NISTIR 5459

Quality Characteristics and Metrics for Reusable Software (Preliminary Report)

W. J. Salamon
D. R. Wallace

Prepared by the

U.S. DEPARTMENT OF COMMERCE
Technology Administration
National Institute of Standards and Technology
Gaithersburg, MD 20899

for the

Department of Defense
Ballistic Missile Defense Organization

NIST

NISTIR 5459

Quality Characteristics and Metrics for Reusable Software (Preliminary Report)

W. J. Salamon
D. R. Wallace

Prepared by the

U.S. DEPARTMENT OF COMMERCE
Technology Administration
National Institute of Standards
and Technology
Gaithersburg, MD 20899

for the

Department of Defense
Ballistic Missile Defense Organization

May 1994

ABSTRACT

This report identifies a set of quality characteristics of software and provides a summary of software metrics that are useful in measuring these quality characteristics for software products. The metrics are useful in assessing the reusability of software products.

This report is preliminary. Additional research is needed to ensure the completeness of the quality characteristics and supporting metrics, and to provide guidance on using the metrics.

KEYWORDS

Reusable Software; Quality Characteristics; Software Reliability; Completeness; Correctness; Software Metrics

EXECUTIVE SUMMARY

The Software Producibility Manufacturing Operations Development and Integration Laboratory (Software Producibility MODIL) was established at the National Institute of Standards and Technology (NIST) in 1992. The Software Producibility MODIL was one of four MODILs instituted at national laboratories by the U.S. Department of Defense/Ballistic Missile Defense Organization (BMDO).[1] The purpose of the MODILs was to investigate technology that could be transitioned from the research community to improve software development within the BMDO.

The initial focus of the Software Producibility MODIL was software reuse. When software is considered for reuse, especially in the high-integrity[2] applications for the BMDO, the quality of the software is of paramount importance. Organizations considering existing software for reuse must consider many variables to decide if the software is fit for reuse. One variable is the quality of the software; hence, organizations must be able to assess the quality of software. Organizations who are developing new software that is intended for reuse must also be able to assess whether the software meets reusability criteria.

This report on quality characteristics and metrics for reusable software is preliminary. It identifies a set of quality characteristics that are most commonly referenced in technical literature and standards. These quality characteristics are common to all software products (e.g., requirements documentation, design documentation, code, and test documentation.) Different metrics for each product may be used to assess the degree to which the product contains each quality characteristic. This report provides some explanation of the value of each metric for determining the reusability of the software. However, more research is needed to ensure the completeness of the quality characteristics and associated metrics.

This preliminary study alone does not provide sufficient measurement information for determining the reusability of software. For example, the value of each characteristic and each metric for each product should be correlated to a reusability index. The reusability index, which no one yet has defined, should also include process metrics (e.g., effort). Most of the metrics have been used for several years and are well-understood relative to structured design methods but most have not yet been applied to software developed with object-oriented technology. Another important topic to be considered for reusing software in high integrity applications is the knowledge about the software relative to its specific domain and application within that domain and the type of information that must be present for analysts to decide how the software must change to meet new requirements.

[1]The BMDO was formerly known as the Strategic Defense Initiative Organization (SDIO).

[2]High integrity software is software that can, and must be, trusted to operate dependably in some critical function (e.g., national defense systems).

Additional research topics include, but are not limited to, the following:

- algorithms for relating measures to a reusability index
- criteria for selecting specific metrics
- user of the data (program manager, technical staff)
- purpose for which data will be used
- constraints (e.g., language, development environment, products available)
- value (e.g., quality characteristic most important)
- ability to collect and analyze data (e.g., automated support for specific measures).
- application of the quality characteristics and metrics to object-oriented development methods
- mapping of characteristics and metrics to existing and draft standards
- guidance on collecting measurement data for the metrics
- guidance on using the resulting measures

TABLE OF CONTENTS

1. INTRODUCTION

The purpose of this report is to summarize a set of metrics that are useful in measuring certain quality characteristics for software. In particular, these characteristics are applicable in assessing the reusability of software products. The quality characteristics are defined first. For each software product (requirements, design, code, test documentation), several metrics are given that will help to qualify the quality characteristics for that product.

A definition of software quality characteristics as given in [ISO9126] is "A set of attributes of a software product by which its quality is described and evaluated." The set of attributes includes functionality, reliability, usability, efficiency, maintainability, and portability. Several other documents ([SAC], [GPALS]) include other characteristics (e.g. adaptability, complexity) that apply to assessing the reusability of software products.

The metrics listed in this report are product metrics, and therefore, are meant to be applied to software products. The products are requirements documentation, design documentation, code, and test documentation. Because the focus on quality for software in the past has been on code, there are many more code metrics listed in this report.

A complete measure of reusability should also include process metrics applied to the development process. This report does not include process metrics.

One document, [SAC2], gives three criteria that should be used to evaluate the metrics themselves. These criteria are:

Objective	We can base the metric on the physical attribute of the object of interest and minimize the amount of guessing required.
Observable	We can extract the metric from existing products, such as source code, documentation, and maintenance histories.
Predictive	The metric correlates well with an attribute of interest, such as development effort and maintenance cost.

In this report, the attributes of interest are the quality characteristics of the software product. The metrics and characteristics can then be used to assess the reusability of the product, and that is the overall attribute of interest.

Most of the metrics in this report have been used for several years and are well-understood relative to structured design methods, but most have not yet been applied to software developed with object-oriented technology. Another important topic to be considered for reusing software in high integrity applications is the knowledge about the software relative to its specific domain and application within that domain and the type of information that must be present for analysts to decide how the software must change to meet new requirements. Some work on this topic has been conducted by NIST and may be found in [NIST5309].

Section 2 of this report provides definitions for each quality characteristic. Section 3 lists suggested metrics for each quality characteristic which pertain to a specific software product. Section 4 provides a summary of this report.

2. QUALITY CHARACTERISTICS FOR REUSABLE SOFTWARE

The definitions in sections 2.1 and 2.2 are stated as they appear in the referenced documents and may be associated with software reuse. Several quality characteristics have more than one definition.

This report makes no preference on the source of the definitions. The definition used for each characteristic in section 3 of this report is an aggregate meaning taken from the various sources, mixed with the intuitive definition.

2.1 Definitions of Software Quality Characteristics

One standard, [ISO9126], has identified several software quality characteristics: portability, efficiency, reliability, functionality, usability, and maintainability. In this section, completeness and correctness are equivalent to functionality as described in [ISO9126]. Understandability is substituted for usability in this section because the quality characteristics used in this report are meant to be applied to individual software products and not the only final software system.

The other quality characteristics in the following list have been extracted from several documents, including [AFSCP800-14] and [CONTE]. There is no standard set of quality characteristics for assessing software which is widely accepted. The list below is an aggregation of the most common quality characteristics.

adaptability	• The ease with which software can be modified to meet new requirements. [GPALS] [MODIL] • The ease with which software allows differing system constraints and user needs to be satisfied. [MODIL] • (*Flexibility*) The ease with which a system or component can be modified for use in applications or environments other than those for which it was specifically designed. [IEEEGLOSS] • The ease with which software can accommodate to change. [NSWC]
completeness	• The degree to which the component implements all required capabilities. [GPALS] • Contains all references and required items. [SOFTECH]
correctness	• (1) The degree to which a component is free from faults in its specification, design, and implementation; (2) The degree to which a component meets specified requirements or user needs and expectations; (3) The ability of a component to produce specified outputs when given specified inputs, and the extent to which they match or satisfy the requirements. [NISTIR4909] • (1) The degree to which a system or component is free from faults in its specification, design, and implementation; (2) The degree to which

	software, documentation, or other items meet specified requirements; (3) The degree to which software, documentation, or other items meet user needs and expectations, whether specified or not. [IEEEGLOSS] Strict adherence to specified requirements. [NSWC]
efficiency	• The degree to which a component performs its designated functions with minimum consumption of resources. [IEEEGLOSS] • A set of attributes that bear on the relationship between the level of performance of the software and the amount of resources used, under stated conditions. [ISO9126]
generality	• The breadth of applicability of the component. [GPALS] • The degree to which a system or component performs a broad range of functions. [IEEEGLOSS]
maintainability	• The ease with which a component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment. [IEEEGLOSS] • The ease with which software can be maintained, for example, enhanced, adapted, or corrected to satisfy specified requirements. [FIPS106] • Modifiable with minimal impact. [SOFTECH] • The ease with which corrections can be made in response to recognized inadequacies. [NSWC] • A set of attributes that bear on the effort needed to make specified modifications. [ISO9126] • Degree to which perfective, adaptive and corrective changes can be cost effectively made to the component. [SAC2]
modularity	• The degree to which a system or computer program is composed of discrete components such that a change to one component has minimal impact on other components. [IEEEGLOSS] • The way the component is decomposed into sub-components. [GPALS]
portability	• The ease with which a system or component can be transferred from one hardware or software environment to another. [IEEEGLOSS] • The extent to which a module originally developed on one computer or operating system can be used on another computer or operating system. [MODIL] • Operating environment independence. [GPALS] • Platform independence. [SOFTECH] • The ease in transferring software to another environment. [NSWC] • A set of attributes that bear on the ability of software to be transferred from one environment to another. [ISO9126]

