Software Quality Measurement: Validation of a Foundational Approach

By James D. Arthur, Richard E. Nance, Gary N. Bundy, Edward V. Dorsey, and Joel Henry

TR 91-16*

*Cross-referenced from Technical Report SRC-91-002, published May 1, 1991, by the Systems Research Center and Department of Computer Science at Virginia Polytechnic Institute and State University, Blacksburg, Virginia 24061.

Work supported by the Joint Logistics Commanders through the Systems Research Center under contract number N60921-89-D-A239 B004.

ABSTRACT

This report discusses the first year findings of a proposed three year investigation effort that focuses on the assessment and prediction of software quality. The research exploits fundamental linkages among software engineering Objectives, Principles and Attributes (the OPA framework). Process, code and document quality indicators are presented relative to the OPA framework, with elaboration on their individual roles in assessing and predicting software quality. The synthesis of an Ada code analyzer is discussed as well as proposed complementary tools comprising an automated data collection and report generation system.

Key Words and Phrases: Simulation quality assessment and prediction; software engineering objectives, principles and attributes; software quality indicators, process indicators, code indicators, document quality indicators.

Table of Contents

1. Introduction

The critical nature and long life expectancy of today's complex software systems mandate the production of quality software products [PARD85]. Past and current experiences have shown that product quality cannot be retrofitted, but instead, must be built into a product from the beginning. Perhaps one manifestation of failure to instill quality has been the substantial cost of maintaining a product after deployment. Estimates of the proportionate cost after development range from 50% to 80% [HALD88].

Insuring that quality is built into a product, however, raises the following question: *How does one measure product quality, beginning with requirements specification, and continuing through deployment?* Current attempts to assess software quality through the use of product metrics alone have met with significant criticism. Such efforts have been described as being narrowly focused and providing measures that are often based on questionable metrics [KEAJ86]. Research efforts that address software quality assessment primarily from the process perspective are suspect too. For example, the Assessment of Contractor Capability entails an examination of the development process to assess the maturity level [HUMW87]. Certainly such an assessment can be a significant aid to improving the process, but maturity in definition does not guarantee effectiveness in execution, to produce the consequent high quality product.

Insurance that quality is being built into a product requires (1) the recognition that the software development process is governed by specific principles, (2) those principles are supportive of clearly defined project objectives, and (3) those executing the process are doing so with the knowledge and understanding to reflect detectable attributes in both the development process and consequent product. This is the basic premise that has guided a software quality assessment approach, termed the Objectives/Principles/Attributes (OPA) framework. In several papers and presentations over the past six years the authors have contended that the OPA framework represents

- a tractable approach for assessing and predicting product quality through the use of measures that reflect the quality of both the software development process and the products of the process, and
- an approach that admits to scientific validation through statistical analysis and unit comparisons employing predicted and observed quality rankings.

In support of the approach stated above, the authors have proposed a four-year research investigation that focuses on establishing a validated procedure for assessing software quality. Within the OPA framework, steps toward realizing a software quality assessment procedure are [ARTJ90]:

(1) the formal definition of software quality indicators,

(2) the development of an automated analysis tool,

(3) the identification of a software development project suitable for and amenable to the collection of data for computing software quality indicators,

(4) on-site data collection, experimentation, and refinement of the software quality indicators, and instrumentation of the development process utilizing a test set of software components, and

(5) validation of the assessment and predictive capabilities through a statistical analysis, using both the test set and an experimental set.

Year one of the proposed four year effort focuses on the first three steps outlined above. The remainder of this report discusses the findings relative to those three steps, followed by a brief description of the major research directions for the current (second) year investigation.

2. The Task Statement : Year One

The set of subtasks presented below outline the separate research directions for year one of the Software Quality Measurement project. They also reflect, in order, the subjects addressed in the following major sections.

Subtask 1: Refine and Extend the Indicator Definitions: Review the design indicators for document assessment capabilities and extend as necessary to reflect the standards and guidelines for the anticipated validation project. Modify and extend design indicators to reflect the specific capabilities of Ada.

Subtask 2: Modify and Extend the Automated Analyzer: Augment and extend the internal analysis procedures of the analyzer to reflect the completed software quality indicators - exploiting both design and management indicators. Extend the report generator procedure (back end) of the analyzer to accommodate the software engineering objectives of maintainability and reliability. Design, code and test the language analysis procedures specific to Ada (an Ada implementation is intended). System test the entire analyzer.

Subtask 3: Select the Validation Project: Develop a set of selection criteria for project/site qualification. Investigate candidates with project sponsor and mutually agree on at least one, with a second (backup) only if funds permit.

