to total number of documents or software components.
Consistency	Ratio of number of documents or software components that are free of contradictions to total number of documents or software components.
Correctness	Ratio of number of user mission objectives that have been correctly implemented in software components to total number of user mission objectives.
Interoperability	Ratio of number of interface specifications or software components whose interfaces are compatible to total number of interface specifications or software components.
Maintainability	Ratio of total error correction labor time to total number of corrections.
Reliability	Ratio of number of computer runs that produce correct outputs to total number of computer runs.
Testability	Identification of independent paths corresponding to McCabe metric for path testing.
Traceability	Ratio of number of documents or software components for this phase that can be traced to the previous phase to total number of documents or software components for this phase.

B.1.1 Halstead—software science

Purpose. To measure properties and structure of computer programs in order to predict the following for a computer program:

a) Length, volume, difficulty, and level
b) Number of errors
c) Effort and time required for development

Given variables

η_1 = number of distinct operators in a program
η_2 = number of distinct operands in a program
N_1 = number of occurrences of operators in a program
N_2 = number of occurrences of operands in a program

Compute the results.

η	= Program Vocabulary	=	$\eta_1 + \eta_2$
N	= Observed Program Length	=	$N_1 + N_2$
$\hat{N}$	= Estimated Program Length	=	$\eta_1(\log_2(\eta_1)) + \eta_2(\log_2(\eta_2))$
V	= Program Volume	=	$N(\log_2(\eta))$
D	= Program Difficulty	=	$(\eta_1/2)/(N_2/\eta_2)$
L	= Program Level	=	1/D
E	= Effort	=	V/L
$\hat{B}$	= Number of Errors	=	$(V)/E_0$ (E_0 is the programming error rate, $3000 \leq E_0 \leq 3200$)
$\hat{T}$	= Time (in seconds)	=	E/S (S is the Stroud number $5 \leq S \leq 20$)

Application considerations. A correlation may be demonstrated between a high degree of difficulty and a high number of errors and increased maintenance effort. A similar correlation may be demonstrated with very large programs.

References. Annex D.1, [B88] and annex D.2, [B116]

B.1.2 Boehm—constructive cost model (COCOMO)

Purpose. To estimate cost driver factors and lines of code in order to predict project effort, cost, and schedule.

Estimate KDSI_____ (Thousands of Delivered Source Instructions) for a project.

Choose project type and its associated difficulty factor.

Calculate reference value for effort and schedule (see annex D.2, [B113]).

Project Type	**Effort (SM = Staff Months)**	**Schedule (TDEV = Time)**
Stand-alone (Organic)	$SM = 2.4(KDSI)^{1.05}$	$TDEV = 2.5(SM)^{0.38}$
Mixed (Semi-detached)	$SM = 3.0(KDSI)^{1.12}$	$TDEV = 2.5(SM)^{0.35}$
Embedded	$SM = 3.6(KDSI)^{1.20}$	$TDEV = 2.5(SM)^{0.32}$

Choose values for cost driver factors.

Very Low	Low	Nom*	High	Very High	Value	
____	____	____	____	____	____	Required development schedule
____	____	____	____	____	____	Required software reliability
____	____	____	____	____	____	Database size
____	____	____	____	____	____	Product complexity
____	____	____	____	____	____	Execution time constraint
____	____	____	____	____	____	Main storage constraint
____	____	____	____	____	____	Virtual machine volatility
____	____	____	____	____	____	Analyst capability
____	____	____	____	____	____	Applications experience
____	____	____	____	____	____	Programmer capability
____	____	____	____	____	____	Virtual machine experience
____	____	____	____	____	____	Programming language experience
____	____	____	____	____	____	Computer turnaround time
____	____	____	____	____	____	Use of modern programming practices
____	____	____	____	____	____	Use of software tools

* Nom (nominal) = 1, indicating no change.

NOTE—These factors must be calibrated for a software development organization before the factors are applied (see part IV B, chapters 23–27 of Annex D.2, [B113]).

Distribute reference estimates to life cycle phase (by percentages).

Apply cost driver factors (phase sensitive and product sensitive).

Apply factors for maintained-code verses new-code.

Sum up the estimates by phase and product level.

Application considerations. Use of the COCOMO model may provide more accurate cost and schedule estimates when used on a consistent basis with training.

References. Annex D.2, [B113]

B.1.3 Albrecht—function points

Purpose. To measure the functions the software is to perform from requirements information in order to estimate the effort required to develop the software.

Count the following functions.

EI = External Inputs
EO = External Outputs
LF = Logical Internal Files
EF = External Interface Files
IQ = External Inquiries

Assign weights to the functions as follows and total.

Type	Complexity			Total
	Simple	Average	Complex	
EI	__× 3 =__	__× 4 =__	__× 6 =__	_____
EO	__× 4 =__	__× 5 =__	__× 7 =__	_____
LF	__× 7 =__	__×10 =__	__×15 =__	_____
EF	__× 5 =__	__× 7 =__	__×10 =__	_____
IQ	__× 3 =__	__× 4 =__	__× 6 =__	_____
FC = Total Unadjusted Function Points			=	_____

Given values for degrees of influence (DI)

DI	Description
0	Not present, or no influence if present
1	Insignificant influence
2	Moderate influence
3	Average influence
4	Significant influence
5	Strong influence throughout

Assign degree of influence (DI) values to the following characteristics.

DI	Characteristic
__	Data Communications
__	Distributed Functions
__	Performance
__	Heavily Used Configuration
__	Transaction Rate
__	On-line Entry
__	End-user Efficiency
__	On-line Update
__	Complex Processing
__	Reusability
__	Installation Ease

__ Operational Ease
__ Multiple Sites
__ Facilitate Changes
__ Total Degree of Influence (PC)

Compute the following results.

PCA = Processing Complexity Adjustment = 0.65 + (0.01 × PC)
FP = Function Points Measure = FC × PCA

Application considerations. Albrecht's function points may be measured earlier in the life cycle than some of the other metrics. The number of function points may be correlated with the lines of code and effort required for development.

References. Annex D.2, [B110] and [B111]

B.1.4 McCabe—cyclomatic complexity

Purpose. To measure the control flow structure in order to predict the following about a program:

a) Difficulty of understanding
b) Extent to which it is likely to contain defects

Count the following values.

P = number of control paths into the program
E = number of edges (transfers of control)
N = number of nodes (sequential group of statements containing only one transfer of control)

Compute the result. Cyclomatic Complexity $= E - N + 2P$

Application considerations. Use of McCabe's cyclomatic complexity metric may help to ensure that developers are sensitive to which programs are likely to be difficult to understand and which programs contain defects. The results may be correlated to lines of code as well as number of defects.

References. Annex D.2, [B112] and [B118]

B.2 Sample metrics results

The following lists were obtained in using metrics on various projects in terms of advantages, limitations, accuracy/precision, and effectiveness of the measurements and comments.

Within the tables *R* stands for simple linear correlation coefficient and *r* stands for Spearman rank order correlation coefficient.

B.2.1 Halstead effort

Definition. See B.1.1.

Advantages

a) Fairly easy to compute.
b) Can automate.
c) Tools exist.

Limitations

a) Cannot be computed until software is designed.
b) There is controversy about validity.
c) Does not represent all aspects of complexity.

Project	Accuracy/Precision	Effectiveness	Comments
NASA Goddard Software Engineering Laboratory (SEL)	$R = 0.65$ with programming effort.	No significant relationship was observed.	
	$r = 0.488$ with observed programming effort.		
	$R = 0.51$ with known errors.	Same as NASA	
Naval Air Dev. Center	$R = 0.46$ with total changes.		
Purdue Univ.	$R = 0.98$ with a number of errors.	High association with number of errors	
General Electric Co.	$R = 0.76$ with number of delivered errors.		
	$R = 0.20$ with median debug time.		
	$R = 0.78$ with the mean number of errors in debugging.		
	$R = -0.68$ with an understanding of program based on quiz.		

B.2.2 Halstead difficulty

Definition. See B.1.1.

Advantages. Same as Halstead Effort.

Limitations. Same as Halstead Effort.

Project	Accuracy/Precision	Effectiveness	Comments
NASA Goddard (SEL)	$R = 0.417$ with observed programming effort.	Same as Halstead Effort.	

B.2.3 McCabe complexity V(G)

Definition. See B.1.4.

Advantages

a) Easy to compute.
b) Tools exist.
c) Can apply to both code and preliminary design.

Limitations

a) Shall have some idea of design structure before it can be applied.
b) There is controversy about validity.
c) Does not represent all aspects of complexity.

Project	Accuracy/Precision	Effectiveness	Comments
NASA Goddard (SEL)	As module size varied between 50 and > 200 LOC, McCabe metric varied between 6.2 and 77.5, and errors/1000 LOC varied between 65.0 and 9.7. $r = 0.467$ with observed programming effort. $R = 0.65$ with time to locate and correct bugs.	No significant relationship was observed.	
Naval P. G. School	$R = 0.78$ with a number of errors. $R = 0.67$ with time to find errors. $R = 0.72$ with time to correct errors. Ratio of V(G) for error modules/V(G) for zero error modules = 2.79. Ratio of avg error finding time for high V(G) modules/avg error finding time for low V(G) modules = 1.94. Ratio of avg error correction time for high V(G) modules/avg error correction time for low V(G) modules = 1.72.	Good association when data partitioned between high/low V(G) and high/low error counts.	Obtain correct understanding and interpretation of directed graphs. Easy to automate.
Naval Air Dev. Center	$R = 0.46$ with known errors. $R = 0.43$ with total changes.	Same as Naval P. G. School.	Same as Naval P. G. School.

B.2.4 Source of software error

Definition. The count of errors by type (for example, related to error in or misunderstanding of functional specification).

Advantages. Relates problem to source of error so that problem can be fixed.

Limitations. Not easy to automate and not easy to identify source of error.

Project	Accuracy/Precision	Effectiveness	Comments
NASA Goddard (SEL)	44% of errors related to functional specs. 39% of errors were interface errors.	Provided good insight into sources of major errors.	

B.2.5 Number of changes that affect module

Definition. The count of changes to a module.

Advantages

a) Easy to compute.
b) Can automate.
c) Tools exist.

Limitations. Cannot use until software exists.

Project	Accuracy/Precision	Effectiveness	Comments
NASA Goddard (SEL)	$r = 0.469$ with observed programming effort	No significant relationship was observed.	

B.2.6 Defect metrics

Definition. The error count, time between errors, error rate, error correction count, and time needed to correct errors.

Advantages. Can evaluate reliability and design approach.

Limitations. Data may be difficult to collect.

Project	Accuracy/Precision	Effectiveness	Comments
Schneidewind reliability model applied to Naval Tactical Data System and Naval Surface Weapons Center error data.	Error prediction within + or – 10% of actual errors for 40-week prediction period.	Effective for indicating what to test and how long to test and ranking reliability of modules.	Difficult to get projects to collect the kind of data needed to feed the model. Data collection shall be automated to capture error data in real time. Model shall be automated.

B.2.7 Traceability

Definition. The identification of independent paths corresponding to the McCabe metric for path testing.

Advantages. Excellent for assessing whether software and documentation is fully integrated.

Limitations

a) Subject to various interpretations.
b) Difficult to define quantitatively.
c) Difficult to automate.

Project	Accuracy/Precision	Effectiveness	Comments
Naval Trident Command and Control System Maint. Agency	Traceability was able to pinpoint where certain Navy software specifications and standards were not adequate for doing software maintenance.	Demonstrated the infeasibility of using development standards for maintenance.	Tedious to do traceability manually. Need automated tool.

B.2.8 Number of program paths (Np)

Definition. The number of distinct execution sequences from entry node to exit node.

Advantages. Representative of execution sequences in a program.

Limitations. Can be difficult to compute because number of paths may be very large. Computer time may be excessive.

Project	Accuracy/Precision	Effectiveness	Comments
Naval P. G. School	$R = 0.76$ with a number of errors. $R = 0.90$ with time to find errors. $R = 0.65$ with time to correct errors. Ratio of avg Np for modules with errors/ avg Np for modules with no errors = 13.05.	Same as McCabe metric for Naval P. G. School.	Number of paths can be very large and even infinite. Difficult to automate.

Annex C
Examples of use of the methodology

(informative)

NOTE—Numerical values used in the two examples in this annex are for illustrative purposes only. These values would not necessarily apply in other situations.

This section presents two examples of the use of the methodology described in clause 5. One example is a mission critical example from the military sector, and the other example is an off-the-shelf software application from the commercial sector. There are two primary differences between the examples.

a) Emphasis in the mission critical example is on development-process quality requirements, while the emphasis in the commercial example is on end-user quality requirements.
b) The quality factors and subfactors used in the mission critical example differ significantly from those used by the commercial example. This is due to differing environments. Neither application of the standard is better than the other.

C.1 Mission critical example

A command and control system example is used to illustrate some of the steps outlined in clause 5. The command and control system used in this example is adapted from RADC-TR-85-37, Volume II, Final Technical Report, February, 1985. The system is described by the following characteristics and functions.

System:	Airborne Radar System
Life cycle:	15–20 years
Computing system:	Centralized, Redundant Processors
System functions:	—Surveillance and Identification
	—Threat Evaluation
	—Weapons Assignment/Control
	—Battlestaff Management
	—Communications

The software quality metrics methodology should be applied to each system function individually; however, for the purposes of this example it will be applied only to the surveillance and identification function.

C.1.1 Establish software quality requirements

Establish software quality requirements by first identifying a possible list. Next order the quality requirements and survey all involved parties.Then determine the actual list of quality requirements. When the list is final, quantify each factor.

C.1.1.1 Identify a list of possible requirements

The surveillance and identification system function can be decomposed into a number of smaller system functions. These functions are listed in the first column of table C1. Table C1 is a tabulation of the quality factors that are associated with these individual system functions. The process used to identify these functions will not be discussed in this example.

Table C1—Important software quality factors for surveillance and identification

	Software Quality Factor	Efficiency	Integrity	Reliability	Survivability	Usability	Correctness	Maintainability	Verifiability	Expandability	Flexibility	Interoperability	Portability	Reusability
System Function	*Surveillance and Identification*													
	Data Collection/Reduction	X		X	X						X	X		
	Dynamic Graphic Display	X		X		X								
	Target Recognition	X		X										
	Threat Detection/ Identification	X		X										
	Threat Display	X		X		X	X	X			X			
	Threat Response Aids	X	X	X		X	X	X	X		X			X

It is also helpful to consider the application and/or environmental characteristics of the system insofar as these characteristics have implications for particular quality factors. See table C2.