	• Ease with which a software component can be implemented in new applications and virtual machine environments. [SAC2]
reliability	• The ability of a component to perform its required functions under stated conditions for a specified period of time. [IEEEGLOSS] • The error-free use of software over time. [NSWC] • Low error rate. [SOFTECH] • A set of attributes that bear on the capability of software to maintain its level of performance under stated conditions for a stated period of time. [ISO9126] • Expected time between failures while running the application in it's operational environment. [SAC2]
understandability	• The degree to which the meaning of a software component is clear to a user. [NISTIR4909] • (*Clarity*, *Self-Descriptiveness*) Ease of comprehending the meaning of the software (opposite of complexity). [GPALS] • Low complexity and Documentation. [SOFTECH]

2.2 Other Definitions

The definitions in this section are definitions of terms used throughout this report. Several of the terms defined here are characteristics of software products which can be measured (e.g. cohesion and complexity.) These characteristics are used to further define the quality characteristics from section 2.1 in sections 3.1 through 3.5.

application domain	• The knowledge and concepts that pertain to a particular computer application area. [STARS] • An identifiable area or subarea of an organization's software development activities in which similar software requirements occur. [GPALS]
cohesion	• The binding of statements within a software component. [NSWC] • The manner and degree to which the tasks performed by a single software module are related to one another. [IEEEGLOSS] • The degree to which the functions or processing elements within a module are related or bound together. [FIPS106] • The degree to which a component's structure is unified in support of its function. [MODIL]
cohesiveness	• The degree to which a component's structure is unified in support of its function. [GPALS]

complexity	• An abstract measure of work associated with a software component. [NSWC] • The degree to which a system or component has a design or implementation that is difficult to understand and verify. [IEEEGLOSS] • The degree of complication of a system or system component, determined by such factors as the number and intricacy of interfaces, the number and intricacy of conditional branches, the degree of nesting, the types of data structures, and other system characteristics. [FIPS106]
component	• One of the parts that make up a system. A component may be hardware or software and may be subdivided into other components. [IEEEGLOSS]
coupling	• The interdependence among software components. [NSWC] • The manner and degree of interdependence between software modules. [IEEEGLOSS] [MODIL] • (1) The degree of data or control connectivity between different components of a software system; (2) A measure of the strength of interconnection between one component and another. [MODIL] • (*Interconnectivity*) The degree of connectivity between different components. [GPALS] • The degree that modules are dependent upon each other in a computer program. [FIPS106]
domain	• An area of activity or knowledge. [STARS] • A group or family of related systems that share a set of common capabilities and/or data. [SOFTECH] • A distinct functional area that can be supported by a class of software systems with similar requirements and capabilities. [MODIL]
expandability	• (*Augmentability*) The ability to support expansion of data-storage requirements. [GPALS] • (*Extendibility*, *Extensibility*) The ease with which a system or component can be modified to increase its storage or functional capacity. [IEEEGLOSS] • (*Extensibility*) The extent to which a component allows new capabilities to be added and existing capabilities to be easily tailored to user needs. [MODIL]
fault	• (*Defect*) An incorrect step, process, or data definition in a computer program. [IEEEGLOSS]
functionality	• A set of attributes that bear on the existence of a set of functions and their specified properties. The functions are those that satisfy stated or implied needs. [ISO9126]

readability	• The difficulty in understanding a software component. [NSWC]
reusability	• The degree to which a component can be used in more than one software system, or in building other components, with little or no adaptation. [MODIL]
robustness	• The degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions. [IEEEGLOSS]
	• Measure of the breadth of the new problem domain addressed by the current system, its subcharacteristics are generality and expandability. [SAC]
	• Ability to produce correct results despite input errors. [GPALS]
testability	• The ability to evaluate conformance with requirements. [NSWC]
	• Equipped with test plans. [SOFTECH]
traceability	• The ease in retracing the complete history of a software component from its current status to its requirements specification. [NSWC]
	• (1) The degree to which a relationship can be established between two or more products of the development process, especially products having a predecessor-successor or master-subordinate relationship to one another; for example, the degree to which the requirements and design of a given software component match; (2) The degree to which each element in a software development product establishes its reason for existing; for example, the degree to which each element in a bubble chart references the requirement that it satisfies. [IEEEGLOSS]
	• The characteristic of software systems or designs or architectures or domain models that identifies and documents the derivations path (upward) and allocation/flowdown path (downward) of requirements or constraints. [STARS]
unit	• (1) A separately testable element specified in the design of a computer software component; (2) A logically separable part of a computer program; (3) A software component that is not subdivided into other components. [IEEEGLOSS]
usability	• A set of attributes that bear on the effort needed for use, and on the individual assessment of such use, by a stated or implied set of users. [ISO9126]

3. QUALITY METRICS FOR REUSABLE SOFTWARE

This section suggests metrics for each quality characteristic associated with software products (requirements documentation, design documentation, code, test documentation). Each product has several quality characteristics, along with metrics to be used in assessing the quality characteristic for the product.

The analyst must identify the objectives to be achieved from collecting and analyzing metric data. It must be possible to collect the data, and methods should exist for analyzing the data. The analyst must present the results such that recommendations can be made concerning the reusability of the software. Obviously, optimum values for all metrics cannot be achieved simultaneously. Selection criteria for requirements for metrics include tradeoffs between the quality characteristics and their priorities. For example, efficiency and understandability often are in conflict with each other; efficient code may be necessarily coded in an assembly language, which reduces the understandability (and hence, maintainability) of the software.

Another selection criterion is the ability to collect the required data for the metric. The simplicity of data collection is one reason lines-of-code remains a popular metric despite its deficiencies. Many other metrics have data collection requirements that can be automated, and therefore, may be less costly to implement. Examples include defect counts and some complexity metrics. Other metrics require more human involvement, such as requirements tracing, function point analysis, and test coverage.

Section 3.1 identifies the metrics for quality characteristics that may be applied to all software products, with minor adjustment. Sections 3.2 through 3.5 identify metrics for the quality characteristics of products of requirements, design, code, and test.

3.1 Quality Metrics for All Products

The metrics for the quality characteristics in this section can be applied to all software products. Most of these metrics are primitive in the sense that they are simple counts of problem report values. Cause and effect graphing and RELY[3] require more analysis of the products.

3.1.1 Completeness

- cause and effect graphing [IEEE982.1]
  Cause and effect graphing aids in identifying requirements that are incomplete and ambiguous. Also, it explores inputs and expected outputs of a program and identifies ambiguities. For test purposes, it is useful for evaluating test cases for probability of detecting faults.

[3]RELY is not an acronym, but is a short form of Required Software Reliability

Primitives:

List of causes: distinct input conditions
List of effects: distinct output conditions or system transformations
$A_{existing}$ = number of ambiguities remaining
A_{tot} = total number of ambiguities identified

The requirements are analyzed and the requirements are broken down into specific causes and effects. Each cause and each effect is assigned a unique node identifier. A Boolean graph is made by connecting the cause and effect nodes based on the semantic content of the requirements. Ambiguities are defined as any cause with no effect, and effect with no cause, and any combination of cause and effects that are inconsistent with the requirements or are impossible to achieve.

Calculation of Cause/Effect Completeness:

$$CE(\%) = 100 \times \left(1 - \frac{A_{existing}}{A_{tot}}\right)$$

When no ambiguities exist, and all causes and effects are represented, then the *CE* measure is 100%. See [IEEE982.2] for use of this metric.