Each of these subtasks are individually addressed in the following sections.

3. Refine and Extend the Indicator Definition

An indicator is a directly measurable surrogate for a concept that is not directly measurable. For example, the concept of "cultural opportunity" cannot be measured directly, but indicators such as the number of galleries, museums or exhibits per square mile can be measured and reflect the extent to which cultural activities are supported, in a relative if not an absolute sense.* Most important is the validity of indicators; that is, an indicator measure should be undeniably linked to the concept that it purports to measure, and the measure should necessarily and sufficiently capture the intended characteristics of the concept.

From the perspective of software quality, the authors desire indicators that allow one to assess current product quality and to predict the quality of a product "down stream". The OPA framework suggests that such indicators be derived from observable properties of (a) the software development process, (b) supporting documentation, and (c) code. The research effort for this first year has focused on completing the definition of code indicators and on making significant progress toward the formulation of documentation and process-related indicators.

* The number of display activities (galleries, museums, exhibits) per square mile might provide a comparative indicator of which of two communities offers more cultural opportunities of this type.

3.1 Process Indicators

To create a quality product, the appropriate steps must be performed (or processed) effectively throughout the software development. That processing is accomplished through the proper use of defined sets of tools, methods, and practices that together produce a deliverable software product [HUMW89]. To meet quality goals the development process must be instrumented at selected points to provide measures that convey the appropriateness and effectiveness of those tools, methods, and practices, thereby providing an indication of the quality being achieved in the deliverable product. Process indicators:

- assimilate such measures to predict the presence or absence of quality in the final product,
- gauge the level of quality being instilled by the development process, and
- provide predictive indications of product quality, e.g., faulty or missing activities in the development process.

3.1.1 Process Indicator: Background and Development

Process activity measures play a crucial role in the identification and formulation of process indicators. Initial research efforts focusing on activity measures is described by Rosson [ROSC88], who advances measures supporting the definition of management indicators. Rosson's management indicators are associated with activities like design reviews, code inspections, as well as characteristics reflecting operational behavior and maintenance activities. Rosson's work, however, concentrates on measures obtained at acquisition time and during deployment. Process indicators, on the other hand, employ data sampling throughout the software development process and provide constant feedback as to the quality of the product being produced. Viewing the similarities and differences between management indicators and process indicators provides a consistent, well-defined approach for identifying and defining process indicators.

The initial step in formulating process indicators and relating them to quality measurement and prediction has been to identify salient characteristics particular to all such indicators. In short process indicators:

- need not satisfy a rigorous mathematical proof, rather, the values from the indicator metrics should provide evidence of the presence or absence of quality in the final product,
- can represent metric values extracted or derived from data items, and
- must have an undeniable and natural linkage to an objective, principle, or attribute defined by the OPA framework.

Recognizing the necessity of such characteristics, the formal identification and definition of process indicators are achieved through a sequence of steps:

(1) identifying specific activities within the development process,

(2) recognizing the intended impact of each activity on the software product under construction,

(3) relating each activity to a specific entity in the OPA framework,

(4) proposing a measurement approach for assessing the presence or absence of quality imparted on the product by each activity,

(5) proposing and refining a metric for each process indicator, and

(6) identifying process instrumentation points for capturing the necessary metric data.

Preliminary application of this sequence to development activities confirms the feasibility of the procedure. Care must be taken, however, when applying step (3) to insure that a natural, intuitive linkage to the OPA framework is established.

3.1.2 Research Status

Research focusing on process activities, and in particular process indicators, is a new and relatively recent endeavor within software metrics. In such an undeveloped research endeavor, unforeseen yet critical questions can arise, demanding answers before additional "mainstream" progress can be made. The investigation of process indicators is no exception. As described below, the formal definition of process indicators requires a framework for relating process activities and perceived process indicators to the software development life-cycle model.

The Process Indicator Life-cycle Relationship Model

Early in the research effort, the authors realized that progress was being impeded because several important issues remained unresolved. These issues can be stated as follows:

- How do process indicators fit into the waterfall life-cycle model of software development?
- What is the relationship between process indicators obtained during different phases of software development?
- How does one over time integrate the refinement of process indicators into the prediction process?

Clearly, a framework relating process indicators within and among software development life-cycle phases is needed to resolve these issues. Such a model is proposed and discussed below.