Table C2—Example of application/environmental characteristics and software quality factors

Application/Environmental Characteristics	Software Quality Factors
Human lives affected	Integrity
	Reliability
	Correctness
	Verifiability
	Survivability
Long life cycle	Maintainability
	Expandability
Experimental system or high rate of change	Flexibility
Many changes over life cycle	Flexibility
	Reusability
	Expandability
Real time application	Efficiency
	Reliability
	Correctness

Table C2—Example of application/environmental characteristics and software quality factors (*Continued*)

Application/Environmental Characteristics	Software Quality Factors
On-board computer application	Efficiency
	Reliability
	Correctness
	Survivability
Processing of classified information	Integrity
Interrelated systems	Interoperability

For example, the long life cycle of the Airborne Radar System dictates that specific quality factors—maintainability and expandability—be considered.

Finally, system-level requirements should be examined to determine if they have any software requirements implications. Table C3 is an example of the relationship between a system quality factor and software quality factors. The system availability, defined as the portion of total operational time that the system performs or supports critical functions, requires that the software be reliable, survivable, and maintainable.

Table C3—System/software quality factor correlation

Acquisition Concern	Performance				Design		Adaptation					
Software Quality Factor	Efficiency	Integrity	Reliability	Survivability	Usability	Correctness	Maintainability	Verifiability	Expandability	Flexibility	Interoperability	Portability	Reusability
System Quality Factor: Availability				X	X			X					

C.1.1.2 Order the quality requirements

Techniques that facilitate this step include conducting a quality requirements survey and analyzing the interrelationships among quality factors and life-cycle costs.

C.1.1.2.1 Survey all involved parties

The thirteen software quality factors listed in table C1 represent aspects of software quality that are recognized as being important for certain software products. A survey is used to identify and rate the quality factors for each software function.

Participants in the survey are given a copy of table C4, which defines the software quality factors, and are asked to indicate whether they consider the factor to be: very important (*E* for excellent quality), important (*G* for good quality), somewhat important (*A* for average quality), or either not important (blank) or not applicable (*NA*). Table C5 is the consensus view of the importance of each quality factor as it pertains to the surveillance and identification function.

Table C4—Software quality requirements survey

Acquisition Concern	Quality Factor	Definition
Performance	Efficiency	Relative extent to which a resource is utilized (for example, storage space, processing time, or communication time).
	Integrity	Extent to which the software will perform without failures due to unauthorized access to the code or data within a specified time period.
	Reliability	Extent to which the software will perform without failures within a specified time period.
	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.
	Usability	Relative effort for training or software operation (for example, familiarization, input preparation, execution, or output interpretation).
Design	Correctness	Extent to which the software conforms to its specifications and standards.
	Maintainability	Ease of effort for locating and fixing a software failure within a time period.
	Verifiability	Relative effort to verify the specified software operation and performance.
Adaptation	Expandability	Relative effort to increase the software capability or performance by enhancing current functions or by adding new functions or data.
	Flexibility	Ease of effort for changing the software missions, functions, or data to satisfy other requirements.
	Interoperability	Relative effort to couple the software of one system to the software of another system.
	Portability	Relative effort to transport the software for use in another environment (hardware configuration and/or software system environment).
	Reusability	Relative effort to convert a software component for use in another application.

Table C5—Software quality factor identification form: survey results

Acquisition Concern	Performance				Design			Adaptation					
Software Quality Factor	Efficiency	Integrity	Reliability	Survivability	Usability	Correctness	Maintainability	Verifiability	Expandability	Flexibility	Interoperability	Portability	Reusability
System Function: Surveillance & Identification	E	E	E	E	G	E	E	G	G	G	G	N/A	G

C.1.1.2.2 Determine the actual list of quality requirements

Analyzing interrelationships among factors is important in order to assess the impact of either low or high quality levels of some factors on other factors. For example, in the surveillance and identification function, a high quality level (*E*) was specified for reliability and a good quality level (*G*) for verifiability. However, even though metric scores for reliability might be high, if the software is difficult to verify then the actual reliability could be low. A complete specification for reliability would require that if reliability is given a high quality level, then verifiability should also be high. Thus, the *G* for verifiability in table C5 is changed to *E* in table C6.

Table C6—Software quality factor identification form: revised requirements

Acquisition Concern	Performance				Design			Adaptation					
Software Quality Factor	Efficiency	Integrity	Reliability	Survivability	Usability	Correctness	Maintainability	Verifiability	Expandability	Flexibility	Interoperability	Portability	Reusability
System Function: Surveillance & Identification	G**	E	E	G**	E**	E	E	E**	G	G	G	N/A	G

NOTE—** indicates a change from a previous step.

Further analysis of the interrelationships among factors explores the fact that factors may have cooperative or conflicting relationships. These relationships are presented in table C7.

Table C7—Quality factor relationships

Software Quality Factor	Efficiency	Flexibility	Integrity	Interoperability	Maintainability	Portability	Reliability	Survivability	Correctness	Reusability	Verifiability	Usability	Expandability
Software Quality Factor													
Efficiency													
Flexibility	X												
Integrity	X												
Interoperability	X	O	X										
Maintainability	X	O											
Portability	X			O									
Reliability	X				O								
Survivability	X	X	X										
Correctness		O											
Reusability	X		X		O	O	O	X	O				
Verifiability	X				O	O	O		O	O			
Usability	X	O			O		O	O			X		
Expandability	X		X		O			X	O				

X means the factors conflict.
O means the factors support one another.
A blank space means that there is no relationship.

In order to arrive at a prioritized set of factors and associated metrics, one shall assess the potential costs and benefits for performing any additional quality-related activities. A detailed cost analysis of this type is not easy and will initially rest on judgment rather than data. This analysis might show that in the initial stage in which requirements are examined and allocated costs for all factors increase. That is because quality requirements may be difficult to determine for a given function. During full-scale development, the costs for reliability, for example, may include cost additions due to additional verification effort. However, cost savings may be realized through fewer errors and through emphasis on error handling and error avoidance. Actual cost savings and cost additions will be known via the data.

The main conflict in the requirements initially assigned to the surveillance and identification function is between efficiency and those conflicting factors assigned an *E*. Those factors are: integrity, reliability, survivability, maintainability, and verifiability.

A solution is to lower the requirement for efficiency to *G* and to require efficient processing hardware as a means of relaxing the requirement for highly efficient software. Also, a further analysis of table C7 indicates that survivability conflicts with other factors, and usability is relatively supportive of other factors. As a consequence, survivability and usability are changed from *E* and *G* respectively in table C5 to *G* and *E* in table C6. The revised quality requirements for surveillance and identification are shown in table C6.

C.1.1.3 Quantify each factor

A direct metric was assigned to each quality factor and a direct metric value was chosen as the requirement for that factor. The following are the direct metrics and the direct metric values used for the reliability factor.

Factor	Direct Metric	Acceptance Value
Reliability	Errors per HOL	Statement < 0.05
Reliability	Errors per HOL	Procedure < 0.60

C.1.2 Identify software quality metrics

When identifying software quality metrics, apply the software quality metrics framework and perform a cost-benefit analysis. Then gain commitment to the metrics from management.

C.1.2.1 Apply the software quality metrics framework

Once the quality factors have been selected, it is then necessary to select the subfactors for these quality factors.

Since integrity, reliability, usability, correctness, maintainability, and verifiability are considered to be the most important quality factors, subfactors from these shall be considered. The subfactors related to these factors are:

Factors	Subfactors
Integrity	System Accessibility
Reliability	Accuracy Anomaly Management Simplicity
Usability	Operability Training
Correctness	Completeness Consistency Traceability
Maintainability	Consistency Visibility Modularity Self-descriptiveness Simplicity
Verifiability	Visibility Modularity Self-descriptiveness Simplicity

The subfactor simplicity is related to three quality factors. The subfactors modularity, self-descriptiveness, visibility, and consistency are related to two quality factors. This might be a consideration in choice of metrics if trade-offs become necessary.

For each subfactor, one or more metrics may be required. These metrics will vary in different phases of the life cycle. As an example, metrics will be considered for modularity and simplicity in the various phases of the life cycle. Note that this is not a complete set of metrics. The phases of the life cycle considered will be software requirements analysis, preliminary design, detailed design, and coding and testing. The following are example metrics descriptions.

a) *Software Requirements Analysis*

Modularity

— Modular Implementation M1: Are all software functions and computer software systems developed according to structured design techniques?
— Modular Design M2: What is the average cohesive value of all software functions in this software system?

Simplicity

— Design Structure S1: Are there diagrams identifying all functions in a structured fashion?
— Design Structure S2: Has a programming standard been established?

b) *Preliminary Design*

Modularity

— Modular Implementation M1: Are all software functions and software components developed according to structured design techniques?
— Modular Design M2: What is the average cohesive value of all top level software components in this system?

Simplicity

— Design Structure S2: Has a programming standard been established?

c) *Detailed Design*

Modularity

— Modular Design M2: What is the cohesive value of each software component?
— Design Structure M3: Is control always returned to the calling software component when execution is completed?

Simplicity

— Coding Simplicity S3: Is the flow of control simple?
— Cyclomatic Number S4: What is the cyclomatic number (complexity) of each software component?
— Size S5: How large is each software component (for example, High Order Language (HOL) procedure)?

d) *Coding and Testing*

Modularity

— Modular Design M2: What is the cohesive value of each software component?
— Design Structure M3: Is control always returned to the calling software component when execution is completed?

Simplicity

— Coding Simplicity S3: Is the flow of control simple?
— Cyclomatic Number S4: What is the cyclomatic number (complexity) of each software component?
— Size S5: How large is each software component (for example, HOL procedure)?

C.1.2.1.1 Metric documentation

For each of the sample metrics given above, an example of the metric description is given in table 2 located in 5.2.1.

Table C8—Metrics set: metric description #1

Item	Description
Name	Modular Implementation M1
Costs	2 labor weeks plus training
Benefits	This metric serves as a check to assure that proper design techniques are used.
Impact	Software that does not satisfy these metrics should be examined and possibly corrected.
Target Value	The target value is 1.
Factors	Maintainability and verifiability
Tools	No specific tools required.
Application	This metric is used to check proper design technique.
Data Items	List of structured design techniques together with evaluation of functions and software systems
Computation	Metric requires a *yes* or *no* answer based on subjective evaluation of whether all software development has been done according to structured design techniques.
Interpretation	This metric will be factored in with other metrics to derive an estimate of modularity. It is important that this metric be evaluated as a problem indicator, but statistical regression analysis would not be appropriate.
Considerations	Value of the metric is subjective.
Training Required	To properly use this metric would require substantial knowledge of structured design techniques.
Example	N/A
Validation History	None for this application
References	RADC-TR-85-37

Table C9—Metrics set: metric description #2

Item	Description
Name	Modular Design M2
Costs	4–6 labor weeks
Benefits	If used to enforce high value of cohesion, should provide improved modularity.
Impact	If value is less than 0.4, further examination is required.
Target Value	The target value is an average equal to 0.4 or higher.
Factors	Maintainability and verifiability
Tools	No specific tools required.
Application	Used to determine modular strength of the appropriate portion of software (phase dependent).
Data Items	Cohesion values of appropriate portion of software (for example, software component)
Computation	Cohesion values of appropriate software are determined (see reference below as well as RADC-TR-85-37) and an average is calculated.
Interpretation	Since the metric is subjective, the results will depend on the ability and judgment of persons performing the measurements. However, a low score would certainly indicate problems.
Considerations	Somewhat difficult to implement. Determination of metric values somewhat subjective.
Training Required	Due to the subjective nature of metric values, substantial knowledge of the subject is required.
Example	Module — Cohesion value A — 0.5 B — 0.6 C — 0.5 D — 0.7 E — 0.7 Sum — 3.0 Average — 0.6
Validation History	None for this application
References	Annex D.2, [B119]

Table C10—Metrics set: metric description #3

Item	Description
Name	Design Structure M3
Costs	4–6 labor weeks
Benefits	The metric checks that one aspect of software development is done properly.
Impact	If the answer is 0, the programming characteristics of other software components should be checked.
Target Value	The target value is 1.
Factors	Reliability, maintainability, and verifiability
Tools	No specific tools are required.
Application	The metric is used to check that control is always returned to the calling software component, thus retaining the proper control structure.
Data Items	Representation of the control structure of appropriate software segments in usable form
Computation	Metric requires a *yes* or *no* answer based on whether proper call returns have been used for all control structures.
Interpretation	If the answer to the metric is *no* then there is an indication of poor software design.
Considerations	Appropriate as a flag to indicate problems.
Training Required	Need to be able to read appropriate description of control structures.
Example	N/A
Validation History	None for this application
References	RADC-TR-85-37

Table C11—Metrics set: metric description #4

Item	Description
Name	Design Structure S1
Costs	1–2 labor weeks
Benefits	By using the metric to enforce use of diagrams representing functions in a structured manner, the diagrams produced form a useful tool for use in proper development of the software system.
Impact	Low impact. This metric alone would probably not affect development.
Target Value	The target value is 1.
Factors	Reliability, maintainability, and verifiability
Tools	No specific tools are required.
Application	Check on quality (or existence) of diagram representation of functions.
Data Items	Diagrams of functions of the system
Computation	Metric requires a *yes* or *no* answer based on subjective examination of all diagrams to determine whether all functions have been identified in a structured fashion.
Interpretation	Since the metric is subjective, the results will depend on the ability and judgment of persons performing the measurements.
Considerations	Although hard to apply objectively, this metric is a useful check to make sure that necessary work is done properly.
Training Required	Application of this metric requires use of experienced personnel.
Example	N/A
Validation History	None for this application
References	RADC-TR-85-37

Table C12—Metrics set: metric description #5

Item	Description
Name	Design Structure S2
Costs	1 labor day
Benefits	This metric system serves as a a reminder to use standards.
Impact	High impact if used early in the software project.
Target Value	The target value is 1.
Factors	Reliability, maintainability, and verifiability
Tools	No specific tools are required.
Application	This metric is simply used as a reminder that standards need to be used.
Data Items	N/A
Computation	Simple *yes* or *no* answer about whether standards have been applied.
Interpretation	Standards are certainly desirable, but any mathematical treatment of this metric would be difficult.
Considerations	This metric does not take into account the quality of the programming standard being used.
Training Required	Knowledge of what constitutes a standard.
Example	N/A
Validation History	None for this application
References	RADC-TR-85-37