3.1.2 Correctness

- number of problem reports per phase, priority, category, or cause [NIST500-209]
- number of reported problems per time period [NIST500-209]
- number of open real problems per time period [NIST500-209]
- number of closed real problems per time period [NIST500-209]
- number of unevaluated problem reports [NIST500-209]
- age of open real problem reports [NIST500-209]
- age of unevaluated problem reports [NIST500-209]
- age of real closed problem reports [NIST500-209]
- rate of error discovery [NIST500-209]

3.1.3 Reliability

- RELY - Required Software Reliability [IEEE982.1]
  RELY can be used to provide a reliability rating for a product through examination of the processes used to develop it. At the early planning phases of a project, RELY can be used to measure the trade-offs of cost and degree of reliability. RELY provides ratings very low, low, nominal, high, and very high. See [IEEE982.2] for a table of reliability factors for requirements and product design, detailed design, code and unit test, and integration and test.

3.2 Quality Metrics for Requirements Documentation

Because requirements documentation is usually written in human readable format, the metrics that have been defined for requirements typically require manual analysis. The data needed for many of the metrics must be gathered by hand (i.e. counting requirements). Many of the metrics are subjective in nature, and it is up to the analyst to decide what is a "good" value. For example, the analyst must determine the acceptable readability level of the documents.

Many of the metrics for quality characteristics in this section can be used for other products. The readability metrics may be applied to preliminary and software system design documents, for example.

3.2.1 Completeness

- completeness metric [IEEE982.1] [AFSCP800-14]
  This metric uses eighteen primitives (number of functions not satisfactorily defined, number of functions, number of functions not used, etc.). Ten derivatives are defined (ratio of functions satisfactorily defined to functions defined, ratio of defined functions used to defined functions, etc). The completeness measure (*CM*) is calculated as the weighted sum of the ten derivatives expressed as: (See [IEEE982.2])

$$CM = \sum_{i=1}^{10} w_i D_i$$

  where:

  w_i = weighting factor for derivative D_i

  D_i = derivative number i

- requirements traceability [IEEE982.1]
  Aids in identifying requirements that are missing from, or in addition to, the original requirements. *R1* is number of requirements met by the architecture. *R2* is the number of original requirements. The traceability measure (*TM*) is computed:

$$TM = \frac{R1}{R2} \times 100\%$$

- *deviation between* planned number of System/Segment Design Document (SSDD) software requirements to be documented in the Software Requirements Specification (SRS) *and* actual number of SSDD software requirements completely documented in the SRS [AFP800-48]

- Document Relationships [NSWC2]
  Document relationship is defined as "the structure among and within the individual documents of a set of documentation, which ties the documents together into a single, unified set." There are two categories: *Decompositional* and *Referential.*

  Decompositional relationship is the natural, hierarchal ranking of a set of documentation. Documents at a lower level in the hierarchy give more detail on a given subject. For completeness, there should exist a definite or obvious hierarchial ranking among the documents.

  Referential relationships are the links between documents; i.e. a specific cited reference to another document or another section in the same document. Appropriate references to another documents (or within the same document) should be maintained.

3.2.2 Correctness

- number of discrepancies as a result of each review [STEP]
- number of conflicting requirements [IEEE982.1]
- requirements compliance [IEEE982.1]
  System Verification Diagrams (SVDs) are used to detect inconsistencies, incompleteness, and misinterpretations in the requirements. Requirement errors detected using the SVDs are:

  N_1 = Number of errors due to inconsistencies
  N_2 = Number of errors due to incompleteness
  N_3 = Number of errors due to misinterpretation

- requirement errors reported / total number of requirements [GPALS2]
- requirement errors corrected / total number of requirements [GPALS2]
- number of requirements faults and structural design faults detected during detailed design [NIST500-209]

3.2.3 Generality

- size of the application domain [STARS]
  Reverse engineering performed on the requirements documentation is useful in identifying common design and architectural features of existing systems. Reusable requirements specifications must define the boundaries of the problem space, as well as a set of variability descriptions. Requirements documentation can then be assessed for application as subdomains of other domain areas.

3.2.4 Understandability

- size metrics
  - number of requirements [NIST500-209]
    this is often expressed as the number of *shalls*
  - number document pages [AMI] [NIST500-209]
  - number of document words [NIST500-209]
  - number of functions [NIST500-209]

- readability metrics
  - number of grammatically incorrect statements
  - number of misspellings
  - readability indices such as Flesch-Kincaid, Gunning's Fog Index [MURRAY]
    These readability formulas are based on sentence length and polysyllable frequency. Gunning's Fog index is used to produce a grade level required for reading the document. For example, an index of 12 means that 12 years of education is required to understand the document.

  - physical readability [NSWC2]
    Four indicators of physical readability are given: *format appropriateness, adequacy of print, format consistency,* and *module appropriateness.*
    Format appropriateness assesses the suitability of the presentation style or layout. For example, numeric data should be presented in a table or graph.
    Adequacy of print includes quality of reproduction, font sizes, paper quality, and font styles. Assessment can be done by sampling document pages.
    Format consistency means that tables should all have the same format and indexes should all have the same format. The same rule should be applied to Tables of Contents, chapters, sections, etc. Again, sampling is used to provide measurements.

    Module appropriateness measures the suitability of the physical division of the chapters, sections, paragraphs, tables, figures, and so forth. The assessment is whether the document is physically divided in a manner consistent with the logical division of the material.

- complexity
  - function points [NIST500-209] [SQE] [SYMONS]
    Function point analysis was developed by Alan Albrecht at IBM in the 1970's, and furthered refined by Albrecht and others. As a measurement of size, function points are language independent (as opposed to lines-of-code), and can be applied to requirements specifications. Applying to requirements allows estimates of size and complexity earlier in the life cycle.

The first part of calculating function points is determining *information processing size*, measured in unadjusted function points (*UFP*s). The application is divided into five types of components: inputs, outputs, enquiries (combinations of inputs and outputs that give immediate results), interfaces to other applications, and logical internal files. Each component is classified as simple, average, or complex, and is assigned a simple number of weighting points. The *UFP* count for the system is the sum of the individual *UFP*s.

Second, a *technical complexity adjustment* (*TCA*) is determined by estimating the degree of influence of general application characteristics, such as data communications, performance, on-line update, etc. *TCA* is calculated:

$$TCA = 0.65 + 0.01 \times DI$$

where *DI* is the total degree of influence.

Finally, the *function point* (*FP*) count is computed as:

$$FP = UFP \times TCA$$

- Mk II Information Processing Size [SYMONS]
This metric looks evaluates the system as a collection of *logical transactions*, each consisting of input, process, and output components. A logical transaction is defined as a unique input/process/output combination triggered by a unique event, or a need to retrieve information. Mk II Information Processing Size also uses *Unadjusted Function Points* (*UFP*s) calculated with the formula:

$$\begin{aligned} UFP's = {} & W_I \times (\textit{no. of input data element-types}) \\ & + W_E \times (\textit{no. of entity-types referenced}) \\ & + W_O \times (\textit{no. of outputdata element-types}) \end{aligned}$$

where:

W_I = weighting factor for input data element-types
W_E = weighting factor for entity-types
W_O = weighting factor for output data element-types

The weighting factors can be determined by calibration.

Next, the *technical complexity adjustment* (*TCA*) is calculated:

$$TCA = 0.65 + C \times DI$$

where:

DI = the total degree of influence
C = coefficient obtained by calibration

Finally, the *MkII Function Points* is calculated as *UFP* x *TCA*.

3.3 Quality Metrics for Design Documentation

Software design documentation is often divided into three activities: functional allocation, software system design, and unit design. These software design activities occur in three chronological phases: preliminary design, detailed design, and unit design [CARD].

The first phase is preliminary design. Designers collect related requirements into functional groups and identify dependencies among functions. The preliminary design may be represented by data flow diagrams, high-level structure charts, or a simple list of requirements by subsystems [CARD]. The choice of metrics depends on the representation used. Requirements traceability can be applied to any representation to verify that requirements are being met. Data flow complexity can be applied to data flow diagrams to evaluate the understandability of the diagrams.

Next comes detailed design, where the overall architecture of the software system is defined. This step allocates data and functions to individual units or design parts. Internal interfaces must also be specified at this stage [CARD]. A structure chart is commonly used to represent the system design. Some of the metrics applied to this type of design are data or information flow complexity, external (D_e) complexity, fan-in/fan-out per module, graph-theoretic complexity, and readability metrics.

The final design phase is unit design. In this phase, algorithms and data structures are defined. Application and implementation specific information is added to the design. The design itself is often represented as pseudo-code and module prologues [CARD]. Nearly all of the metrics specified below can be applied to unit design documents. Some metrics specific to unit designs are internal (D_i) complexity, document lines of code, and number of states per parameter.