The life-cycle model shown in Figure 1 relates process activities and corresponding quality measures across the software development life-cycle. In particular, the authors draw attention to the boxes containing various subscripted Fs. These boxes represent families of process indicators applicable within and among phases of the development process. The notation in Figure 1 is as follows:

- F_R - intraphase requirements quality functions,
- F_D - intraphase design quality functions,
- F_C - intraphase code quality functions,
- F_{RD} - interphase requirements/design function,
- F_{DC} - interphase design/code quality functions, and
- F_{Del} - process activity functions spanning the entire process.

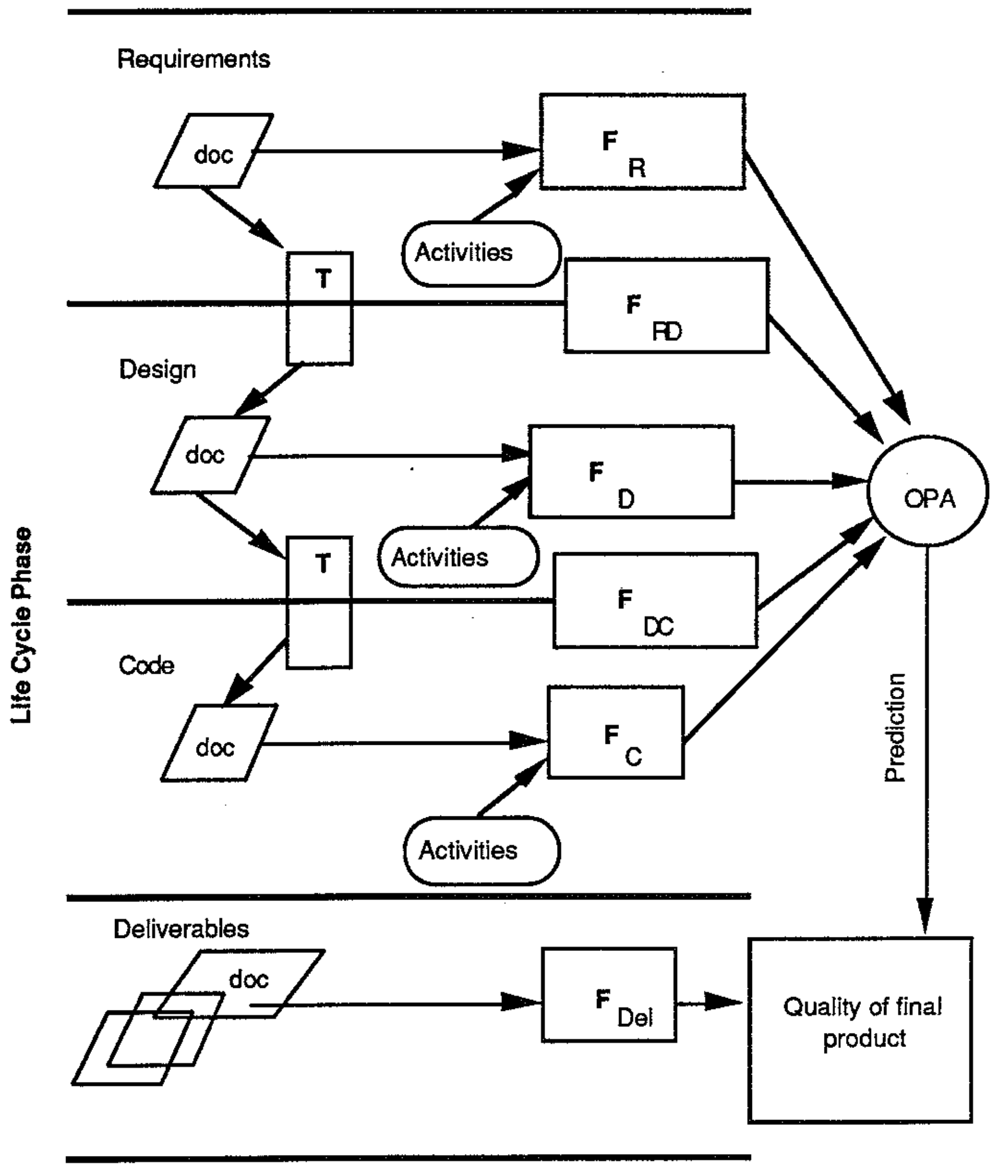


Figure 1:
Process Indicator Life-Cycle Relationship Model

For each family instance, metric values computed for indicators are linked to the OPA framework. The OPA framework provides an aggregation mechanism for all measures within each family of functions. In turn, the aggregated values provide a prediction of quality of the final product.