Table C13—Metrics set: metric description #6

Item	Description
Name	Coding Simplicity S3
Costs	4–6 labor weeks
Benefits	The metric checks that an important aspect of proper software development is done correctly.
Impact	If the answer is 0, faulty software components will be corrected.
Target Value	The target value is 1.
Factors	Reliability, maintainability, and verifiability
Tools	No specific tools are required.
Application	Checks that control flow is from top to bottom in proper hierarchical decomposition.
Data Items	Representation of the control structure of appropriate software segments in a usable form
Computation	Metric requires a *yes* or *no* answer based on whether there is proper hierarchical control in all control structures.
Interpretation	If answer is *no*, there is an indication of poor software design.
Considerations	Appropriate as a flag to indicate a problem.
Training Required	Minimal. Need to be able to read appropriate description of control structure.
Example	N/A
Validation History	None for this application
References	RADC-TR-85-37

Table C14—Metrics set: metric description #7

Item	Description
Name	Cyclomatic Number S4
Costs	3 labor weeks plus cost of software
Benefits	Gives a measure of complexity of the software that may be used to flag low quality software, develop priorities for testing, and allocate resources to testing.
Impact	If the value of the metric is exceedingly high, software components will be further divided.
Target Value	Not to exceed 3.
Factors	Reliability and maintainability
Tools	Metric data collection and processing software
Application	Used to estimate complexity of the software.
Data Items	Number of edges (E) and number of nodes (N) in directed graph representation of software component (for example, HOL procedure)
Computation	$E - N + 2$
Interpretation	A high value on a software component indicates that the software component is too long or too complex.
Considerations	This metric is one of several complexity metrics and may or may not be the best choice. The correlation of this metric with the actual complexity of the software is a controversial issue.
Training Required	Minimal. Work can be computerized.
Example	$E = 12$, $N = 10$, metric value = $12 - 10 + 2 = 4$
Validation History	See C.1.5.
References	Annex D.2, [B118]

Table C15—Metrics set: metric description #8

Item	Description
Name	Size S5 (Number of statements in an (HOL) procedure)
Costs	1 labor week plus cost of software
Benefits	Gives a measure of size and complexity (for example, excessive size of the software) that may be used to flag low quality software and to estimate programmer effort.
Impact	If the value of the metric is exceedingly high, software components will be further divided.
Target Value	Not to exceed 13 for HOL procedures.
Factors	Reliability and maintainability
Tools	Metric data collection and processing software
Application	Used to estimate size and complexity of the software.
Data Items	Identification of syntactical elements that comprise the statements of the language (for example, IF, WHILE, assignment).
Computation	Count of statements in HOL procedure, excluding comments.
Interpretation	A high value on a software component indicates that the software component is too long or too complex.
Considerations	This metric is one of several size metrics and may or may not be the best choice. The correlation of this metric with actual size and complexity of the software is a controversial issue.
Training Required	Minimal. Work can be computerized.
Example	Count statements by computer
Validation History	See C.1.5.
References	There are no references.

C.1.2.2 Perform a cost-benefit analysis

To perform a cost-benefit analysis, identify the costs of implementing the metrics, identify the benefits of applying the metrics, then if necessary adjust the metrics set.

C.1.2.2.1 Identify the costs of implementing the metrics

Below is a list of various impacts and costs for a metric.

— *Metric Utilization Costs.* Detailed work breakdown structures will be used to allocate cost of collecting analyzing, interpreting, and reporting metric results.
— *Software Development Process Change Costs.* It is not anticipated at this time that there will be software development process change costs.
— *Organizational Structure Change Costs.* Organizational structure change costs will be distributed equally among the metrics for cost-analysis purposes.

— *Special Equipment.* Only new hardware and software specifically obtained for collecting data and implementing metrics for this project will be included in the cost. Cost for equipment used for implementing more than one metric will be prorated among the metrics.
— *Training.* If training for implementing metrics will be used for more than one project, cost will be distributed among the projects and the metrics within the projects.

C.1.2.2.2 Identify the benefits of applying the metrics

Identify and document the benefits that will result from the implementation of each metric in the metrics set, and specify the appropriate action to be taken based on metric results. Where possible, assign monetary values to these benefits. Identify costs other than metric costs necessary to realize these benefits.

C.1.2.2.3 Adjust the metrics set

Based on a cost-benefit analysis using information obtained above for each metric, add, alter, or delete metrics in the metrics set. Reallocation of costs for other metrics may be necessary if metrics are added or deleted.

C.1.2.3 Gain commitment to the metrics set

Benefits of the use of metrics will be presented to management in a series of seminars. When management is committed to the plan, similar seminars will be presented to the others involved in implementing the plan.

C.1.3 Implement the software quality metrics

To implement the software quality metrics, define the data collection procedures, prototype the measurement process, select tools, then collect the data and compute the metrics values.

C.1.3.1 Define the data collection procedures

Data collection will follow the software quality metrics methodology. Descriptions are provided for each data item. An example of a data item description is given below.

Table C16—Data collection procedures

Item	Description
Name	Cohesive value of module
Metrics	Modular Design M2
Definition	Cohesion is the measure of the strength of functional association of elements within a module (see annex D.2, [B120]). The following cohesive values are assigned to module types: Type — Value Functional — 1.0 Informational — 0.7 Communicational — 0.5 Procedural — 0.3 Classical — 0.1 Logical — 0.1 Coincidental — 0.0 For definition of module types see annex D.2, [B119].
Source	Detailed design documentation
Collector	Specifically trained personnel from Quality Assurance.
Timing	At the end of the preliminary design phase
Procedures	Values determined manually by examination of each module.
Storage	Information stored in a database file on a microcomputer disk.
Representation	Fixed-point notation with significant digits correct to nearest tenth.
Sample	Ten percent of available data selected randomly is to be collected.
Verification	Modules are to be examined and evaluated by two independent persons or teams.
Alternatives	None
Integrity	The supervisor for data collection may authorize altering data if an error is determined or if the modules being evaluated are altered.

C.1.3.2 Prototype the measurement process

A small similar project has been selected for testing data collection and metric implementation procedures. All procedures will be performed twice, using different personnel to check for uniformity.

Results of the prototype testing will be used to evaluate cost analysis for the metric set and to evaluate clarity of instructions and descriptions used for collection of data and implementation of metrics.

C.1.3.3 Select tools to apply to metrics

When possible, tools will be purchased off the shelf. Remaining tools will be custom made by outside sources.

C.1.3.4 Collect the data and compute the metrics values

A statistical analysis will be made to determine that the samples are appropriately selected. When more than one person is collecting the data, an analysis will be made to determine that the data collection is uniform.

C.1.4 Analyze the software metrics results

For each metric, results will be interpreted. The system will be evaluated with regard to the metric and the results will be reported. An example is given below for the metrics cyclomatic number, S4, and size, S5, evaluated during the preliminary design phase.

C.1.4.1 Interpret the results

Identify all Computer Software Components (CSCs) for which the value of S4 exceeds 3 or the value of S5 exceeds 13.

C.1.4.2 Identify software quality

CSCs with values that exceed target values are examined and, if possible, redesigned to reduce complexity or reduce size as appropriate. If this cannot be done, then the reasons should be documented. Exceptions to the target value requirement can only be given by the supervisor of the project.

C.1.4.3 Make software quality predictions

Although the ability to make mathematical predictions about software reliability failed the validity test, the ability to discriminate between quality levels (for example, procedures with errors vs. procedures with no errors) was validated. Therefore, it is predicted that errors in the software will occur if complexity and software component size exceed critical values of approximately 3 for cyclomatic number and 13 statements, per HOL procedure respectively (see C.1.5).

C.1.4.4 Ensure compliance with requirements

The direct metrics errors/statement and errors/procedure, listed in C.1.1.3, are used to ensure that the target values are achieved. The target values given in C.1.1.3 are 0.05 errors/statement and 0.60 errors/procedure. From the data presented in C.1.5, the number of errors is 64, the number of statements is 1600, and the number of procedures is 112, giving errors/statement and errors/procedure a value of 0.04 and 0.57, respectively. These numbers are less than the target values.

C.1.5 Validate the software quality metrics

The following example is provided to show how to make metric validation tests (see annex D2, [B114], [B115], and [B117]). No inferences should be drawn from this example regarding the validity of these metrics for other applications. These metrics are used for illustrative purposes only. The results of the validation tests could be different for other applications. The data used in the validation tests were collected from actual software projects. For the purpose of this example, these projects will be assumed to be similar to the surveillance and identification application. Characteristics of the data, such as sample size, format, and data categories, are illustrative and could be different for other applications.

C.1.5.1 Purpose of metrics validation

The purpose of the validation is to determine whether the complexity (cyclomatic number) and size (number of source statements) metrics, either singly or in combination, could be used to predict and assess the factor reliability, as represented by the direct metric error count, for the surveillance and identification application.

C.1.5.2 Validity criteria

Select values for *V, B, A,* and *P.* The values of *V, B, A,* and *P* (see 5.5.2) used in the example are 0.7, 0.7, 20%, and 80%, respectively. The criterion used for selecting these values is reasonableness (that is, judgment shall be exercised in selecting values in order to strike a balance between the one extreme of causing a metric that has a high degree of association with a factor to fail validation and the other extreme of allowing a metric of questionable validity to pass validation). It is not possible for validation to be completely mechanistic. Judgment shall be exercised in any statistical analysis.

Apply the validity criteria of 5.5.2 in the validation procedure that follows.

C.1.5.3 Validation procedure

Follow the steps below when performing a metrics validation.

C.1.5.3.1 Identify the quality factors sample

Draw a sample of procedures (that is, software component) from the metrics database of the quality factor reliability that is represented by the direct metric error count (errors). This sample shows procedures with no errors (table C17) and procedures with errors (table C18) for four software projects.

C.1.5.3.2 Identify the metrics sample

Using the same procedures as in C.1.5.3.1, identify the metrics samples for the cyclomatic number (complexity) and size (statements). The samples are listed in tables C17 and C18.

Table C17—Procedures with no errors

Complexity	Statements	Errors	Project
2	6	0	1
1	8	0	1
1	11	0	1
1	4	0	1
3	18	0	1
3	15	0	1
1	3	0	2
1	3	0	2
1	3	0	2
1	3	0	2
1	3	0	2
1	3	0	2
1	3	0	2
1	3	0	2
1	3	0	2
1	5	0	2

Table C17—Procedures with no errors (*Continued*)

Complexity	Statements	Errors	Project
1	5	0	2
1	5	0	2
1	13	0	2
1	3	0	2
1	3	0	2
1	3	0	2
1	3	0	2
1	3	0	2
1	3	0	2
1	3	0	2
1	2	0	4
1	2	0	4
1	7	0	4
1	5	0	4
1	7	0	4
1	5	0	4
1	5	0	4
1	5	0	4
1	5	0	4
1	4	0	4
1	3	0	4
1	3	0	4
1	3	0	4
1	3	0	4
1	3	0	4
1	3	0	4
1	3	0	4
1	3	0	4
1	5	0	4
1	5	0	4
1	6	0	4
1	9	0	4
1	6	0	4
1	8	0	4
1	9	0	4
1	9	0	4
1	9	0	4
2	4	0	4

Table C17—Procedures with no errors (*Continued*)

Complexity	Statements	Errors	Project
2	7	0	4
2	9	0	4
4	56	0	4
1	24	0	4
2	13	0	4
2	13	0	4
2	10	0	4
2	9	0	4
2	12	0	4
5	21	0	4
5	49	0	4
3	19	0	4
4	20	0	4
2	6	0	4
2	12	0	4
2	9	0	4
2	10	0	4
1	21	0	4
4	21	0	4
3	11	0	4
2	13	0	4
3	14	0	4
7	19	0	4
2	15	0	4
2	10	0	4
2	17	0	4
3	19	0	4
3	15	0	4
2	15	0	4

Table C18—Procedures with errors

Complexity	Statements	Errors	Project
2	14	1	1
6	26	5	1
5	7	2	1
5	21	1	1
2	6	1	1
1	3	1	2
1	11	1	2
1	8	1	2
2	15	3	2
8	45	3	2
4	18	1	2
6	54	3	2
2	34	2	2
4	19	1	2
5	30	2	2
4	26	1	2
6	94	8	2
2	13	1	13
6	83	1	4
5	28	1	4
8	37	5	4
3	13	2	4
3	16	1	4
7	34	1	4
5	24	1	4
4	18	3	4
5	35	2	4
13	49	5	4
4	19	1	4
4	27	1	4
4	17	2	4

C.1.5.3.3 Perform a statistical analysis

Perform the tests described in 5.5.2. Significance level and sample size are denoted by α and N, respectively. When it is necessary to specify a critical level of α in hypothesis tests, 0.05 is used. Statistical formulas are not given; these can be found in the references. Only results are shown.

C.1.5.3.4 Document the results

Below are detailed examples of the application of metrics validation described in 5.5.2.

C.1.5.3.4.1 Correlation

NOTE—See annex D.2, [B117].

Test 1: Linear Correlation

Compute the sample linear correlation coefficient (R) for errors (E) and complexity (C) and for errors (E) and statements (S) and compare each R^2 with $V = 0.7$.

Table C19—Sample correlations of procedures with errors

Item	Complexity	Statements
Errors with	$R = 0.7834$	$R = 0.5880$
Confidence Level	100.0% ($\alpha = 0.0000$)	99.95% ($\alpha = 0.0005$)
Number of Observations (N)	31	31

Table C20—Sample correlations of all procedures (errors and no errors)

Item	Complexity	Statements
Errors with	$R = 0.8010$	$R = 0.6596$
Confidence Level	100.0% ($\alpha = 0.0000$)	100.0% ($\alpha = 0.0000$)
Number of Observations (N)	112	112

Result: $R^2 < V = 0.7$ in all cases. Thus, neither the complexity nor the statements correlate with the errors to the required degree.

Test 2: Null Hypothesis

Perform a null hypothesis test H_0: $\rho = 0$ for E and C. This test is usually a part of Test 1.
Result: Reject H_0 with $\alpha = 0.0000$ and $N = 31$ (see table C19).