The unit design phase is the first indication of the reusability of individual modules. For example, the complexity of a module can be determined by its design, before any coding is done. Many of the same metrics that are typically applied to code can be applied to unit designs. Modularity can often be assessed at the unit design level. See [CARD] for references to studies on heuristics for achieving modularity. These heuristics are small modules, limited data coupling, medium span of control, and singleness of purpose. Modularity is a good indicator of the reusability of an individual software module.

3.3.1 Completeness

- requirements traceability [IEEE982.1] (Applied to all designs)
  For design, this metric indicates the percent of requirements that have been documented in the design. See **Quality Metrics for Requirements Documentation** (section 3.2) for a description of calculating this metric. See [IEEE982.2] for a complete description of this metric.

o *deviation between* planned number of Software Requirements Specification (SRS) requirements to be documented as Computer Software Components (CSC) into the Software Design Document (SDD) *and* actual number of SRS requirements completely documented as CSCs in the SDD [AFP800-48] (Applied to system design)

3.3.2 Correctness

o defect density [IEEE982.1] [NIST500-209] (Applied to unit design)
For design, the defect density is calculated after each design inspection of new development or large block modifications. A structured design language is assumed. See [IEEE982.2] for a description on using this metric.
Primitives:
D_i = total number of defects found during i^{th} design inspection
I = total number of inspections to date
$KSLOD$ = number of source lines of design statements in thousands
Calculation of Defect Density:

$$DD = \frac{\sum_{i=1}^{I} D_i}{KSLOD}$$

Design defect density may also be measured in terms of *design defects/KSLOC* [SPC]. Each defect from later phases that is determined to be a design defect is counted.

o number of structural (architectural) design faults detected during detailed design [NIST500-209] (Applied during unit design)
o number of design faults associated with each module [NIST500-209] (Applied to system and unit design)
o number of integration test cases planned/executed involving each module [NIST500-209] (Applied to system and unit design)
o number of black box test cases planned/executed per module [NIST500-209] (Applied to system and unit design)
o number of design errors reported / total number of units [GPALS2] (Applied to all)
o number of design errors corrected / total number of units [GPALS2] (Applied to all)

3.3.3 Generality

o size of application domain [STARS] (Applied all designs)
Reverse engineering can be done on system designs to identify data structures and data management patterns to aid in separating the functionality into two categories: functionality that supports general domain concepts and functionality that achieves a computer-based solution.

Reverse engineering can be done on unit designs to identify the modularization, relationship among structural elements, declare/set/use patterns for variables, control flow within structural elements, and scoping information. The information extracted can be used to identify solution concepts and their interrelationships. Also, requirement choices can be connected with design choices and the effect of performance, timing, sizing, and functionality.

3.3.4 Efficiency

The percent metrics that follow can be applied to system and unit designs if the target capacity for CPU and I/O usage are known. Likewise, if the target random access memory (RAM) and storage capacities are known, then percent usage calculations may be made. These estimates may then be used to assess the efficiency of the design and its reusability in new environments. However, comparisons made between environments must take into account processor speeds, I/O devices, and operating system effects.

- target CPU usage as percent of capacity [STEP]
- target I/O usage as percent of capacity [STEP]
- target upper bound storage usage [STEP]
- percent actual of target upper bound storage usage [STEP]
- target upper bound RAM usage [STEP]
- percent actual of target upper bound RAM usage [STEP]

3.3.5 Modularity

- cohesion metric [NIST500-209] [SQE] [CARD] (Applied to unit design)
  Cohesion, or module strength, refers to the relationship among the elements of a module. The cohesion value for a module is assigned using a standard rating chart, that can be found in [SQE]. The best cohesion level is *functional*, and the worst is *coincidental.*

  A high-strength (functional) module performs one function. A low-strength (coincidental) module includes multiple unrelated functions. Studies referenced in [CARD] find that *higher strength modules tend to have lower fault rates*, and *tend to cost less to develop*. Also, the modularity is greater for higher strength modules.

  The Software Measurement Guidebook [SPC] gives a strength metric proposed by Cruickshank and Gaffney. Strength is a measure of the degree of cohesiveness of the elements of a software module.

Calculation of Strength:

$$Strength = \sqrt{(X^2 + Y^2)}$$

where:

X = reciprocal of the number of assignment statements in the module
Y = number of unique function outputs divided by number of unique function inputs

- coupling [NIST500-209] [SQE] [CARD] (Applied to system and unit designs)
  Coupling is a measure of the degree to which modules share data. Module coupling is rated using a standard rating chart, that can be found in [SQE]. *Data coupling* is the best type of coupling, while *content coupling* is the worst.

  Data coupling is the sharing of data via parameter lists, while *common coupling* is the sharing of data via global (common) areas. Earlier recommendations stated that common coupling should be avoided. However, later studies have shown that *the distribution of error rate does not depend on the coupling mechanism* [CARD]. However, In terms of modularity and the modules independence of external factors, a lower coupling value is better.

  A coupling metric given in [SPC] and originally proposed by Cruickshank and Gaffney is calculated as follows:
  Calculation of Coupling:

$$Coupling = \frac{\sum_{i=1}^{n} Z_i}{n}$$

where:

$$Z_i = \frac{\sum_{j=1}^{m} M_j}{m}$$

M_j = sum of the number of input and output items shared between components i & j
Z_i = average number of input and output items shared over m components with component i
n = number of components in the software product

3.3.6 Portability

- number of features that are language-specific (Applied to unit design)
- number of features that are operating system specific (Applied to unit design)
  Features that are operating system, hardware, or interface specific should be isolated in modules that can then be changed for other platforms.

3.3.7 Reliability

- cumulative failure profile [IEEE982.1]
  For design, a graph is made of the cumulative failures in the software resulting from design deficiencies. The curve of the graph can be used to predict the reliability of the design. (Applied to unit design)

3.3.8 Understandability

- Design Structure Metric [IEEE982.1]
  Used to determine the simplicity of the detailed design of a software program. The values determined for the primitives can be used to identify problem areas within the software design. See [IEEE982.2] for information on using this metric. (Applied to system and unit designs)
  Primitives:
  P1 = total number of modules in the program
  P2 = number of modules dependent on the input or output
  P3 = number of modules dependent on prior processing (state)
  P4 = number of database elements
  P5 = number of non-unique database elements
  P6 = number of database segments (partition of the state)
  P7 = number of modules not single entrance/single exit
  Derived Metrics:
  D_1 = design organized top-down (Boolean)
  D_2 = module dependence (*P2*/*P1*)
  D_3 = module dependent on prior processing (*P3*/*P1*)
  D_4 = database size (*P5*/*P4*)
  D_5 = database compartmentalization (*P6*/*P4*)
  D_6 = module single entrance/single exit
  Calculation of Design Structure Measure:

$$DSM = \sum_{i=1}^{6} W_i D_i$$

  where W_i is the weight given to the i^{th} derived measure

- size metrics
  Several studies have been done to investigate the affect of module size on the modularity (and hence, reusability) of single modules. The conclusion, as given by [CARD], is that *module size alone does not affect fault rate*. Also, *larger modules cost less per executable statement than smaller ones*. These statements suggest that lines-of-design and -code are not good indicators of modularity. However, when used with other measurements (complexity, etc.), size measures can provide an indication that a

module may need to be separated into smaller modules. The unit design phase is a better time to perform the separation of function, rather than during coding.

- number of design modules [NIST500-209] (Applied to system design)
- number document pages [NIST500-209] (Applied to all design)
- document lines-of-code [NIST500-209] (Applied to unit design)
- number of functions [NIST500-209] (Applied to preliminary and system designs)
- number of inputs and outputs [NIST500-209] (Applied to system and unit designs)
- number of interfaces [NIST500-209] (Applied to system and unit designs)

o complexity metrics
- graph-theoretic complexity for architecture [IEEE982.1] (Applied to system and unit designs)

There are three graph-theoretic complexity metrics:

Static complexity is used to measure the complexity of the software architecture, as represented by a network of modules, useful for design tradeoff analysis.

Generalized static complexity is used to measure of the complexity of the software architecture, as represented by a network of modules and the resources used.

Dynamic complexity is used to measure the complexity of the software architecture as represented by a network of modules during execution.

Primitives:

K = number of resources, indexed by $k = 1,...,K$
E = number of edges, indexed by $i = 1,...,E$
N = number of modules, indexed by $j = 1,...,N$
c_i = complexity for program invocation and return along each edge e
r_{ki} = 1 if k^{th} resource required for i^{th} edge, 0 otherwise
d_k = complexity for allocation of resource k

Calculation of Complexity Measures:

Static Complexity:

$$C = E - N + 1$$

Generalized Static Complexity:

$$C = \sum_{i=1}^{E} \left(C_i + \sum_{k=1}^{K} (d_k \times r_{k_i}) \right)$$

Dynamic Complexity:

Dynamic complexity is calculated using the formula for static complexity at various points in time. The behavior of the measure is then used to indicate the evolution of the complexity of the software.