The domain of each indicator within an intraphase family of indicators (e.g., F_R, F_D, and F_C) is a process activity or an intermediate product of the development effort. These indicators have a common range, i.e., objectives, principles, or attributes defined by the OPA framework. The domain of each indicator within the interphase family of functions (e.g., F_{RD} and F_{DC}) is an activity (or collection of activities) denoted by a transition function T. Effectively, the interphase functions provide the capability to "refine" predictive calculations from a preceding phase by using data elements from past and current activities.

In the aggregation process the most recently measured indicators have a more significant impact on quality assessment than those obtained earlier in the life-cycle. That is, the impact of phase dependent process indicators on the prediction of product quality diminishes as the development effort proceeds. Aggregation supports a time-sensitive view of the product quality effected in the development process.

Because the proposed framework employs the conventional waterfall model of software development as its basis, it easily supports the aggregation of process indicators by life-cycle phases. Moreover, because the framework distinguishes between inter- and intra-phase, indicators can be developed (a) within each family of functions without concern for the impact on other life-cycle phases and (b) independently of previously developed indicators.

Process Indicator Development

Our investigation of process indicators begins with the consideration of activities in the requirements phase of the development life-cycle. Many of these activities focus on identifying and succinctly stating the desired set of requirements. Consequently, obtaining meaningful measures in the early stages of the requirements specification process is difficult. Recognizing this fact, the authors have concentrated the investigation on identifying indicators utilizing data generated after the software specification review, i.e. after the requirements are fixed. Thus far three prospective indicators are identified in the requirements phase, two of which cannot be measured until after the requirements are fixed:

- Requirements Defects related to the attribute of Early Error Detection,
- Added Software Functionality related to the principle of Decomposition, and

- Requirements Testability related to the attribute of Early Error Detection.

Initial research has also led to the identification of process indicators that span multiple phases of the development process, e.g.,

- The creation of Software Development Folders (SDFs) related to the attribute of Traceability, and
- The updating of SDFs related to the attribute of Visibility of Behavior.

In addition to process indicators reflective of software development activities, the authors note other indicators claimed by others to impact product quality. In particular, Boehm suggests that development personnel experience is an important factor in the quality of software [BOEB81]. Similarly, Abdel-Hamid [ABDT89] describes personnel turnover as a major factor affecting software quality. Following these observations, two prospective indicators in the personnel area are currently under investigation:

- Personnel Experience and its impact on Visibility of Behavior, and
- Personnel Turnover pertaining to the introduction and detection of errors, i.e., Early Error Detection.

Activities within the design phase are also being examined for characteristics that might suggest other indicators. For example, the thoroughness of the design review process significantly impacts early error detection. Activities related to defect detection, system and specification changes, and the work-force organization are among other factors being considered.

Appendix A provides additional information on prospective indicators. A more detailed description of the indicators are given, together with an explanation of their importance and approaches to measurement.

Data Acquisition

Recognizing that proper data acquisition methods are crucial to the validation effort, the authors have initiated a search for published articles describing data collection procedures, techniques, tools, and experiences. This review is important for several reasons:

- procedural errors reported by other researchers should be avoided,
- data must be properly validated to assure understanding and accuracy, and
- the data acquisition procedures must be defined to assure the integrity of the data.

Basili [BASV84] and Ross [ROSN90] provide excellent suggestions and insights relative to all three reasons described above.

3.1.3 Future Research

The Process Indicator Life-Cycle Model provides one framework for describing the relationship among process indicators throughout the software development life-cycle. The identification and definition of indicators applicable to the requirements specification and design phases have led to a better understanding of their relationships to process activities and to the OPA framework. Much work still remains, however, to understand and exploit process indicators for assessing and predicting product quality. Future research directions are outlined below.

Process Indicators: Clearly, the continued identification and formulation of process indicators within and across life-cycle phases is needed. The immediate task is indicator development focused on the high- and low-level design phases of the development process. These indicators are expected to be slightly different from those developed thus far. In particular, two indicators associated with requirements phase activities rely on downstream data for measuring software quality. Indicators related to design activities and process trends, are expected to use data acquired during the design phase, and thereby permit better software quality prediction immediately following an activity.

Metric Data Acquisition: The acquisition of pertinent data at the selected validation site is also a primary concern. An on-site examination of the existing configuration management system, software development process and personnel organization is necessary to determine when and where data should be acquired. Techniques required to validate metric data must also be determined. Basili and Weiss [BASV84] outline basic procedures for achieving such goals.