Test 3: Null Hypothesis

Perform a null hypothesis test H_0: $\rho > \sqrt{V} = 0.836$ for E and C, since $R^2 > V = 0.7$ is wanted.
Result: Accept H_0 with $\alpha = 0.01$ and $N = 31$ (see table C19).

Test 4: Partial Correlations

Compute the partial correlation coefficients for *E*, *C*, and *S*. These coefficients give the strength of the linear relationship between two variables while controlling for the effects of the remaining variables.

Table C21—Sample partial correlations of procedures with errors

Correlations with 31 observations (*N*)	*R* Values
Errors with Complexity	0.6430
Errors with Statements	–0.0816
Complexity with Statements	0.6557

Result: Looking at the low *R* values for *E* and *S*, it can be seen that the statements contribute essentially no additional information about errors, once complexity has been used as a predictor of errors.

Test 5: Confidence Interval

Compute a confidence interval of ρ for *E* and *C*.
Result: $0.593 < \rho < 0.891$ with $\alpha = 0.05$ and $N = 31$ (see table C19).

Conclusion. The results are mixed. Complexity failed Test 1, which was mandatory, but had favorable results for Tests 2, 3, and 5. Statements failed Test 1 and did not have as favorable results in the other tests as complexity. Neither the complexity nor the statements correlate with the errors.

C.1.5.3.4.2 Tracking

NOTE—See annex D.2, [B114] and [B115].

Test 1: Rank Correlation

Compute the Spearman coefficient of rank correlation (*r*) for *E* with *C* for Projects 1, 2, and 4 separately. (Project 3 is not used because it has only one error.) Note that the correlation is lower for *E* with *S* than for *E* with *C* and is therefore not shown. Compare *r* with $B = 0.7$ and α with 0.05. Procedures with errors are used rather than all procedures because the latter has too many ties in the sample. Rank correlation should not be used when there is a large number of ties. A moderate number of ties is tolerable.

Table C22—Spearman rank correlations of procedures with errors

	Complexity	Confidence Level	Remarks
Errors, Project 1, with ($N = 5$)	$r = 0.8250$	90.10% ($\alpha = 0.0990$) $\alpha > 0.05$	$r > 0.7$ Small sample size
Errors, Project 2, with ($N = 12$)	$r = 0.6723$	97.42% ($\alpha = 0.0258$) $\alpha < 0.05$	$r < 0.7$
Errors, Project 4, with ($N = 13$)	$r = 0.2522$	61.78% ($\alpha = 0.3284$) $\alpha > 0.05$	$r < 0.7$

Result: The desired result is $r > 0.7$ and $\alpha < 0.05$ (that is, indication of a nonzero correlation) for each project. Complexity does not track with changes in errors sufficiently for any of the projects. Therefore, neither changes in complexity nor changes in statements track with the observed changes in errors.

C.1.5.3.4.3 Consistency

NOTE—See annex D.2, [B114] and [B115].

Test 1: Rank Correlation

Compute r for E and C over all procedures with errors. Note that the correlation is lower for E and S than for E and C and is therefore not shown. Compare r with $B = 0.7$ and α with 0.05.

Table C23—Spearman rank correlations of procedures with errors

	Complexity	Confidence Level	Remarks
Errors with ($N = 31$)	$r = 0.5119$	99.48% ($\alpha = 0.0051$) $\alpha < 0.05$	$r < 0.7$

Result: The desired result is $r > 0.7$ and $\alpha < 0.05$. Complexity does not change consistently with changes in errors across all procedures with errors. Therefore neither complexity nor statements are consistent with errors.

C.1.5.3.4.4 Predictability

NOTE—See annex D.2, [B117].

Test 1: Scatter Plot

Make a scatter plot of E and C for procedures with errors to obtain a rough analysis of linearity.
Result: The dots in figure C1 show the relationship.

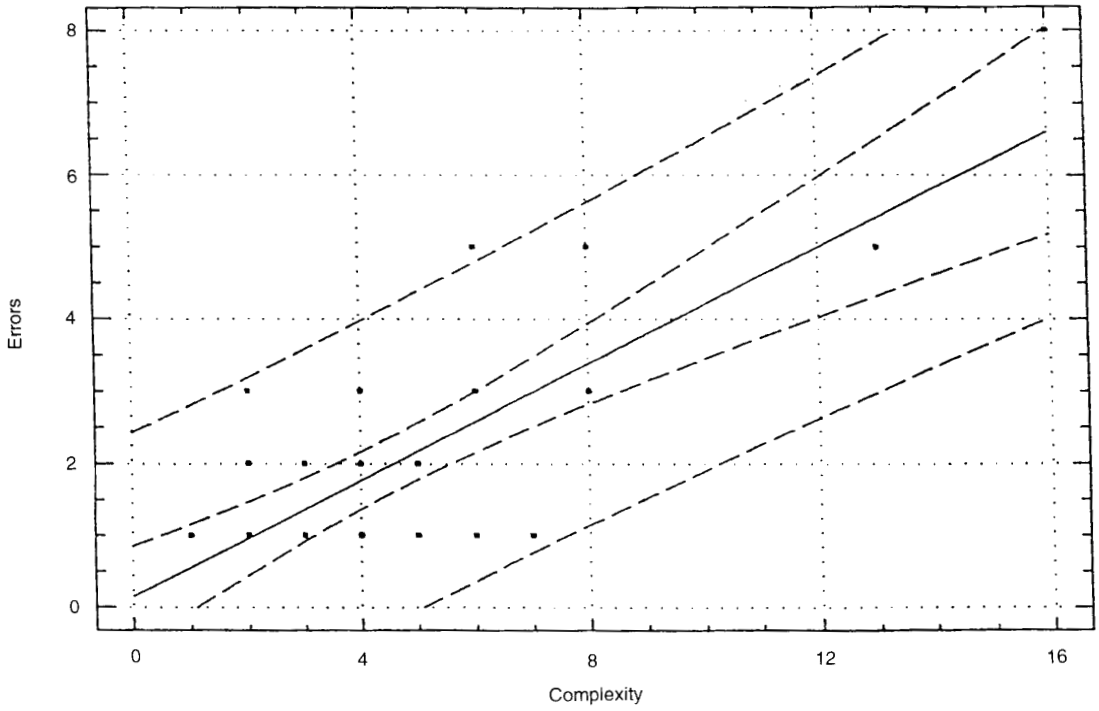

Figure C1—Regression of errors on complexity: procedures with errors

Test 2: Linear Regression

Perform a linear regression analysis of E on C for procedures with errors.

a) Test whether the assumptions of linear regression analysis hold true for these data. Two of the important assumptions are:

1) E is normally distributed for given values of C.

2) The variances of E are equal for given values of C.

Result: For cases of $C = 1$ and $C = 2$, where there was an adequate sample size, tests were conducted and it was found that neither assumption holds. In addition, E was not normally distributed when all 112 procedures were used in the analysis. The best fit for E is a negative binomial distribution.

b) Examine the residuals of E (the difference between observed and predicted as a function of C).

Result: Residuals increase with increasing C. This indicates that prediction error increases with increasing C. This is an undesirable result since a prediction error independent of C is wanted. The same results were obtained in a) and b) when all procedures were used.

Test 3: Regression Model

Plot the regression model in figure C1 for E on C for procedures with errors. The equation is:

$$E = 0.151 + 0.404C$$

The inner band is the 95% confidence interval for average E (that is, a 95% chance that, for a given C, the estimate of average E will fall within the band) and the outer band is the 95% prediction interval of E (that is, a 95% chance that, for a given C, the estimate of E will fall within the band). The fit is worse for regression of E on S (not shown).

Test 4: Observed vs. Predicted Errors

Compare observed errors with predicted errors (obtained from regression model) and note whether predictability $< A = 20\%$, for $P \geq 80\%$ of the predictions.
Result: Table C24 indicates that predictability < 20% for only 11 out of 31 cases, or only 35%. The result is 16% when all procedures are used (not shown). The test fails.

Table C24—Procedures with errors

Observed Errors	Predicted Errors	Prediction Error	Predictability (%)	Project
1	0.957831	0.04217	4.21686	1
5	2.572289	2.42771	48.55421	1
2	2.168674	–0.16868	8.43373	1
1	2.168674	–1.16867	116.86747	1
1	0.957831	0.04217	4.21686	1
1	0.554216	0.44578	44.57831	2
1	0.554216	0.44578	44.57831	2
1	0.554216	0.44578	44.57831	2
3	0.957831	2.04217	68.07228	2
3	3.379518	–0.37952	12.65060	2
1	1.765060	–0.76506	76.50602	2
3	2.572289	0.42771	14.25702	2
2	0.957831	1.04217	52.10843	2
1	1.765060	–0.76506	76.50602	2
2	2.168674	–0.16868	8.43373	2
1	1.765060	–0.76506	76.50602	2
8	6.608433	1.39157	17.39457	2
1	0.957831	0.04217	4.21686	3
1	2.572289	–1.57229	157.22891	4
1	2.168674	–1.16867	116.86747	4
5	3.379518	1.62048	32.40963	4

Table C24—Procedures with errors (*Continued*)

Observed Errors	Predicted Errors	Prediction Error	Predictability (%)	Project
2	1.361445	0.63855	31.92771	4
1	1.361445	–0.36145	36.14457	4
1	2.975903	–1.97590	197.59036	4
1	2.168674	–1.16867	116.86747	4
3	1.76506	1.23494	41.16465	4
2	2.168674	–0.16868	8.43373	4
5	5.397590	–0.39759	7.95180	4
1	1.765060	–0.76506	76.50602	4
1	1.765060	–0.76506	76.50602	4
2	1.765060	0.23494	11.74698	4

Test 5: Nonlinear Regression

Try nonlinear single independent variable regression models.
Result: Several nonlinear (for example, exponential) regressions of E on C for procedures with errors had lower correlation and worse fit (not shown) than the linear model.

Test 6: Multiple Linear Regression

Perform multiple linear regression analysis, using E as dependent variable and C and S as independent variables.

a) Test whether the assumptions of the multiple regression model hold. An important assumption of this method is that the independent variables are actually independent.

Result: The significant R between C and S of 0.833 for all procedures indicates dependence.

b) Examine the residuals of E for all procedures.

Result: Residuals increase with increasing C and S, indicating that prediction error would increase with increasing C and S, which is an undesirable result.

c) Plot the multiple regression model and compare with results of Test 3.

Result: The plots were made but are not shown because the fit is worse than in Test 3. For procedures with errors the regression equation is:

$$E = 0.174 + 0.437C - 0.00672S$$

Statements contribute little to the relationship. The comparison between simple and multiple regression is summarized in table C25, where F-Ratio is a measure of goodness of fit (generally, high value signifies good fit) and P is the percentage of predictions that are within the prediction error tolerance ($A = 20\%$).

Table C25—Regression comparison

	E vs. *C* Procedures with Errors	*E* vs. *C* All Procedures	*E* vs. *C* & *S* Procedures with Errors	*E* vs. *C* & *S* All Procedures
R	0.783	0.801	0.785	0.801
F-Ratio	46.1	196.9	22.5	97.6
P for *A*< 20%	35%	16%	35%	22%

Conclusion: Neither *C* nor *S* meets the predictability criterion, either singly or in combination, for predicting *E*. Multiple regression has no advantage over single independent variable regression for these data. Also, the assumptions of both models are not satisfied. Therefore, neither complexity nor statements are valid predictors of errors.

C.1.5.3.4.5 Discriminative power

NOTE—See annex D.2, [B114] and [B115].

Test 1: Mann-Whitney Test

Divide the data into two sets: procedures with errors and procedures with no errors. Rank these sets according to their *C* and *S* values (statistical programs will do the ranking automatically) and perform the Mann-Whitney test to see whether *C* and *S* can discriminate between the two sets of procedures.

Result: The results of the Mann-Whitney test for *C* and *S* are shown in table C26. The average ranks of *C* and *S* for procedures with errors are much higher than the average ranks for procedures with no errors, respectively. It can be concluded from the low probabilities of higher statistics that *C* and *S* for procedures with errors have significantly higher medians.

This passes the tests. Caution—A large number of ties weakens this test. There are a large number of ties in *C* but not in *S*.

Table C26—Mann-Whitney test of two samples

Sample	Average Rank	Number of Values	Large Sample Test Statistic	Two-tailed probability ≥ Z	Observations
Complexity procedures with errors	85.9032	31	$Z = -6.30181$	$2.95465E-10$	112
Complexity procedures with no errors	45.2469	81			
Statements with errors	85.2419	31	$Z = -5.82408$	$5.76106E-9$	112
Statements with no errors	45.5	81			

Test 2a: Chi-Square Test (Complexity)

Divide the data into four categories, as shown in table C27 according to a critical value of complexity, C_c, so that a Chi-square test can be performed to determine whether C_c can discriminate between procedures with errors and those with no errors. C_c is chosen to provide at least five observations for each cell in the table in order to ensure the validity of the test. This may involve trial and error.

Table C27—Complexity contingency table

Procedures with	Complexity ≤ 3	Complexity > 3	Row Total
No Errors	75	6	81
Errors	10	21	31
All	85	27	112

Result: The result of the Chi-square test is shown in table C28. The high value of the Chi-square and the very small significance level indicate that C_c can discriminate between procedures with errors and those without errors. This passes the test.

Table C28—Summary statistics for contingency tables ($C_c = 3$)

Chi-square	D. F.	Significance
44.6081	1	$2.40692E-11$

Test 2b: Chi-Square Test (Statements)

Do the Chi-square test for statements. The information is shown in table C29.

Table C29—Statement contingency table

Procedures with	Statement ≤ 13	Statement > 13	Row Total
No Errors	64	17	81
Errors	7	24	31
All	71	41	112

Result: The same comments made in Test 2a apply to S_c. This passes the test.

Table C30—Summary statistics for contingency tables ($S_c = 13$)

Chi-square	D. F.	Significance
30.7658	1	$2.91118E-81$

Conclusion: Complexity and statements are valid with respect to the discriminative power criterion and both can be used to distinguish between acceptable ($C \leq 3$, $S \leq 13$) and unacceptable ($C > 3$, $S > 13$) quality for the surveillance and identification application during life cycle phases when this data can be collected. It should be noted that it is less expensive to collect statement counts than to compute complexity.

C.1.5.4 Additional requirements

In addition to the above stated requirements, be aware of the following.