- number of entries/exits (fan-in/fan-out) per module [IEEE982.1] [NIST500-209] (Applied to unit design)
  This metric can be used to determine the difficulty of the software architecture. It is assumed that a modular specification/design language is used. This metric can also be used to evaluate the encapsulation of the data at the design phase. If data is properly encapsulated, the number of entry and exit points for each module function will be small [IEEE982.2].
  Primitives:
  e_i = number of entry points for the i^{th} module
  x_i = number of exit points for the i^{th} module
  Calculation of Entries/Exits:
  $m_i = e_i + x_i$

- data or information flow complexity [IEEE982.1] (Applied to all designs)
  This metric can be used to evaluate the information flow structure of large systems, the procedure and module information flow structure, and the complexity of interconnections between modules. This metric can also be used for code evaluations. See **Quality Metrics for Code** (section 3.4) for a description of the metric primitives and calculations. See [IEEE982.2] for a complete description of this metric.

- number of parameters per module [NIST500-209] (Applied to system and unit design)
- number of states or data partitions per parameter [NIST500-209] (Applied to unit design)
- decision count [CONTE] (Applied to unit design)

See **Quality Metrics for Code** (section 3.4) for a description of this metric. Can be used, with the above two metrics, to identify early in development modules that are potentially complex or hard to test.

- external (D_e) complexity [ZAGE] (Applied to all designs)
  Based on information available during architecture design such as hierarchial module diagrams, dataflow, functional descriptions, and interface descriptions.

  Calculation of D_e:

  $$D_e = e_1(inflow * outflow) + e_2(fan\text{-}in * fan\text{-}out)$$

  where:
  inflow is the number of data entities passed to the module
  outflow is the number of data entities passed from the module
  fan-in is the number of superordinate modules directly connected to the module
  fan-out is the number of subordinate modules directly connected to the module
  e_1 and e_2 are weighting factors for the two terms

- internal (D_i) complexity [ZAGE] (Applied to unit design)
  Based on information available after detailed design, including information used for D_e plus the chosen algorithms and possibly pseudo-code or program-design-language representations.
  Calculation of D_i:

  $$D_i = i_1(CC) + i_2(DSM) + i_3(I/O)$$

  where:
  CC (central calls) is the number of procedure or function invocations
  DSM (data-structure manipulations) is the number references to complex data types
  I/O number of external device accesses
  i_1, i_2, and i_3 are weighting factors

- composite metric ($D(G)$) to measure design quality [ZAGE]:

  $$D(G) = D_e + D_i$$

o readability metrics
  - number of grammatically incorrect statements (Applied to all designs)
  - number of misspellings (Applied to all designs)
  - readability indices such as Flesch-Kincaid, Gunning's Fog Index [MURRAY] (Applied to preliminary and system designs)
    See **Quality Metrics for Requirements Documentation** (Section 3.2) for a description of these indices)

3.4 Quality Metrics for Code

Much of the research into metrics has focused on code metrics. Hence, there are many different kinds of code metrics, and many variations on common metrics. In terms of reusability, useful metrics are the product metrics are used to measure the size, complexity, and readability of the source program. In order for a component to be reusable, it must be understandable by the software engineers. Also, the component should encapsulate as much implementation detail as possible. Well-defined, simple interfaces are desirable.

In assessing existing components for reusability, it is useful to examine the history of the component in actual use. Fault density, code-related problem counts, defect density, and efficiency are some of the metrics used for this assessment. The longer a component has been in actual use, the higher the confidence in the component's correctness, assuming low fault and defect counts. Also, the testability of the component is critical when reusing the software. A well-defined set of test cases aids in quickly assessing the components use in a new environment. The testability of a component is defined in part by its complexity, as well as its size.

There are many methods used to calculate lines-of-code. Two documents, [IEEE1045] and [SEI], give methods which are used to ensure consistent counting of lines-of-code.

3.4.1 Completeness

- number of ambiguous references [SAC]
  References to inputs, functions, and outputs should be unique. An example of an ambiguous reference is a function being called one name by one module and a different name by another module.

- number of improper data references [SAC]
  All data references should be properly defined, computed, or obtained from identifiable external sources.

- percentage of defined functions used [SAC]
  All functions defined within the software should be used.

- percentage of referenced functions defined [SAC]
  All functions referenced within the software should be defined. There should be no dummy functions present.

- percentage of conditional processing defined [SAC]
  All conditional logic and alternative processing paths for each decision point should be defined.

3.4.2 Correctness

- fault density [IEEE982.1]
  This metric can be used to predict remaining faults by comparison with expected fault density, and determine if sufficient testing has been completed [IEEE982.2]. A fault density is calculated for each severity level.
  Calculation of Fault Density:
  $F_d = F / KSLOC$
  where:
  F = total number of unique faults found in a given time interval resulting in failures of a specified severity level
  $KSLOC$ = number of source lines of executable code and non-executable data declarations in thousands

- number of code-related problems/errors reported [CONTE]
- number of code-related problems fixed [CONTE]
- number of program changes per time period [CONTE]
- number of changed lines of code per time period [CONTE]
- number of coding errors / total number of units [GPALS2]

- defect density [IEEE982.1] [NIST500-209]
  For code, the defect density is calculated after each code inspection of new development or large block modifications. See [IEEE982.2] for information on using this metric.
  Primitives:
  D_i = total number of defects during i^{th} code inspection
  I = total number of inspections to date
  $KSLOC$ = number of source lines of executable code and non-executable data declarations in thousands
  Calculation of Defect Density:

$$DD = \frac{\sum_{i=1}^{I} D_i}{KSLOC}$$

- defect indices [IEEE982.1]
  Defect indices provide a relative index of how correct the software is as it proceeds through the development cycle. For each phase of development, calculate index PI:

$$PI_i = W_1\frac{S_i}{D_i} + W_2\frac{M_i}{D_i} + W_3\frac{T_i}{D_i}$$

where:

D_i = Total number defects detected during the i^{th} phase
S_i = Number of serious defects found
M_i = Number of medium defects found
T_i = Number of trivial defects found
PS = Size of product at the i^{th} phase
W_1 = Weighting factor for serious defects
W_2 = Weighting factor for medium defects
W_3 = Weighting factor for trivial defects

Defect index (DI) is calculated at each phase by cumulatively adding the calculation of P_i as the software proceeds through development.

Calculation of Defect Index:

$$DI = \sum (i \times PI_i)/PS$$

3.4.3 Efficiency (of execution)

- non-loop dependent statement in loops: (number of modules with non-loop dependent statement in loops) / (total number of modules) [SAC]
  Practices such as calculating values treated as constants within loops should be avoided.

- compound expression evaluation: (number of modules with repeated compound expression evaluation) / (total number of modules) [SAC]
  Repeated compound statements should be avoided.

- total number of memory overlays [SAC]
  The use of memory overlays imposes processing overhead and should be avoided.

- amount of non-functional executable code: (number of modules with non-functional executable code) / (total number of modules) [SAC]
  The presence of non-functional executable code is an obvious inefficiency. This condition often arises during maintenance or redesign updates with incomplete removal of obsolete code.

o coding of decision statements: (number of modules with inefficient decision coding) / (total number of modules) [SAC]
Decision statements should be coded for efficient execution, e.g., the most frequently exercised alternative of an IF statement should normally be specified in the THEN clause, rather than in the ELSE clause.

o data grouping: (number of modules with inefficient data grouping) / (total number of modules) [SAC]
Example of inefficient data grouping: complicated nesting of pointers and indices.

o initialization of variables: (number of modules with variables not initialized when declared) / (total number of modules) [SAC]
Efficiency is lost when variables are initialized during execution or repeatedly initialized during iterative processing.

o target CPU usage as percent of capacity [STEP]
o actual CPU usage as percent of capacity [STEP]
o projected CPU usage as percent of capacity [STEP]
o target I/O usage as percent of capacity [STEP]
o actual I/O usage as percent of capacity [STEP]
o projected I/O usage as percent of capacity [STEP]
Target and actual CPU and I/O usage counts are useful in determining the degree CPU and I/O usage is approaching or exceeding the maximums specified in the requirements. As software modules are reused in new environments, it is necessary to assess the impact of the resource usage in the new environment. Ideally, projected CPU and I/O usage should be specified in the design phase, while upper bounds should be specified in the requirements.