C.1.5.4.1 The need for revalidation

Repeat all validation tests for C and S on future projects, keeping track of P, the success rate (that is, the percentage of applications for which a metric shall pass validity tests in order to be certified as valid).

C.1.5.4.2 Confidence in analysis results

Confidence in analysis results will be achieved if C and S, when used on this and future projects, are able to flag poor quality software during development so that it can be corrected before being delivered to the customer.

C.1.5.4.3 Stability of environment

To the extent practical, perform future validity tests for applications and environments that are similar to this one so that validation results can be compared and aggregated across projects.

C.2 Example from a commercial environment

This subclause describes how a company in the commercial sector might apply the steps outlined in clause 5. It presents a portion of the company's quality framework and discusses some of the methods used to implement the methodology.

There are two primary differences between this example and the mission critical example discussed in C.1 of this annex:

a) Emphasis in the mission critical example is on development process quality requirements, while the emphasis in this example is on end-user quality requirements.
b) All software in this example is sold off the shelf. A result of this difference is the fact that the choice of factors and subfactors used by this company directly conflict with the factor and subfactor definitions used by the mission critical example. This contradiction is to be expected from the differing environments, and in no way indicates that one application of the standard is better than the other.

C.2.1 Establish software quality requirements

Establish software quality requirements by identifying possible quality requirements, ordering the quality requirements, and then quantifying each factor.

C.2.1.1 Identify a list of possible quality requirements

While the quality factors listed in military handbooks such as RADC-TR-85-37 ("Boeing Report") are of interest as possible examples, companies pursuing an off-the-shelf software marketing strategy may need to add functionality to the list of quality requirements. (For example, *Does our product provide all of the functionality that the target market wants? needs? Does it beat the competition's functionality?*) In this example, the requirements have been expressed in terms of: functionality, usability, reliability, performance, and supportability (FURPS). Table C4 provides examples of how environmental characteristics may affect the software quality requirements identification process.

C.2.1.2 Order the quality requirements

Techniques that facilitate this step include a quality requirement survey and performing an analysis on the interrelationships among quality factors and life-cycle costs.

C.2.1.2.1 Survey all involved parties

A business team composed of, but not limited to, marketing, R&D, QA, technical support, and publications is formed to determine the relative importance of the quality requirements. Other organizational entities that may be involved could be sales, legal, internal standards, or metrics administration. There may be significant differences between the involved survey parties about the relative importance of different quality requirements. For example, developers may desire more importance for some factors than end users, and vice versa.

C.2.1.2.2 Determine the actual list of quality requirements

Discussions, and if necessary, additional analysis, are held until a consensus is reached on a single, ordered list of requirements. It is very important that an agreement is reached for the final ordering of the list, because it may directly affect cost as well as other characteristics. In this example, the company creates real time software so that performance and reliability are very important and require an excellent quality level. Functionality and supportability are important and require a good quality level, and usability is least important. At this point, wherever conflicts arise between the various quality factors, the developers have clear instructions on how the conflicts are to be resolved.

C.2.1.3 Quantify each factor

Now that each quality factor has been identified and ordered relative to importance, cost, and schedule constraints, the involved organizations need to assign a set of direct metrics to represent each quality factor. More importantly, quantitative values shall be assigned to each direct metric as a way of determining if the delivered system meets its quality requirements.

C.2.2 Identify software quality metrics

Remember to apply the software quality metrics framework and perform a cost-benefit analysis when identifying software quality metrics. Then gain commitment to the metrics set from management.

C.2.2.1 Apply the software quality metrics framework

Once the quality factors relevant to the project are identified, it is time to fill in the rest of the metrics framework (the subfactors) referred to in figure 1 in clause 4.

There are several ways to choose subfactors. One way would be to apply a pre-existing hierarchy such as a corporate metrics standard or the RADC Report (RADC-TR-85-37). In this case, simply refer to the pages documenting the desired factors and gather a list of the related subfactors. The list could then be prioritized based on the rank order of the factors.

A second way would be to form a committee to select technical attributes that are related to the factors. This is more time consuming than the prior method, but has the advantage of being optimized for the local environment. The actual selections made are archived so that subsequent projects can benefit from the committee's work. (Corporate metrics standards often start this way).

Finally, it may be decided that one or more of the quality factors chosen have technical significance. In this example, the company could define usability as one of its own subfactors. Examples of subfactors associated with each quality factor are shown in annex A. Once a set of subfactors has been selected, the list is circulated and agreed upon. In this example, subfactors were chosen from table C31. (For example, speed, efficiency, resource needs, throughput, and response time are the high-priority subfactors associated with the high-priority factor performance.)

Table C31—Factors and subfactors

Factors	Subfactors
Functionality	Feature Set
	Applicability
	Capability
	Generality
	Compatibility
	Security
Usability	Usability
Reliability	Failure Rate
	Recoverability
	Predictability
	Accuracy
	MTTF
Performance	Speed
	Efficiency
	Resource Needs
	Throughput
	Response Time

Table C31—Factors and subfactors (*Continued*)

Factors	Subfactors
Supportability	Testability
	Extensibility
	Adaptability
	Maintainability
	Configurability
	Serviceability
	Installability
	Integratability
	Localizability

Having chosen subfactors, actual metrics shall be chosen using a similar process. If validated metrics are already available, they are used; if not, those metrics that are reasonable and are accepted by all of those involved are applied. Annex D.1 lists several references that contain metric suggestions. It may be useful to include nonvalidated metrics at this point in order to validate them in the future; however, these metrics shall not be used as predictors. In this example, the company has decided that its operating systems' speed (a high priority characteristic related to the performance factor, see table C31) may be considered a function of millions of user instructions executed per second (MIPS) and the percentage of CPU time spent in the operating system (system overhead). It has also been decided that efficiency is characterized by both system overhead and memory utilization. A partial hierarchy based on these decisions is shown in figure C2.

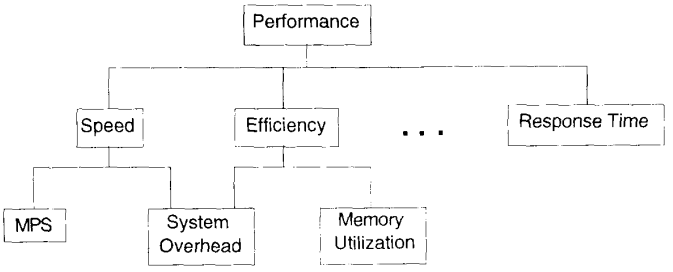

Figure C2—Partial hierarchy from the metrics framework

Table C32—Metrics set: metric description #1

Item	Description
Name	System Overhead (Number of cycles divided by MIPS [during test time])
Costs	a) Logic analyzers shall be purchased and attached to the test system at $5K per system. User training: 1 labor week b) Parts of the operating system shall be instrumented:2 labor weeks c) Technician shall be assigned to run equipment: 5 labor weeks
Benefits	Useful for characterizing cost of software to the customers. Will flag an alert when the operating system is no longer true real time.
Impact	If system overhead exceeds target value plus 2%, the project shall undergo management review.
Target Value	The goal is to keep system overhead to less than 5% of the CPU time.
Factors	Performance
Tools	Logic analyzer, database, and plotting software
Application	This metric is required during system test to verify that the operating system meets its performance requirements. It is also required at the end of the development phase to verify that the system is ready for third-party testing.
Data Items	Fraction of CPU time
Computation	The metric value is the fraction of the total CPU cycles spent executing operating system instructions (privileged bit enabled). The measurement is to be taken while the machine is executing the standard regression test suite (see the project test plan for further details).
Interpretation	The metric value is the reciprocal of the available CPU power.
Considerations	The metric value is only valid when the system is executing a representative sample of application code.
Training Required	Logic analyzer training
Example	Project *X* has a goal that the OS only requires 5% of the CPU cycles. At final verification, the test technician connects the programmable logic analyzer to the CPU and sets it to count the number of cycles in which the privileged bit is set. Then, the full test package is run, the overhead is calculated as: (# of cycles)/(MIPS)(test time).
Validation History	None for this application
References	Refer to project test plan and programmable logic analyzer for instructions.

Table C33—Metrics set: metric description #2

Item	Description
Name	Defect Rate (Defects reported divided by number of labor weeks [See Computation].)
Costs	a) Defect database shall be maintained: 4 labor hours per week. b) All defects discovered shall be logged: 1 hour per tester per week. c) Periodic reports shall be generated: 2 hours per week. d) Labor for developers and testers to attend the defect database class.
Benefits	Useful for predicting post-release Mean Time To Failure (MTTF). Also useful for tracking progress of the test phase.
Impact	If the defect rate differs from an historical average for this type of project by more than one standard deviation, then review the testing procedures.
Target Value	There is no target value for the defect finding rate per se. The goal is to find all of the defects prior to system release.
Factors	Reliability
Tools	Defect database and report generation software
Application	This metric is required during all defect finding activities including design and code review, functional and system test, and post system release for validation purposes.
Data Items	Defect descriptions including date found, date fixed, type of defect (real error or tester misunderstanding), and severity
Computation	The defect rate is calculated by dividing the number of defects reported by the number of labor weeks. The defect rate may be calculated for all reported defects, all defects that have to be fixed, and all defects of high severity.
Interpretation	The defect rate is to be plotted on a chart vs. the average defect rate for this type of project. If the rate is far higher than average, the software may be exceptionally poor. Then extra investigation of the software is required. If the rate is far lower than average, the test procedures may not be stringent enough. Then a review of the test is required. The MTTF is to be predicted using the nonhomogeneous Poisson model.
Considerations	The metric is only to be used as a sanity check and potential predictor. Do not base software release decision on it.
Training Required	All developers and testers shall take the defect database class (4 hours).
Example	For the 5 weeks of testing, the number of defects reported against a software package is: 20, 20, 14, 35, and 44. The rate is out of the bounds allowable for that type of package—it is reaching a peak at the wrong time (too soon). The software under test is sent back for additional code inspections, and the data is not used for prediction of MTTF—it will not converge.
Validation History	None for this application
References	Annex D.2, [B122]

C.2.2.2 Perform a cost-benefit analysis

When performing a cost-benefit analysis, identify the costs of implementing the metrics and the benefits of applying the metrics. Then, if necessary, adjust the metrics set.

C.2.2.2.1 Identify the costs of implementing the metrics

At this time, each proposed metric shall be examined to consider how much it would cost to implement. For example, *What is the overhead involved in tracking every defect? What is the (anticipated) gain in knowing this information?* For each metric that it uses the example company considers:

— *What tools exist to use this metric, and, if necessary, how much would it cost to acquire them?*
— *How much staffing and training is required, by each organization, in order to effectively use each tool?* (Use of a tool may also require a change in the overall development process.)
— *How much time and associated cost will it take the development team to collect, analyze, interpret, and report each metric?*
— *How much of the data-collection process can be automated?* (The cost of the hardware and software used for this purpose shall be included.)

C.2.2.2.2 Identify the benefits of applying the metrics

For each proposed metric consider:

— *Who* (developer, management, marketing) *benefits from the information?*
— *How much will the final software product benefit, in quality terms, by having this metric data?*
— *How much, if anything, will be saved over the software life cycle by using this information?* (For example, *Will having defect information early shorten regression testing time?*)

C.2.2.2.3 Adjust the metrics set

Based on the cost-benefit analysis performed above, decisions shall be made to either use, alter, or delete each proposed metric. It would be helpful, for future projects, to note the rationale for altering or deleting a metric from the metrics set.

C.2.2.3 Gain commitment to the metrics set

The final metrics set, including the cost-benefit analysis, shall be presented to management, understood, and accepted before implementation. This may be done by a combination of circulating review copies of the metrics set, having inspections, or holding meetings.

C.2.3 Implement the software quality metrics

To implement the software quality metrics, define the data collection procedures, prototype the measurement process, then collect the data and compute the metrics values.

C.2.3.1 Define the data collection procedures

At this point, explicit information shall be recorded regarding who will collect what metrics data and when. These procedures should be recorded according to the format stated in table 3 in clause 5.

An illustration of the defect description data item used by the defect rate metric (see C.2.2.1) follows.

Table C34—Description of a data item

Item	Description
Name	Defect Description (test phase)
Metrics	Defect Rate, Defect Locality, and Unresolved Defects
Definition	The Defect Description is a set of data stored at the time a tester discovers an anomaly in the software. The description shall contain, but is not limited to, the following data items: a) Tester name b) Suspect software c) Date found d) General description e) Defect type f) Resolution (includes tester error) g) Date resolved h) Modules fixed (if appropriate) i) Related defects
Source	The primary source is the test procedure(s). However, any person associated with the project that finds a defect, regardless of how, shall report it.
Collector	See Source.
Timing	This type of defect description is valid after the software has passed inspections through its release.
Procedure	The descriptions are to be entered manually, using the relational database access programs.
Storage	Defect database
Representation	ASCII text with keyed fields
Sample	All defects during testing shall be recorded and entered into the database.
Verification	All defect resolutions shall have the approval of the originator or the project management team.
Alternatives	Software problem report form
Integrity	Any member of the project may submit a defect description at any time. However, only the database administrator may remove a description defect from the system.

C.2.3.2 Prototype the measurement process

The company has decided to test the metrics program on a small project. In this case, they have decided to apply as many of the metrics as possible to already-released software, and to compare them to the direct measures they received from the field. (This is particularly useful with predictive measures on defect rates.) The company has also decided to apply the metrics to a minor upgrade of the product. This has the advantages of being a small enough project to manage, and of providing a baseline for cost-benefit analysis. (That is, how much longer or shorter did it take to release this version of the product.)

C.2.3.3 Select tools to apply the metrics

In order to save time and money, the company has decided to use a third-party defect database and defect tracking system from a previous project, and will develop its own data-entry procedures.

C.2.3.4 Collect the data and compute the metrics values

As the project life cycle progresses, the responsible parties are collecting and storing defect-rate data into the newly acquired database. An organization shall be elected to monitor and analyze the data collection process and metric computations for accuracy, uniformity, and validity.

C.2.4 Analyze the software metrics results

The organization responsible for archiving, analyzing, and reporting the software quality metrics for the example company is quality assurance. QA's job includes making occasional sanity checks on the metrics database, seeing to it that the data is accumulated (whether automatically or manually), interpreting the documented metrics set, and reporting it (ideally, in tabular or graphic form) to the appropriate project personnel. On the prototype project, close attention is paid to how much QA time is taken by this activity, and how much use is made of the metrics. This information is used for future cost-benefit analysis.

C.2.4.1 Interpret the results

In accordance with Metric Description #2, the defect rate is calculated three ways:

a) For all reported defects
b) For those defects that result in fixes
c) For those defects of high severity

Meetings are held with the proper project personnel to determine the cause of unusually high or low defect rates.

C.2.4.2 Identify software quality

The accuracy and timeliness of the nonpredictive metrics are evaluated as the project progresses.

Before the software becomes operational, all predictive values not satisfying the agreed upon quality-requirement direct-metric target values should be subjected to detailed analysis. It is important to note that if a metric falls outside its target value it does not always result in unacceptable software quality during operation.

Based on the example metrics set, the following should be considered: *Are all defects being reported? Are the system overhead figures reproducible?* and *What is the cost of gathering, storing, and correlating the data?* After delivery of the system, the predictive metrics are re-examined.

C.2.4.3 Make software quality predictions

After project delivery, a metrics report is published complete with an updated metrics analysis where actual cost data replaces the original estimates and predictive figures are published, using the formulas in the metrics set. This information becomes part of the project history, which is archived for future reference.

For the system overhead metric, data has been gathered regarding what percentage of CPU cycles is used by the Operating System (OS). Since the overhead changes are based on what the OS is doing, the data was

gathered during the execution of a set of standard benchmark programs. The metrics set indicates how to normalize this data in order to predict post-release efficiency.

C.2.4.4 Ensure compliance with requirements

All quality requirements, factors, metrics, and associated direct-metric target values are compared to actual metric results. Each measurement that does not meet the specified target value(s) are noncompliant. It is up to project management, QA, and marketing to either:

a) Have the quality requirement met before shipment
b) Waive the quality requirement
c) Schedule the rework of the design in the future to improve the quality of the product

C.2.4.5 Validate the software quality metrics

The methodology used to validate the metrics described in this commercial example was similar to the methodology described in the mission critical example in C.1 of this annex. The methodology is not presented here both to save space and to prevent unnecessary duplication of information.

Annex D
Annotated bibliography, bibliography, and standards and related documents

(informative)

D.1 Annotated bibliography

[B1] Adamov, R. and Lutz, R., "A proposal for measuring the structural complexity of programs," *The Journal of Systems Software,* vol. 12, pp. 55–70, 1990.

This article presents the basic ideas behind a proposed structural complexity metric.

[B2] Bailey, C. T. and Dingee, W. L., "A software study using Halstead metrics," *Performance Evaluation Review,* vol. 10, pp. 189–197, Mar. 1981.

This paper examines two of Halstead's premises: software errors can be correlated with effort or volume, and language level is a constant over all algorithms in a given language. The authors' results did not agree with these premises.

[B3] Baker, A., Bieman, J., Fenton, N., et al, "A philosophy for software measurement," *The Journal of Systems and Software,* vol. 12, no. 3, pp. 277–281, July 1990.

This paper presents a discussion on validation of software measurement and discusses uses for structural measures.

[B4] Baker, A. L. and Zweben, S. H., "A comparison of measures of control flow complexity," *IEEE Transactions on Software Engineering,* vol. SE- 6, no. 6, pp. 506–512, Nov. 1980.

The authors analytically examine the strengths and weaknesses of three measures, namely software efforts (Halstead), program knots (Woodard, Henner, and Hedley), and cyclomatic complexity (McCabe), with respect to three control flow issues. They come to the conclusion that McCabe's cyclomatic complexity measure is the best measure of program control flow complexity as it is based on firm analytical ground and, unlike the knots measure, it does not suffer from the disadvantage of language dependency. The authors also conclude that it would be worthwhile to develop a metric that is a combination of software effort and cyclomatic complexity since their respective strengths and weaknesses complement each other.

[B5] Basili, V. R., *Tutorial on Models and Metrics for Software Management and Engineering.* Los Alamitos, CA: IEEE Computer Society Press, 1980.

This tutorial collects several papers on different topics of software metrics. Some of the papers listed in this bibliography are included in this tutorial.

[B6] Basili, V. R. and Phillips, T., "Evaluating and comparing the software metrics in the software engineering laboratory," *ACM Sigmetrics, 1981 ACM Workshop and Symposium on Measurement and Evaluation of Software Quality,* vol. 10, pp. 95–106, Mar. 1981.

This paper presents results of a statistical regression carried out on data to find a relationship between Halstead's effort metric and various stages of the software development process. It also examines whether other measures, such as lines of source code and executable statements, provide a better relationship with the effort metric and whether factors such as size, complexity, and testing level affect the correlations.

[B7] Basili, V. R., Selby, R. W. and Phillips, T.Y., "Metric analysis and data validation across FORTRAN projects," *IEEE Transactions on Software Engineering,* vol. SE-9, no. 6, pp. 652–663, Nov. 1983.

As a step toward validating some of the available quality metrics, the authors have analyzed the software science metrics, cyclomatic complexity, and various standard program measures and their relation to effort, to development errors, and to one another. The data investigated are collected from a production

FORTRAN environment and examined across several projects at once, within individual projects, and by individual programmers across projects.

[B8] Bastani, F. G., "An approach to measuring program complexity," *Proceedings of COMPSAC 83,* Chicago, IL, pp. 90–94, 1983.
The author proposes a measure of program complexity based on the program dependency graph and on the simulation of the program understanding process. The model uses two parameters, induction ability and a good memory ability, to determine complexity.

[B9] Beizer, B., *Software Testing Techniques. (Second ed.)* New York: Van Nostrand Reinhold, 1990.
This book has a chapter devoted to metrics that includes a summary of the strengths, weaknesses, and criticism of McCabe's metrics; Halstead's metrics; and various structural, linguistic, and hybrid metrics in addition to extensive bibliographical references to the metrics literature.

[B10] Berns, G. M., "Assessing software maintainability," *Communications of the ACM,* vol. 27, no. 1, pp. 15–23 Jan. 1984.
This paper suggests that the ease of maintaining a program depends on how difficult the program is to understand. The paper presents a technique to measure program difficulty by quantifying carefully selected weights and factors. This technique is implemented in the Maintainability Analysis Tool (MAT), a superset of FORTRAN 77, for the VAX computer.

[B11] Berry, R. E. and Meekings, B. A. E., "A style analysis of C programs," *Communications of the ACM,* vol. 28, no. 1, pp. 80–88, Jan. 1985.
This paper presents a metric for measuring the style of C programs.

[B12] Bowen, T., Wigle, G., and Tsai, J., "Specification of software quality attributes," RADC-TR-85-37, 3 volumes, Feb. 1985.
This document is a three-volume set that was produced to provide methods, procedures, techniques, and guidance to Rome Air Development Center Air Force software acquisition managers who specify the requirements for software quality.

[B13] Card, D. and Glass, R. L., *Measuring Software Design Quality.* Englewood Cliffs, NJ: Prentice Hall, 1990.
This book proposes a set of metrics relating to software design complexity, describes a software complexity model, and validates this model using data from actual projects.

[B14] Cheung, R. C., "A user-oriented software reliability model," *IEEE Transactions on Software Engineering,* vol. SE-6, no. 2, pp. 118–125, Mar. 1980.
This paper gives some calculations of program reliability based on how reliable its modules are. The effect of module change on the whole program and ways of estimating probability measures of module reliability are presented.

[B15] Christensen, K., Fitsos, G.P., and Smith, C.P., "A perspective on software science," *IBM Systems Journal,* vol. 20, no. 4, pp. 372–387, Apr. 1981.
This paper provides an overview of an approach to the measurement of software based on the count of operators and operands contained in a program, and is consistent across programming language barriers. This paper is critical of software science as a practical engineering tool, but praises it as a step.

[B16] Conte, S., Dunsmore, H., and Shen, V., *Software Engineering Metrics and Models.* Redwood City, CA: Benjamin Cummings Publishing Co., 1986.
This textbook describes software metrics.

[B17] Cook, M. L., "Software metrics: An introduction and annotated bibliography," *Software Engineering Notes,* vol. 7, no. 2, pp. 41–60, Apr. 1982.

The author introduces and presents an annotated bibliography on software metrics as well as program complexity metrics. He discusses resource measurement and system measurement. The author also gives overviews of software engineering and software metrics in general.

[B18] Cote, V., Bourque, P., Oligny, S., and Rivard, N., "Software metrics: an overview of recent results," *The Journal of Systems and Software,* vol. 8, 1988.

This paper classifies and compares 124 articles on software metrics.

[B19] Coulter, N. S., Cooper, R. B., and Solomon, M. K., "Information-theoretic complexity of program specifications," *The Computer Journal,* vol. 30, no. 3, pp. 223–227, June 1987.

This paper proposes an entropy metric for measuring the complexity of a program specification.

[B20] Crawford, S. G., McIntosh, A. A., and Pregibon, D., "An analysis of metrics and faults in C software," *The Journal of Systems and Software,* vol. 5, no. 1, pp. 37–48, Feb. 1985.

This paper evaluates the extent to which a set of software measures (including McCabe's and Halstead's) correlate with the number of faults and total estimated repair effort for a large software system.

[B21] Currans, N., "A comparison of counting methods for software science and cyclomatic complexity," *Pacific Northwest Software Quality Conference,* 1986.

Counting methods for Halstead's metrics, McCabe's metric, lines of code, and bugs are correlated and compared. This article is based on a study of 30 C modules (no module size is given) and presents a correlation between alternative ways of counting McCabe's complexity.

[B22] Curtis, B., "Measurement and experimentation in software engineering," *Proceedings of the IEEE,* vol. 68, no. 9, pp. 1144–1157, Sept. 1980.

The contributions of measurement and experimentation to the state of the art in software engineering are reviewed. The role of measurement in developing theoretical models is discussed, and concerns for reliability and validity are stressed.

[B23] Davcev, D., "Some new observations about software science indicators for estimating software quality," *Empirical Foundations of Information and Software Science, Proceedings of the First Symposium,* Atlanta, GA, pp. 245–248, Nov. 1982.

In this paper the author points out the drawbacks of Halstead's and Mccabe's metrics in estimating implementation time, and proposes an estimator for computing the total number of mental comparisons for implementing a program in a given language and estimating implementation time, taking into account the level of nesting in a program. Experiments on FORTRAN programs showed a high degree of agreement between observed times and predicted times.

[B24] Davis, J. S., "Chunks: A basis for complexity measurement," *Empirical Foundations of Information and Software Science, Proceedings of the First Symposium,* Atlanta, GA, pp. 119–128, Nov. 1982.

The author contends that chunks play an important role in cognitive process and are thus a more convincing basis for complexity measurement than existing direct measures, such as operators and operands. Chunks are loosely defined as statements that are related functionally and conceptually. Any complexity measure shall account for the complexity of the chunks themselves and their interconnections, to determine overall program complexity.

[B25] DeMarco, T., *Controlling Software Project, Management, Measurements, and Estimation.* New York: Yourdon Press, 1982.

This book provides various methods to estimate the costs of software development and to determine the validity of the estimation as the software is developed. The metrics suggested in this book are based on the methods provided in the author's earlier book, Structured Analysis and System Specifications.

[B26] Deutsch, M. S. and Willis, R. R., *Software Quality Engineering.* Englewood Cliffs, NJ: Prentice Hall, 1988.

This book contains a section that describes an approach to specifying software quality, achieving quality, and mapping the quality framework into a set of activities.

[B27] Dunsmore, H. E., "Software metrics: an overview of an evolving methodology," *Empirical Foundations of Information and Software Science, Proceedings of the First Symposium,* Atlanta, GA, pp. 183–192, Nov. 1982.

This paper describes (with examples) lines of code, cyclomatic complexity, average nesting depth, and the software science length, effort, and time metrics. Software metrics that are to be used for time, cost, or reliability estimates should be validated statistically via data analyses that take into consideration the application, size, implementation language, and programming techniques employed. The critical need is for software metrics that can be calculated early in the software development cycle to estimate the time and cost involved in software construction.

[B28] Ejiogu, L. O., "The critical issues of software metrics," *ACM SIGPLAN Notices,* pp. 59–63, Mar. 1987.

This paper defines software metrics as the science of measurements of software productivity for the purposes of comparison, cost estimation, and forecasting. This paper also discusses four techniques of measurement: enumeration, estimation, dimensionalization, and parameterization and when it is appropriate to use each technique.

[B29] Elshoff, J. L., "Characteristic program complexity measures," *Seventh International Conference on Software Engineering,* Orlando, FL, March 26–29, 1984.

Twenty metrics (including Halstead's software science) are applied to 585 PL/I procedures, and the results are correlated.

[B30] Evangelist, M., "Program complexity and programming style," *Proceedings of the International Conference on Data Engineering (COMPDEC), IEEE Computer Society,* pp. 534–541, Apr. 24–27, 1984.

This paper shows that popular software complexity metrics are not consistently sensitive to the application of programming style rules.

[B31] Evans, M. and Marciniak, J., *Software Quality Assurance & Management.* New York: John Wiley & Sons, 1987.

This book discusses the importance of measurement to software quality and outlines a software quality framework.

[B32] Evans, M. and Marciniak, J. J., "Software quality metrics," *Software Quality Assurance and Management,* pp. 157–186. New York: John Wiley & Sons, 1987.

This chapter states that in order to achieve a true indication of quality software it must be subjected to measurement. That is, the attributes of quality must be related to specific product requirements and quantified.

[B33] Fagan, M., "Advances in software inspections," *IEEE Transactions on Software Engineering,* vol. SE-12, no. 7, pp. 774–751, July 1986.

This paper contains a description of the Fagan inspection technique.

[B34] Fitsos, G. P., "Vocabulary effects in software science," *Seventh International Conference on Software Engineering,* Orlando, FL, March 26–29, 1984.

This paper states that the number of unique operators for a set of programs tend to be a constant for PL/S and possibly for PL/I (and postulates that this may hold true for other high level languages). It proposes a methodology to provide a quantitative measure of difficulty against which an individual programmer can measure himself or herself.

[B35] Gaffney, J. E., "Estimating the number of faults in code," *IEEE Transactions on Software Engineering*, vol. SE-10, no. 4, pp. 459–465, July 1984.