3.4.4 Efficiency (of storage)

o duplicate global data definitions: (number of modules with duplicated data definitions / (total number of modules) [SAC]
This metric is used to measure the frequency in which global data items and constants (e.g., pi, acceleration of gravity) are defined more than once within a software system. Duplicate data definitions consume additional storage, so the greater the value of the measure, the lower the storage efficiency.

o duplicate code: (number of modules with duplicated code) / (total number of modules) [SAC]
This metric is used to measure the percentage of modules with duplicated code. Code for commonly-used functions (e.g., vector dot product or arithmetic mean) is often duplicated and consumes additional storage. The higher the value of the measure, the lower the storage efficiency.

- o software requirements allocation [SAC]
  This metric can be used indicate whether a storage (sizing) requirement allocation was performed during the system design phase to allocate overall sizing or storage utilization requirements to individual modules. A value of 1 means yes, 0 means no.

- o dynamic memory management [SAC]
  Generally, the use of dynamic memory management techniques (e.g., buffer memory allocation and release as necessary) promotes efficient utilization of storage. A value of 1 means yes, 0 means no.

- o storage optimizer [SAC]
  This metric can be used to indicate whether a storage optimizing compiler or assembler is being used. A value of 1 means yes, 0 means no.

- o target upper bound storage usage [STEP]
- o actual storage usage [STEP]
- o percent actual of target upper bound storage usage [STEP]
- o projected storage usage [STEP]
- o target upper bound RAM usage [STEP]
- o actual RAM usage [STEP]
- o percent actual of target upper bound RAM usage [STEP]
- o projected RAM usage [STEP]
  The target and actual storage and RAM usage measures are useful in determining how well software components "fit" into the allocations documented in the requirements. Storage counts the use of disk space and other mass storage, while RAM counts the use of Random Access Memory. Project usage counts are useful during development to scale the usage counts to the full system based on partial system measurements. Ideally, projected CPU and I/O usage should be specified in the design phase, while upper bounds should be specified in the requirements.

3.4.5 Adaptability

- o expandability
  - processing independent of storage: (number of modules whose size constraints are hard-coded) / (total number of modules with such size constraints) [SAC]
    The processing performed by a module should be independent of storage size, buffer space, array sizes, etc. Provisions for these entities should be provided dynamically, e.g., array sizes passed as parameters.

  - percentage of uncommitted memory: (amount of uncommitted memory) / (total memory available) [SAC]

  - percentage of uncommitted processing capacity: (amount of uncommitted processing capacity) / (total processing capacity available) [SAC]

3.4.6 Generality

- multiple usage metric: (number of modules referenced by more than one module) / (total number of modules) [SAC]
  A module is more general if it is referenced by more than one module, so the larger the value of this metric, the greater the generality.

- mixed function metric: (number of modules that mix functions) / (total number of modules) [SAC]
  A module that performs input/output as well as processing is not as general as one which only performs I/O or only performs processing. The lower the value of this metric, the greater the generality.

- data volume metric: (number of modules that are data volume limited) / (total number of modules) [SAC]
  A module that is designed to process only a certain number of data item inputs is not as general as one that can accept an unlimited number of inputs.

- data value metric: (number of modules that are data value limited) / (total number of modules) [SAC]
  A module that is designed to process only a limited range of data item values is not as general as one that is capable of processing a broader range of values.

- redefinition of constants metric: (number of constants that are redefined) / (total number of constants) [SAC]
  A module should not redefine a constant for the purpose of changing the function of the module, e.g., changing the base of a logarithm function from 10 to *e* for the purpose of providing a natural log function. Such items should be defined as parameters, not constants.

3.4.7 Maintainability

- complexity
  - decision count: count of **IF**, **DO**, **WHILE**, **CASE**, and other conditional and loop control statements [CONTE]
  - number of I/O variables per unit [AMI]
  - cyclomatic complexity [IEEE982.1] [CONTE] [MCCABE]
    Cyclomatic complexity may be used to determine the structural complexity of a code module. The cyclomatic complexity is calculated in a manner similar to the static complexity of the design. The difference is that the cyclomatic complexity is calculated from a flowgraph of the module, with an edge added from the exit node to the entry node.

Calculation of Cyclomatic Complexity:

$v = e - n + 2$

where:

v = complexity of the graph

e = number of edges (program flows between nodes)

n = number of nodes (sequential groups of program statements)

If a strongly connected graph is constructed (one in which there is an edge between the exit node and entry node), the calculation is [IEEE982.2]:

$v = e - n + 1$

The cyclomatic complexity is also equivalent to the number of splitting nodes (S) in the graph plus 1:

$v = S + 1$

(A splitting node is a node with more than one edge emanating from it.) Because each splitting node is associated with a condition, the expression $v = S + 1$ can be calculated by counting the number of conditions in the source code [MCCABE]. Cyclomatic complexity can also be calculated by counting the number of regions in the graph [IEEE982.2] [MCCABE].
The cyclomatic complexity for a multi-module program is sum of the v's for the individual modules [CONTE]:

$$v_{program} = \sum_{i=1}^{m} v_i$$

Alternatively, $v_{program}$ can be calculated as [CONTE]:

$$v_{program} = \sum_{i=1}^{m} DE_i + m$$

where DE_i is the decision count for the ith module, which is the same as the number of conditions.

- average nesting level [CONTE]

  The *nesting level* of a statement is defined by the location of the statement within control structures. Statements in the main flow of the module are at level one. Statements within loops, conditional clauses, etc. are at higher nesting levels. The *average nesting level* is calculated as follows: For each statement, determine the nesting level. Average nesting level is the sum of all the nesting levels divided by the total number of statements. A low average nesting level is an indicator of lower complexity in the logic of the module.

- executable lines of code per module [STEP]
- Software Science Metrics [IEEE982.1]

  Primitives:

  n_1 = number of unique operators
  n_2 = number of unique operands
  N_1 = total number of operators
  N_2 = total number of operands.

  Derived Metrics:

  program vocabulary: $l = n_1 + n_2$
  observed program length: $N = N_1 + N_2$
  estimated program length: $\tilde{N} = n_1(\log_2 n_1) + n_2(\log_2 n_2)$
  Jensen's estimator of program length: $N_F = \log_2 n_1! + \log_2 n_2!$.
  program volume: $V = L(\log_2 l)$
  program difficulty: $D = (n_1/2)(N_2/n_2)$
  program level: $L1 = 1/D$
  effort: $E = V / L1$

- Data or Information Flow Complexity [IEEE982.1] [CONTE]

  Primitives:

  lfi = local flows into a procedure
  datain = number of data structures the procedure accesses
  lfo = local flows from a procedure
  dataout = number of data structures that the procedure updates
  length = number of source statements in a procedure, excluding comments

  Derived Metrics:

  fanin = *lfi* + *datain*
  fanout = *lfo* + *dataout*
  Information Flow Complexity IFC = (*fanin* x *fanout*)^2
  Weighted IFC = *length* x (*fanin* x *fanout*)^2

- number of live variables [NIST500-209] [SQE] [CONTE]

  A variable is *live* from its first to its last reference within a procedure. The average number of live variables is calculated [CONTE]:

$$\overline{LV} = \frac{\sum_{i=1}^{n} lv_i}{n}$$

where:

lv_i is the count of live variables in the ith executable statement
n is the total number of executable statements

The average number of live variables for a program of *m* modules is [CONTE]:

$$\overline{LV}_{program} = \frac{\sum_{i=1}^{m} \overline{LV}_i}{m}$$

- variable spans [NIST500-209] [SQE] [CONTE]
  Variable span is the number of statements between two successive references to the same variable. For a program that references a variable in *n* statements, there are *n* - 1 spans for that variable. Average span size is calculated as the total of the span counts divided by the total number of spans. The average span size of a program of *n* spans is calculated [CONTE]:

$$\overline{SP}_{program} = \frac{\sum_{i=1}^{n} SP_i}{n}$$

- variable scope [NIST500-209] [SQE]
  Variable scope is the number of source statements between the first and last reference of a variable. With large scopes, the understandability and readability of the code is reduced.

o size metrics
  - lines of code - total lines of code including comments [KHOSH]
  - total number of code lines [CONTE]

o effort to fix bugs
  - number of errors to be corrected [AMI]
  - number of hours needed for correction [AMI]
  - number of units that were modified [AMI]
  - number of errors detected during system/integration tests [AMI]

3.4.8 Modularity

o cohesion [NIST500-209] [SQE] [SPC]
  See **Quality Metrics for Design Documentation** (section 3.3) for a description of cohesion.

o coupling [NIST500-209] [SQE] [SPC]
  See **Quality Metrics for Design Documentation** (section 3.3) for a description of coupling.

o number of entries/exits per module [IEEE982.1] [NIST500-209]
It is desirable to have one entry and one exit point per major function, with exceptions for error exits. Also, the number of functions per module should be limited; a suggested maximum number of functions is five per module [IEEE982.2]. See **Quality Metrics for Design Documentation** (section 3.3) for the calculation of measures for this metric.

3.4.9 Portability

o software independence
- number of operating systems software is compatible with [SAC]
- total number of system software utilities utilized [SAC]
  This metric can be used to measure the degree of dependence on system software utilities. The more usage is made of system software utilities, libraries, and operating system calls, the more dependent the system is on that particular software environment.
- common, standard subsets of language used: (number of modules utilizing non-standard constructs) / (total number of modules) [SAC]
  The usage of non-standard constructs or extensions of programming languages provided by particular compilers may impose difficulties in conversion of the system to new or upgraded software environments.

o hardware independence
- open systems [SAC]
  Are the programming languages and tools (e.g., compilers, database management systems, user interface shells) used by the implementation available on other machines? A value of 1 means yes, 0 means no.
- input/output references: (number of modules making I/O references) / (total number of modules) [SAC]
  Input/output references or calls are frequently a cause of machine dependence. Minimization and localization of these references facilitates machine independence and conversion from one machine to another. [SAC]
- word/character size: (number of modules not following convention / total number of modules) [SAC]
  Code that is dependent on machine word or character size should be avoided or parameterized to facilitate use on other machines.

3.4.10 Reliability

o reliability models
The *Jelinski-Moranda* reliability model attempts to predict the time of the next fault by assuming that fault times are independent random variables with exponential distributions. One criticism of the model is that it is assumed that each fault is removed instantaneously and with certainty. Another, more serious, criticism is that

the model assumes that all faults contribute equally to the unreliability of the program. The removal of a fault diminishes the rate of occurrence of failures by a fixed amount [LITTLEWOOD].

The *Littlewood-Verrall* model attempts to capture the uncertainty of the fixing operation; fixes are not certain to improve the reliability of the program. The rate of occurrence of failures is treated as a sequence of independent stochastically decreasing random variables. There is uncertainty about the magnitude of improvement of each fix [LITTLEWOOD].

- testability
  - number of independent paths [CALDIERA]
  - cyclomatic complexity [IEEE982.1] [CONTE] [MCCABE]
    See [MCCABE] for details on using cyclomatic complexity in determining the testability of software modules. McCabe's techniques can be used to assess the testability of software based on its complexity.

3.4.11 Understandability

- size metrics
  - lines of code [CONTE]
  - function points [NIST500-209]
  - function count [CONTE]

- traceability metrics
  - number of comment lines per total source lines of code [GPALS2]
  - percent comment lines of total lines [STEP]
  - correctness of comments

- complexity metrics
  - See complexity metrics defined above under **Maintainability**.
  - number of tokens [CONTE]
    The Halstead software science measure of observed program length can be used as a readability indicator. Observed program length $N = N_1 + N_2$, where N_1 is the total number of operators, and N_2 is the total number of operands. Halstead originally didn't count declaration statements, input/output statements, or statement labels. However, later researchers do count the operands in these types of statements.

- readability metrics
  - number of grammatically incorrect comments
  - number of misspellings
  - total number of characters [KHOSH]
  - total number of comments [KHOSH]
  - number of comment characters [KHOSH]
  - number of code characters [KHOSH]

3.5 Quality Metrics for Test Documentation

Software testing metrics are used to assess the adequacy of the test procedures and test data in verifying the software code. In order to gain confidence in a software component's reusability, a comprehensive set of test cases is necessary. A direct relationship between test cases and components is necessary in order for the component to be adequately tested in a new environment. Component test cases should be traceable to the components, and should be maintained as the components are changed. Also, component test cases should be delivered with the components.

System test cases should be linked to requirements specifications, and ideally, to the domain of interest. In order for system test cases to be reusable, there must be a tie-in to a specific requirements area, and therefore, a specific application domain. System test plans may be extracted from several subdomains and regrouped to test subdomains in a new domain area.

3.5.1 Completeness

- See [MCCABE] for details using cyclomatic complexity to "measure the completeness of the testing that a programmer must satisfy." Specifically, branch coverage and path coverage are verified for completeness. McCabe's technique can be used to develop a set of test cases which test every outcome of each decision, and execute a minimal number of distinct paths.

- test coverage [IEEE982.1]
  Functional (modular) test coverage index = $FE \,/\, FT$, where FE is number of software functions (modules) tested and FT is total number of software functions (modules).

- Test Sufficiency Indicator [AFSCP800-14] [IEEE982.1]
  This indicator is useful in assessing the sufficiency of software integration and system testing, based on the prediction of the remaining software faults. If fewer faults than expected are detected (outside the minimum tolerance limit), an adequate number of tests may not have been designed. See [AFSCP800-14] for a complete discussion of this indicator.
  Primitives:
  PF = total number of predicted faults in the software
  FP = number of faults detected before software integration testing
  UI = number of units integrated
  UT = total number of units in the Computer Software Configuration Item
  FD = total number of faults detected to date during test

Derived Metrics:

Remaining Faults $FR = (PF - FP) \times (UI/UT)$
Maximum Tolerance $MAXT = c_1 \times FR$
Minimum Tolerance $MINT = c_2 \times FR$
Percent of remaining faults to total predicted faults
where c_1 and c_2 are maximum and minimum tolerance coefficients

- coverage metrics
  - statement coverage: percentage of statements executed (to ensure that each statement has been tested at least once) [SQE]
  - branch coverage: percentage of branches executed [SQE]
  - path coverage: percentage of program paths executed, or the number of paths tested divided by total number of paths [SAC]
    It is generally impractical and inefficient to test all paths in a program. The number of paths may be reduced by treating all possible loop iterations as one path.
  - Test Coverage Indicator [AFSCP800-14] [IEEE982.1]

$$TC = \frac{\textit{Number of Implemented Capabilities Tested}}{\textit{Total Required Capabilities}} \times \frac{\textit{Software Structure Tested}}{\textit{Total Software Structure}} \times 100\%$$

- Data flow metrics [WEYUKER]

Data flow testing requires the selection of test data that exercise certain paths from a point in a program where a variable is defined, to points at which the variable definition is subsequently used.

- Categories for variable occurrences:
  - *Definition*: variable is given a new value
  - *P-use*: variable is used in predicate portion of a decision statement
  - *C-use*: all other variable uses, including variable occurrences in the right-hand side of an assignment statement, or an output statement
- Six data flow testing criterion are defined:
  - *all-definitions*: test data be included that causes the traversal of at least one subpath from each variable definition to some p-use or some c-use of that definition
  - *all-c-uses*: test data be included that causes the traversal of at least one path from each variable definition to every c-use of that definition
  - *all-p-uses*: test data be included that causes the traversal of at least one path from each variable definition to every p-use of that definition

- ■ *all-uses*: test data be included that causes the traversal of at least one subpath from each variable definition to every p-use and every c-use of that definition
- ■ *all-du-paths*: test data be included that causes the traversal of every simple subpath from each variable definition to every p-use and c-use of that definition

- Metrics:
  - ■ Percent of *all-definitions* covered by test scenarios
  - ■ Percent of *all-c-uses* covered by test-scenarios
  - ■ Percent of *all-p-uses* covered by test scenarios
  - ■ Percent of *all-uses* covered by test scenarios
  - ■ Percent of *all-du-paths* covered by test scenarios

- o percentage of defects uncovered in testing: (number of defects located by testing) / (total number of system defects) [PERRY]

3.5.2 Efficiency

- o execution time of test cases
- o (number of tests required) / (number of system errors) [PERRY]
  This metric shows the number of tests occurring per detected error. The smaller the ratio, the greater the test efficiency. This shows the efficiency of tests in uncovering errors.
- o (number of defects uncovered) / (size of system) [PERRY]
  This metric assumes there is a common number of defects in an application system based upon its size.

3.5.3 Understandability

- o size metrics
  - number of test steps
  - number of test cases
  - number of unique tests (test cases may cover more that one test)

4. SUMMARY

The initial focus of the Software Producibility MODIL was software reuse. When software is considered for reuse, especially in the high-integrity applications, the quality of the software is of paramount importance. While organizations must consider many variables to decide if software is fit for reuse, one important variable is the quality of the software. The term "quality" has many different meanings and even the characteristics commonly used to define quality have different meanings. Yet, organizations must be able to assess the quality of existing software in terms of its completeness for a new use, its correctness, its maintainability and other characteristics that impact how much work will be necessary to adapt the existing software for another application. Organizations who are developing new software that is intended for reuse must also be able to assess whether the software meets reusability criteria.