This paper provides formulas relating the number of faults to the number of lines of code and the number of faults to the number of conditional jumps. It determines that there are, on average, about 21 bugs per KSLOC discoverable after successful compilation. It shows that the number of bugs appear not to be a function of the level of the coding language employed, and knowledge of items used, such as the size of the vocabulary (operator and operand), appears to be of little consequence to the estimate of bug content beyond that based on SLOC count.

[B36] Gong, H. S. and Schmidt, M., "A complexity metric based on selection and nesting," *ACM Sigmetrics—Performance Evaluation Review* 13-1, pp. 14–19, 1985.

The authors present a structural metric that augments cyclomatic complexity to include nesting depth.

[B37] Grady, R. and Caswell, D., *Software Metrics: Establishing a Company-wide Program.* Englewood Cliffs, NJ: Prentice Hall, 1987.

This book contains a discussion of how Hewlett-Packard recognized a need for a measurable, controllable software process. It describes the method used to meet that need by putting a software metrics program into place. Lessons and examples can be transferred to other companies.

[B38] Hall, W. E. and Zweben, S. H., "The cloze procedure and software comprehensibility measurement," *IEEE Transactions on Software Engineering*, vol. SE-12, no. 5, pp. 608–623, May 1986.

Cloze tests seem to offer software engineers several theoretical and practical advantages over multiple-choice comprehension quizzes, the most common software comprehension measurement tool. A model of software cloze tests was developed to identify a software characteristic that may produce invalid results. The developed model was shown to be consistent with software cloze test results of another researcher and led to suggestions for improving software cloze testing.

[B39] Hamer, P. G. and Frewin, G. D., "M. H. Halstead's software science—A critical examination," *Proceedings of the Sixth International Conference on Software Engineering*, pp. 197–206, Tokyo, Japan, 1982.

Halstead's software science metric is examined and critiqued.

[B40] Harrison, W. A. and Magel, K., "A topological analysis of the complexity of computer programs with less than three binary branches," *ACM SIGPLAN Notices*, vol. 16, pp. 51–63, 1981.

This paper proposes complexity measures called scope number and scope ratio, which are based on the topological analysis of the software's flow of control. The authors recognize that the complexity at each node may not be equal, and assign a complexity to it, depending on the complexity of the statements within the node. The scope number is the aggregate of the complexities of the nodes and gives the complexity of the program as a whole. The authors also introduce the concept of scope ratio, which is a ratio of the number of nodes in the program to the scope number.

[B41] Harrison, W., Magel, K., Kluczny, R., DeKock, A., "Applying software complexity metrics to program maintenance," *IEEE Computer*, vol. 15, no. 9, pp. 65–79, Sept. 1982.

This paper describes many of the popular complexity metrics in each of four categories: control flow, size, data, and hybrid. The conclusion offers a table of metrics surveyed and a qualitative evaluation of the scope of usefulness and the effectiveness for discriminating program text based on complexity.

[B42] Harrison, W. and Cook, C. R., "A note on the Berry-Meekings style metrics," *Communications of the ACM*, vol. 29, no. 2, pp. 123–125, Feb. 1986.

This paper shows that based on how closely a program conforms to a given set of rules, testing C-language programs against a variant of the Berry-Meekings style metrics reveals little relationship between style scores and error proneness.

[B43] Harrison, W., "How complex is your software?," *Computer Language,* pp. 73–75, Jan. 1988.
This article contains an explanation of complexity measures, using PASCAL programs as an example, and explains why software complexity is important to the programmer.

[B44] Henry, S. and Selig, C., "Predicting source-code complexity at the design stage," *IEEE Software,* vol. 7, no. 2, pp. 36–44, Mar. 1990.
This paper states that the use of design metrics allows for determination of the quality of source code by evaluating design specifications before coding, causing a shortened development life cycle.

[B45] Henry, S. M., Kafura, D., and Harris, K., "On the relationships among three software metrics," *Proceedings of ACM SIGMETRICS,* pp. 81–89, 1981.
This paper reports the results of correlation studies made among three complexity metrics that were applied to the same software system. The primary result of this study is that Halstead's and McCabe's metrics are highly correlated, while the information flow metric appears to be an independent measure of complexity.

[B46] Henry, S. M. and Kafura, D., "Software structure metrics based on information flow," *IEEE Transactions on Software Engineering,* vol. 7, no. 5, pp. 510–518, May 1981.
This paper defines the basic notions of the information flow technique and analyzes the UNIX operating system to demonstrate the application of the technique and proposes a structural complexity measure, which is used to evaluate the procedure and module structure of UNIX. The authors validate this metric by demonstrating that its correlation to the occurrence of system changes is as high as 0.98.

[B47] Hess, J. A., "Measuring software for its reuse potential," *1988 Proceedings Annual Reliability and Maintainability Symposium,* pp. 202–206, Jan. 1988.
This article examines a variety of measures for software, and it shows why the physical properties of software are important for determining reuse potential. It also provides design guidelines for the structuring and analysis of software to enhance its reusability.

[B48] Humphrey, W. S., Sweet, W. L., Edwards, R. K., et al, *A Method for Assessing the Software Engineering Capability of Contractors,* CMU/SEI-87-TR-23, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA.
This document provides guidelines and procedures for assessing the ability of potential DOD contractors to develop software in accordance with modern software engineering methods. It includes specific questions and a method for evaluating the results.

[B49] Ince, D., "Software metrics: Introduction," *Information and Software Technology,* vol. 32, no. 4, pp. 297–303 May 1990.
This paper outlines the nature of software quality metrics, places them in historical context, and describes how they might be used.

[B50] Iyengar, S. S., Parameswaran, N., and Fuller, J.W., "A measure of logical complexity of programs," *Journal of Computer Languages,* vol. 7, pp. 147–160, 1982.
The authors point out the shortcomings of Halstead's metrics, McCabe's metric and the knots measure in quantifying complexity of programs. They propose a measure based on the dependency of values of variables upon the past computations in the program, the inductive effort involved in writing the loops, and the data structure complexity.

[B51] Jensen, H. A. and Vairavan, K., "An experimental study of software metrics for real-time software," *IEEE Transactions on Software Engineering,* vol. SE-11, no. 2, pp. 231–234, Feb. 1985.
The authors introduce a new expression, empirically derived, for the program length estimator that predicts program length more accurately for the data gathered than Halstead's length estimator. The relationship between various software metrics were determined by computing Pearson's correlation coefficients for every pair of metrics.

[B52] Johnston, D. B. and Lister, A. M., *An Experiment in Software Science, Language Design and Programming Methodology*, pp. 195–215. Jeffrey M. Tobias, editor, New York: Springer-Verlag, 1980.

Several hundred student Pascal programs were studied. The results do not support Halstead's equations. This may be because of poor program construction and many impurities in the student programs, or may indeed show that Halstead's metrics are not good in practice. This is one of the few papers showing poor results using Halstead's theory.

[B53] Jorgensen, A. H., "A methodology for measuring the readability and modifiability of computer programs," *BIT,* vol. 20, pp. 394–405, 1980.

Various measures of the readability of programs and what program features may be important are discussed. The amount of comments, number of blank lines, average length of variable names, average number of arithmetics operators, and average number of goto's per label are studied.

[B54] Kafura, D. and Redy, G. R., "The use of software complexity in software maintenance," *IEEE Transactions on Software Engineering,* vol. SE-13, no. 3, pp 335–343, Mar. 1987.

This paper reports on a study that relates seven different software complexity metrics to the experience of maintenance activities performed on a medium size software system. Three different versions of the system, including a major revision that occurred during design that evolved over a period of three years, were analyzed in this study.

[B55] Kearney, J. K., Sedlmeyer, R. L., Thompson, W.B., et al, "Software complexity measurement," *Communications of the ACM,* vol. 29, no. 11, pp. 1044–1050, Nov. 1986.

The authors contend that the reason why software metrics have not realized their full potential for the reduction and management of software costs has been due to a lack of a unified approach to the development, testing, and use of these measures. Complexity measures have been developed without any particular use in mind. The authors' suggested approach for the creation of complexity measures is that a clear specification of what is being measured and why it is to be measured must be formulated.

[B56] Khoshgoftaar, T. M. and Munson, J. C., "Predicting software development errors using software complexity metrics," *IEEE Journal on Selected Areas in Communications,* vol. 8, no. 2, pp. 253–261, Feb. 1990.

This paper develops predictive models that incorporate a functional relationship of program error measures with software complexity metrics and metrics based on factor analysis of empirical data. Specific techniques for assessing regression models are presented.

[B57] Kitchenham, B. A., "Measures of programming complexity," *ICL Technical Journal,* vol. 2, no. 3, pp. 298–316, 1981.

This paper reports on an attempt to assess the ability of the measures of complexity proposed to provide objective indicators of the effort involved in software production when applied to selection subsystems of an operating system.

[B58] Kitchenham, B. A., "Towards a constructive quality model," *Software Engineering Journal,* July 1987.

This paper proposes a quality model (COQUAMO) whose goal is to identify a predictive model of product quality based on quality drivers.

[B59] Kitchenham, B. A.and Linkman, S. J., "Design metrics: in practice," *Information and Software Technology,* vol. 32, no. 4, pp. 304–310, May 1990.

This paper describes a method of software quality control based on the use of software metrics and applies this method to software design metrics.

[B60] Kitchenham, B. A. and Walker, J. G., "The meaning of quality," *Software Engineering 86*, pp. 393–406. D. Barnes and P. Brown, editors, United Kingdom: Peter Peregrinus Ltd.

This article discusses some of the problems relating to the use of quality factors and associated criteria.

[B61] Konstam, A. H. and Wood, D. E., "Software science applied to APL," *IEEE Transactions on Software Engineering,* vol. SE-11, no. 10, pp. 994–1000, Oct. 1985.

This paper contains an investigation into the application of software metrics to APL to try to resolve some of the inconsistency and counter-intuitive results found in previous studies. Evidence is presented that verifies that APL has a higher language level than any other common programming language previously studied.

[B62] Koss, W. E., "Software reliability metrics for military systems," *1988 Proceedings Annual Reliability and Maintainability Symposium,* pp. 190–194, Jan. 1988.

This article discusses the statistics of software reliability and why reliability is critical.

[B63] Levitin, A. V., "How to measure software size, and how not to," *Tenth International Computer Software and Applications Conference,* Chicago, IL, Oct. 8–10, 1986.

This article includes critiques of such metrics as lines of code, statement counts, and Halstead's software science. This article suggests the use of token counts.

[B64] Li, H. F. and Chung, W. K., "An empirical study of software metrics," *IEEE Transactions on Software Engineering,* vol. 13, no. 6, pp. 697–708, June 1987.

This article shows a comparison and correlation of 31 different software metrics applied to 250 small programs. This was done by students.

[B65] Lind, R. K.and Vairavan, K. "An experimental investigation of software metrics and their relation to software development effort," *IEEE Transactions on Software Engineering,* vol. 15, no. 5, pp. 649–653, May 1989.

This paper reports the results of a study of software metrics for a large software system used in a real-time application.

[B66] Lister, A. M., "Software science—The emperor's new clothes," Technical Report No 21, Department of Computer Science, Australia: University of Queensland, Oct. 1980.

Halstead's software science is examined and critiqued.

[B67] McCall, J. A., "An assessment of current software metric research," *EASCON '80,* pp. 323–333, 1980.

This paper surveys work in software metrics and covers how metrics may be used, classes of metrics, validation of metrics, applications of metrics, theoretical approaches to metrics, and the role of metrics in quality assurance and meeting the customer's needs.

[B68] Melton, A., Gustafson, D., Bieman, J., and Baker, A., "A mathematical perspective for software measures research," *Software Engineering Journal,* Sept. 1990.

This paper identifies and analyzes the basic principles that underlie software measures research. It also discusses the difference between structure complexity and psychological complexity and the importance of not equating them.

[B69] Mizuno, Y., "Software quality improvement," *IEEE Computer Society,* vol. 16, no. 3, pp. 66–72, Mar. 1983.

This article stresses that teamwork is the key to improved software quality at Nippon Electric Company. The metrics that drive their productivity are specific to volatility, user complexity, generality of the product, requested quality type of contract and human factor, with the last item being the most important.

[B70] Munson, J. C. and Khoshgoftaar, T. M., "The dimensionality of program complexity," *Proceedings of the 11th International Conference on Software Engineering,* May 15–18, 1989, pp. 245–251. Los Alamitos, CA: IEEE Computer Society Press.

This paper examines some recent investigations in the area of software complexity, using factor analysis to begin an exploration of the actual dimensionality of the complexity metrics. Some correlation coeffi-

cients from recent empirical studies on software metrics were factor analyzed, showing the probable existence of five complexity dimensions within thirty-five different complexity measures.

[B71] Musa, J. D., "Software reliability measurement," *The Journal of Systems and Software,* vol. 1, no. 3, pp. 223–241, 1980.

A model of software reliability, based on the execution time of programs, is developed. The model relates execution (CPU) time to calendar time. This information is useful for scheduling projects and for monitoring program testing. The estimates can be changed as the project proceeds.

[B72] Musa, J. D., "The measurement and management of software reliability," *Proceedings of the IEEE,* vol. 68, no. 9, pp. 1131–1143, Sept. 1980.

In this article the author discusses what software reliability is and why it is a useful study. The article includes comparisons between software and hardware reliability, the details of Musa's and Littlewood's reliability theories, and research needs. A lengthy bibliography of related works is included.

[B73] Navlakha, J. K., "A survey of system complexity metrics," *The Computer Journal,* vol. 30, no. 3, pp. 233–238, 1987.

This paper contains a survey of available complexity metrics for software.

[B74] Perlis, A., Sayward, F., and Shaw, M., *Software Metrics: An Analysis and Evaluation.* Cambridge, MA: MIT Press, 1981.

The authors present a survey of metrics literature and research, including a large bibliography of metrics articles.

[B75] Perry, W. E., "Metrics-A tool for defining and measuring quality." Chap. 11 in *Effective Methods of EDP Quality Assurance,* Wellesley, MA: QED Information Sciences Inc., 1981.

This chapter presents procedures and guidelines for introducing and utilizing current quality measurement techniques in a quality assurance program associated with large-scale system developments. These procedures and guidelines explain how to identify and specify software quality requirements.

[B76] Pierce, P., Hartley, R., and Worrells, S., "Software quality measurement demonstration project II," RADC-TR-87-164, Oct. 1987.

This document provides an evaluation of an implementation of the guidelines for specifying software quality attributes presented in RADC-TR-85-37 by Bowen, Wigle, and Tsai [B12]. See also [B104].

[B77] Piwowarski, P., "A nesting level complexity measure," *ACM SIGPLAN Notices,* vol. 17, pp. 44–50, 1980.

The author proposes a new complexity measure to overcome the difficulty of ranking structured versus unstructured, sequential versus nested control structure, and the reduced complexity of case statements. He shows that while metrics based on McCabe's cyclomatic complexity do not necessarily rank structured programs better than unstructured ones or sequential predicates less complex than nested predicates, the proposed measure succeeds in ranking programs so that it penalizes unstructuredness and the nesting of predicates, and rewards the use of case statements.

[B78] Pollock, G. M. and Sheppard, S., "A design methodology for the utilization of metrics within various phases of the software life cycle model," *Eleventh International Computer Software and Applications Conference,* Tokyo, Japan, Oct. 7–9, 1987.

This paper includes a survey of metrics and a description of a general purpose metrics tool.

[B79] Porter, A. A. and Selby, R.W., "Empirically guided software development using metric-based classification trees," *IEEE Software,* vol. 7, no. 2, pp. 46–54, Mar. 1990.

This paper develops a method of generating measurement-based models of high-risk components automatically and a methodology for its application to large software projects. This method, automatic gener-

ation of metric-based classification trees, uses metrics from previous releases or projects to identify components that are likely to have some high-risk property based on historical data.

[B80] Putnam, L. H., "The real metrics of software development," *EASCON '80,* pp. 310–322, 1980.
A set of equations related to Shannon's information theory describing the software development process are given and discussed. Proper use of life-cycle equations can yield optimal solutions and trade-offs in planning the development process.

[B81] Rama Rao, K. V. S. and Iyengar, S., "A general measure for program complexity," *Software Engineering Workshop,* Nice, France, June 1984.
The authors have introduced a general measure that deals with the overall program complexity and can also be extended to encompass all aspects of a program control structure, data structure, and operating sequence. The measure is shown to be more powerful than the cyclomatic measure, because it reflects the structuredness better. Unlike cyclomatic complexity, which is applicable only to flow graphs, the proposed logical effort measure can be applied to programs in any language as well as flow graphs.

[B82] Ramamoorthy, C. V., Tsai, W. T., Yamaura, T., and Bhide, A., "Metrics guided methodology," *Ninth International Computer Software and Applications Conference,* Chicago, IL, Oct. 9–11, 1985.
This paper presents a discussion of metric taxonomy and properties, various program models and associated metrics, and the role of metrics in design and test.

[B83] Redish, K. A. and Smyth, W. F., "Evaluating measures of program quality," *The Computer Journal,* vol. 30, no. 3, pp. 228–232, 1987.
This paper examines available software quality metrics.

[B84] Rodriguez, V. and Tsai, W. T., "Software metrics interpretation through experimentation," *Tenth International Computer Software and Applications Conference,* Chicago, IL, 1986.
This paper presents a statistical comparison of Halstead's metric, cyclomatic complexity, lines-of-code, data flow, and other metrics for bug prediction and maintenance effort.

[B85] Rodriguez, V. and Tsai, W. T., "Evaluation of software metrics using discriminate analysis," *Eleventh International Computer Software and Applications Conference,* Tokyo, Japan, Oct. 7–9, 1987.
This paper presents a statistical investigation of the ability of seven metrics to distinguish between simple and complicated software.

[B86] Roman, D., "A measure of programming," *Computer Decisions,* pp. 32–33, Jan. 1987.
In this article, the author talks about the superiority of the function points technique as a method of measuring programmer productivity. He presents results obtained by various applications of this technique.

[B87] Schneidewind, N. F., "Methodology for validating software metrics," *IEEE Transactions on Software Engineering,* vol. 18, no. 5, pp. 410–421, May 1992.
This paper proposes a comprehensive metrics validation methodology that has six validation criteria, each of which supports certain quality functions. New criteria are defined and illustrated, including consistency, discriminative power, tracking, and repeatability.

[B88] Shen, V. Y., Conte, S. D., and Dunsmore, H. E., "Software science revisited: a critical analysis of the theory and its empirical support," *IEEE Transactions on Software Engineering,* vol. SE-9, no. 2, pp. 155–165, Mar. 1983.
In this article the authors show that the Halstead difficulty (D) has some validity as indicator of error proneness, and the Halstead effort (E) is as good as any other metric for indicating programming effort. Other Halstead metrics are suspect.

[B89] Shen, V. Y., Yu, T. J., Thebaut, S. M., and Paulsen, L. R, "Identifying error-prone software—An empirical study," *IEEE Transactions on Software Engineering,* vol. 11, no. 4, pp. 317–324, Apr. 1985.

Empirical study of 1423 modules averaging 420 LINEs of code relating Halstead's and McCabe's metrics to bug density.

[B90] Shepperd, M., "A critique of cyclomatic complexity as a software metric," *Software Engineering Journal,* March pp. 30–36, 1988.

This paper critiques McCabe's cyclomatic complexity. It states that it is based upon poor theoretical foundations and an inadequate model of software development.

[B91] Shepperd, M., "Early life-cycle metrics and software quality models," *Information and Software Technology,* vol. 32, no. 4, pp. 311–316, May 1990.

This paper describes an mechanism for integrating early metrics into the software development environment to improve software quality.

[B92] Sheppard, S. B., Milliman, P., and Curtis, B., "Experimental evaluation of on-line program construction," *COMPSAC 80,* pp. 505–510, Oct. 1980.

This article proposes that Halstead's effort and volume metrics and McCabe's v(G) were better estimators of programming time than the number of lines of code. Documentation for the problems did not seem to affect performance.

[B93] Sikaczowski, R. O., "Measuring customer satisfaction and software productivity through quality metrics," *Quality Data Processing,* pp. 37–44, Jan. 1988.

Software quality metrics can be used to provide data for cost/benefit analysis while measuring customer satisfaction, evaluating the effects of new tools or methodologies, and helping manage/enhance quality processed during the software life cycle. Metrics of interest to three levels of management (executive, middle, and first line) are presented.

[B94] Singh, R. and Schneidewind, N. F., "Concept of a software quality metrics standard," *Digest of Papers, Spring COMPCON 86,* Mar. 3–6, pp. 362–368. A.G. Bell, editor, Los Alamitos, CA: IEEE Computer Society, 1986.

This paper presents a concept of software quality, which provides the framework for the development of this standard.

[B95] Sunohara, T., Takano, A., Uehara, K., and Ohkawa, R., "Program complexity measure for software development management," *Fifth International Conference on Software Engineering,* San Diego, CA, March 9–12, 1981.

McCabe's metric, Halstead's metrics, lines of code, and other metrics are compared and correlated to productivity and bug rate for 137 modules.

[B96] Symons, C. R., "Function point analysis: Difficulties and improvements," *IEEE Transactions on Software Engineering,* vol. 14, no. 1, pp. 2–11, Jan. 1988.

This article contains a critique and suggested enhancements for the function point metric.

[B97] Tse, T. H., "Towards a single criterion for identifying program unstructuredness," *The Computer Journal,* vol. 30, no. 4, pp. 378–380, Aug. 1987.

This paper defines the concept of fully embedded skeletons and partially overlapping skeletons in program flowgraphs. A program is determined unstructured if it contains partially overlapping skeletons.

[B98] Trachtenberg, M., "Validating Halstead's theory with System 3 data," *IEEE Transactions on Software Engineering,* vol. 12, no. 4, p. 584, Apr. 1986.

This letter disagrees with Thayre's software reliability data being used to validate Halstead's software science theory.

[B99] Tsai, W. T., Lopez, M. A., Rodriguez, V., and Volovik, D., "An approach to measuring data structure complexity," *Tenth International Computer Software and Applications Conference,* Chicago, IL, Oct. 8–10, 1986.

This paper proposes a topological complexity metric for data structures.

[B100] Van Verth, P. B., "A program complexity model that includes procedures," *Eleventh International Computer Software and Applications Conference,* Tokyo, Japan, Oct. 7–9, 1987.

This paper proposes a hybrid data-flow/control-flow structural metric.

[B101] Waguespack, L. J, and Badlani, S., "Software complexity assessment: an introduction and annotated bibliography," *ACM SIGSOFT Software Engineering Notes,* vol. 12, no. 4, pp. 52–71, Oct. 1987.

This article includes an introduction to software complexity, an index of 19 topics, and an annotated bibliography.

[B102] Walker, J. G. and Kitchenham, B. A., "The architecture of an automated quality management system," *ICL Technical Journal,* vol. 6, Issue 1, pp. 156–170, May 1988.

This article shows general support for the draft version of this standard and how it can fit into a wider framework of quality management.

[B103] Wang, A. S. and Dunsmore, H. D., "Back to front programming effort prediction," *Empirical Foundations of Information and Software Science, Proceedings of the First Symposium,* pp. 139–150, Nov. 1982.

This paper discusses metrics for estimating the total software development effort: designing, coding, testing, and documenting. The main obstacle for the proper functioning of prediction models is the lack of availability of reliable estimates of the values of the parameters used in the models. The authors suggest that this could be overcome by trying to estimate the values only at a later point in the development process, when the estimate should be more reliable. They propose a family of metrics that will explain effort after the completion of a project and predict remaining effort when applied at some milestone prior to completion.

[B104] Warthman, J., "Software Quality Measurement Demonstration Project I," RADC-TR-87-247, Dec. 1987.

This document provides an evaluation of an implementation of the guidelines for specifying software quality attributes presented in RADC-TR-85-37 by Bowen, Wigle, and Tsai [B12]. See also [B76].

[B105] Weiser, M. D., Gannon, J. D., and McMullin, P. R., "Comparison of structural test coverage metrics," *IEEE Software,* vol. 2, no. 2, pp. 80–85, Mar.,1985.

This paper contains comparisons of data definition use (du-path), statement, and branch coverage metrics and variations.

[B106] Weyuker, E. J., "evaluating software complexity measures," *IEEE Transactions on Software Engineering,* vol. 14, no. 9, pp. 1357–1365, Sept. 1988.

This paper contains a formal analysis of axioms for complexity metrics and evaluations of several popular metrics and their problems.

[B107] Woodfield, S. N., Shen, V. Y., and Dunsmore, H. E., "A study of several metrics for programming efforts," *J. System Software,* vol. 2, pp. 97–103, Dec. 1981.

This paper reports on an empirical study designed to compare three measures of complexity: lines of code measure, McCabe's cyclomatic complexity, and Halstead's effort measure. It also introduces and evaluates a fourth measure proposed by the authors based on a module of programming. The new measure, which estimates time based on the logical module, was found to have better correlation coefficients with actual time taken, than the other three metrics.

[B108] Yau, S.S. and Collofello, S., "Some stability measures for software maintenance," *IEEE Transactions on Software Engineering*, vol. SE-6, no. 6, pp. 545–552, Nov. 1980.

This article proposes logical stability metrics for software maintenance such as those that result in propagation of change within modules and across module boundaries.

[B109] Yau, S.S. and Collofello, J.S., "Design stability measures for software maintenance," *IEEE Transactions on Software Engineering*, vol. SE-11, no. 9, pp. 849–856, Sept. 1985.

The high cost of software during its life cycle can be attributed largely to software maintenance activities. A major portion of these activities deal with the modifications of the software. In this paper design stability measures, which indicate the potential ripple effect characteristics due to modifications of the program at the design level, are presented.

D.2 Bibliography

[B110] Albrecht, A. J., "Measuring application development productivity," *Proceedings, IBM Application Development Symposium*, Monterey, CA, Oct. 14–17, 1979, GUIDE Int and SHARE, Inc., IBM Corp.

[B111] Albrecht, A. J. and Gaffney, J. E., "Software function, source lines of code and development effort prediction: a software science validation," *IEEE Software Engineering Transactions*, vol. 9, no. 6, Nov. 1983.

[B112] Basili, V. R. and Perricone, B. T., "Software errors and complexity: an empirical investigation," *Communications of the ACM*, Jan. 1984.

[B113] Boehm, B. W., *Software Engineering Economics*. Englewood Cliffs, NJ: Prentice Hall, 1981.

[B114] Conover, W. J., *Practical Nonparametric Statistics*. New York: John Wiley & Sons, 1971.

[B115] Gibbons, J. D., *Nonparametric Statistical Inference*. New York: McGraw-Hill. 1971.

[B116] Halstead, H. M., *Elements of Software Science*. New York: North-Holland, 1977.

[B117] Kleinbaum, D. G. and Kupper, L. L., *Applied Regression Analysis and Other Multivariable Methods*. Boston, MA: Duxbury Press, 1978.

[B118] McCabe, T. J., "Complexity measure," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 308–320, Dec. 1976.

[B119] Myers, G. J., *Reliable Software Through Composite Design*. Princeton, NJ: Petrocelli/Charter, 1975.

[B120] Page-Jones, M., *The Practical Guide to Structured Systems Design*. New York: Yourdon Press, 1980.

[B121] Shen, V. Y., Conte, S. D., and Dunsmore, H. E., "Software science revisited: a critical analysis of the theory and its empirical support," *IEEE Transactions on Software Engineering*, vol. 9, no. 2, pp. 155–165, Mar. 1983.

[B122] Sukert, A. and Goel, A., "A guidebook for software reliability assessment," *1980 Proceedings Annual Reliability and Maintainability Symposium*, pp 186–190.

D.3 Standards and related documents

[B123] AFSCP 800-14, *Air Force Systems Command Software Quality Indicators,* 20 January 1987.

[B124] AFSCP 800-43, *Air Force Systems Command Software Management Indicators,* 31 January 1986.

[B125] DOD STARS, *Guidebook for the STARS Measurement Program,* draft, version 1, 30 September 1985.

[B126] DOD-STD-2167A, *Defense System Software Development,* 29 February 1988.

[B127] DOD-STD-2168, *Defense System Software Quality Program,* 29 April 1988.

[B128] ISO/TC97/SC07/WG3—Document N-102, *Definition of 2nd Quality Characteristics,* March 1987.

[B129] ISO/TC97/SC07/WG3—Document N-106, *Software Quality Characteristics,* June 1987.

[B130] ISO/IEC JTC1/SC7/WG3—Document N-120, Evaluation of Software Product—Software Quality Characteristics & Guidelines for Their Use (DP-9126), October 1988.