This report on quality characteristics and metrics for reusable software is preliminary. It identifies a set of quality characteristics that are most commonly referenced in technical literature and standards; these are completeness, correctness, generality, understandability, efficiency, modularity, portability, reliability, adaptability, and maintainability.

The metrics listed in this report help to define the quality attributes for software products. This report does not address any of the process metrics that should also be considered in assessing reusability. The products associated with metrics in this report are requirements documentation, design documentation, code listings and test documentation. Because the focus on quality for software in the past has been on code, there are many more code metrics listed in this report.

Different metrics for each product may be used to assess the degree to which the product contains each quality characteristic. This report provides some explanation of the value of each metric for determining the reusability of the software. However, more research is needed to ensure the completeness of the quality characteristics and associated metrics.

This preliminary study alone does not provide sufficient measurement information for determining the reusability of software. For example, the value of each characteristic and each metric for each product should be correlated to a reusability index. The reusability index, which no one yet has defined, should also include process metrics (e.g., effort). Most of the metrics have been used for several years and are well-understood relative to structured design methods but most have not yet been applied to software developed with object-oriented technology. Another important topic to be considered for reusing software in high integrity applications is the knowledge about the software relative to its specific domain and application within that domain and the type of information that must be present for analysts to decide how the software must change to meet new requirements.

Additional research issues for measuring the reusability of software include, but are not limited to, the following:

- process metrics relative to the effort needed to achieve the quality requirements of the reusable software
- algorithms for relating measures to a reusability index, for both product and process metrics
- criteria for selecting specific metrics
    - user of the data (program manager, technical staff)
    - purpose for which data will be used
    - constraints (e.g., language, development environment, products available)
    - value (e.g., quality characteristic most important)
- ability to collect data (e.g., automated support for specific measures)
- application of the quality characteristics and metrics to object-oriented development methods
- mapping of characteristics and metrics to existing and draft standards.

5. REFERENCES

[AFP800-48]
Air Force Pamphlet 800-48, "Acquisition Management: SOFTWARE MANAGEMENT INDICATORS," United States Air Force, June 1992.

[AFSCP800-14]
Air Force Systems Command Pamphlet 800-14, "Software Quality Indicators: Management Quality Insight," United States Air Force, January 1987.

[AMI]
"Metrics Users' Handbook: Quantifying Software Projects," Applications of Metrics in Industry (AMI) consortium, draft of version 2, January 1992.

[CALDIERA]
Caldiera, Gianluigi and Victor R. Basili, "Identifying and Qualifying Reusable Software Components," IEEE Computer, February, 1991.

[CARD]
Card, David N. and Robert L. Glass, Measuring Software Design Quality, Prentice Hall, 1990.

[CONTE]
Conte, S.D., H.E. Dunsmore and V.Y. Shen, Software Engineering Metrics and Models, The Benjamin/Cummings Publishing Company, 1986.

[FIPS106]
FIPS 106, "Guideline on Software Maintenance," U.S. Department of Commerce/National Bureau of Standards, June 1984.

[GPALS]
"GPALS Software Reuse Strategy," Department of Defense, Strategic Defense Initiative Organization, February 1992.

[GPALS2]
SDI-S-SD-91-000006, "Software Metrics Evaluation Plan (SMEP) for the Level System Simulator (L2SS)," Department of Defense, Strategic Defense Initiative Organization, January 1992.

[IEEEGLOSS]
ANSI/IEEE Std. 610.12, "IEEE Standard Glossary of Software Engineering Terminology," The Institute of Electrical and Electronics Engineers, February, 1991.

[IEEE982.1]
IEEE Std. 982.1-1988, "IEEE Standard Dictionary of Measures to Produce Reliable Software," The Institute of Electrical and Electronics Engineers, June, 1988.

[IEEE982.2]
IEEE Std. 982.2-1988, "IEEE Guide for the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software," The Institute of Electrical and Electronics Engineers, June, 1989.

[IEEE1045]
IEEE Std. 1045-1993, "Standard for Software Productivity Metrics," The Institute of Electrical and Electronics Engineers.

[ISO9126]
ISO/IEC 9126, "Information technology - Software product evaluation - Quality characteristics and guidelines for their use," International Organization for Standardization and International Electrotechnical Commission, December 1991.

[KHOSH]
Khoshgoftaar, Taghi M., John C. Munson, Bibhuti Bhattacharya and Gary D. Richardson, "Predictive Modeling Techniques of Software Quality from Software Measures," IEEE Transactions on Software Engineering, Vol. 18, No. 11, November 1992.

[LITTLEWOOD]
Littlewood, B., "How Good are Software Reliability Predictions?," Software Reliability Achievement and Assessment, Blackwell Scientific Publications, 1987.

[MCCABE]
McCabe, Thomas J., NBS Special Publication 500-99, "Structured Testing: A Software Testing Methodology Using the Cyclomatic Complexity Metric," U.S. Department of Commerce/National Institute of Standards and Technology, December 1982.

[MODIL]
Katz, Susan, Christopher Dabrowski and Margaret Law, "Glossary of Software Reuse Terms," prepared for The Department of Defense/Ballistic Missile Defense Organization, by the U.S. Department of Commerce/National Institute of Standards and Technology, October 1993.

[MURRAY]
Murray, Melba Jerry and Hugh Hay-Roe, Engineered Writing, Second Edition, PennWell Publishing Company, 1986.

[NIST500-209]
Peng, Wendy W. and Dolores R. Wallace, NIST Special Publication 500-209, "Software Error Analysis," U.S. Department of Commerce/National Institute of Standards and Technology, April 1993.

[NISTIR4909]
Wallace, Dolores R., Wendy W. Peng and Laura M. Ippolito, NIST IR 4909, "Software Quality Assurance: Documentation and Reviews," U.S. Department of Commerce/National Institute of Standards and Technology, September 1992.

[NISTIR5309]
Dabrowski, Christopher and Susan B. Katz, NIST IR 5309, "A Context Analysis of the Network Management Domain," U.S. Department of Commerce/National Institute of Standards and Technology, December 1993.

[NSWC]
Technical Report SRC-88-011, "Management Indicators: Assessing Product Reliability and Maintainability," Naval Surface Warfare Center, August 1988.

[NSWC2]
Technical Report SRC-88-008, "A Taxonomy for the Evaluation of Computer Documentation," Naval Surface Warfare Center, June, 1988.

[PERRY]
Perry, William E., "Measuring the Effectiveness of Testing," Quality Assurance Institute, 1984.

[SAC]
"Software Reuse Metrics Phase I Metrics Model," prepared by Space Applications Corporation, January 8, 1993.

[SAC2]
"Software Reengineering and Software Reuse Technologies," prepared by Space Applications Corporation, March, 1993.

[SEI]
Park, Robert E., "Software Size Measurement: A Framework for Counting Source Statements," Technical Report CMU/SEI-92-TR-20, Software Engineering Institute, September 1992.

[SOFTECH]
Vitaletti, Bill and Ravinn Chhut, "Maximizing Software Reuse, A Comprehensive Reuse Environment," SofTech, Inc., Prepared for the National Institute of Standards and Technology, June 10, 1992.

[SPC]
SPC-91060-CMC, Software Measurement Guidebook, Software Productivity Consortium, December 1992.

[SQE]
"Software Measurement," Seminar Notebook, Version 1.2, Software Quality Engineering, 1991.

[STARS]
"Informal Technical Report for the Software Technology For Adaptable, Reliable Systems (STARS)," prepared by The Boeing Company Defense & Space Group, IBM Federal Sector Division, and Paramax Systems Corporation Tactical Systems Division, February 14, 1992.

[STEP]
Betz, Henry P. and Patrick J. O'Neill, "Software Metrics Initiatives Report," Army Software Test and Evaluation Panel (STEP), March 21, 1991.

[SYMONS]
Symons, Charles R., Software Sizing and Estimating, John Wiley & Sons, 1991.

[WEYUKER]
Weyuker, Elaine J., "More Experience with Data Flow Testing," IEEE Transactions on Software Engineering, Vol. 19, no. 9, September 1992.

[ZAGE]
Zage, Wayne M. and Dolores M. Zage, "Evaluating Design Metrics on Large-Scale Software," IEEE Software, Volume 10, No. 4, July 1993.