Data Recording: Finally, an automated system for recording data must be addressed. Previous metric validation projects warn against the recording of data in a prose-based format (as opposed to a forms-based) because the inherent ambiguity of the English language can lead to misunderstanding between the writer and the reader. Moreover, the probable delay in formally generating the paper report can lead to inaccurate recording. An automated recording system also provides a more reliable method for analyzing process data.

3.2 Documentation Quality Indicators

Software development documentation can be divided into three temporal components: requirements specifications, design specifications, and code. Components in each phase are refinements of their predecessors (e.g., the high-level design is a more detailed version of the requirements specifications). The program (code) is an executable project specification. Subsequently, one can infer that documentation is the blueprint from which a software product is built. Document characteristics such as accuracy, completeness, and usability are crucial, especially when one considers that 50 to 70 percent of the total software life-cycle costs are attributed to maintenance activities [HALD88]. Unlike hardware and software products, technical documentation does not admit so readily to methods and criteria for evaluating its adequacy.

The Objectives/Principles/Attributes (OPA) [ARTJ86] framework provides a basis for assessing documentation quality relative to accuracy, completeness and usability. The following sections describe an approach consistent with that framework.

3.2.1 Document Quality Indicator: Background and Development

Document quality indicators (DQIs) are employed to assess the adequacy of software documentation. A DQI is triple whose principal components are: a quality, a factor of quality, and a quantifier. These three components form a *hierarchical* relationship that provides successive levels of refinement, attaching specific meaning and significance to each individual indicator. Appendix B illustrates the hierarchical relationship among qualities, factors of qualities, and factor quantifiers. Qualities are at the root of each tree, quantifiers are the leaves, and factors are the intermediate elements refining the qualities.

Documentation Qualities: At the topmost level of the DQI hierarchy are the *qualities* of adequate documentation. These qualities are abstract by nature and serve to convey the broader meaning of adequate documentation. Currently four qualities are deemed necessary for adequate documentation [STEK88]:

(1) Accuracy - the consistency among code and all documentation components for all requirements,

(2) Completeness - the existence of all documents required by a set of standards, and the presence of all required components for each of these documents,

(3) Usability - the suitability of the documentation in terms of the ease of extracting needed information, and

(4) Expandability - the capability of the documentation to be modified in reaction to changes in the system (ease of modification).

The efforts to this point address only the first three; no plans exist to include expandability.

Factors of Qualities: The intermediate levels of the DQI hierarchy represent *factors* of qualities. These factors are a more specific qualification of the qualities, supplying refining details for the abstractions identified at the quality level. Factors serve to decompose the qualities into smaller, more comprehensible units.

Quantifiers: The leaves of the DQI hierarchy represent the *quantifiers* of adequate documentation. Quantifiers convey the measurable characteristics of the documentation. This level of the DQI hierarchy is the basis from which assessment criteria are formed.

Separately, the qualities, factors, and quantifiers have individual and diverse meanings associated with them. Considered as quality-factor-quantifier triples however, they form distinct *indicators* of document quality. Viewed from a top-down perspective, qualities are more abstract than either factors or quantifiers, but tend to convey the more high-level characteristics deemed desirable in software documentation. Measuring the extent to which such qualities are achieved, however, requires the introduction of factors that decompose and refine qualities into less abstract, specific, and manageable units. Nonetheless, similar to qualities, factors are not readily amenable to direct measurement. Hence, factors too are further refined through the introduction of quantifiers, which *are* directly measurable.

3.2.2 Research Status

Because "adequate" documentation has many facets, the development of an effective procedure for assessing documentation quality is necessarily an evolutionary process, with each enhancement building from and contributing to understanding this difficult problem. What has evolved is a three phase process for developing an automated approach for assessing the adequacy of documentation, while maintaining consistency within the OPA framework. Those phases are described below with references to appendices where appropriate.

Phase 1: Identifying and defining the documentation quality indicators (DQIs).

Using the work of Stevens as a basis [STEK88], the current thrust of Phase 1 is to examine and completely define each proposed DQI therein and to develop new DQIs when appropriate. A "completely defined" DQI is one for which the following steps have been completed:

(1) Identification and Definition of the DQI. Individual characteristics of documentation that contribute to documentation quality are identified and related to specific DQIs. As discussed later, each DQI takes the form of a quality-factor-quantifier triple.

(2) Determination of Rationale for Inclusion. A justification statement is developed, outlining specific reason(s) why the DQI is important to documentation quality, and thereby warranting further consideration.

(3) Identification of Measurement Approach. The approach to measuring the relationship between a DQI and document quality is identified along with the data elements needed to perform the measurement.

(4) Definition of Appropriate Metrics. Using data elements specified by the measurement approach, a numerically-valued expression (or *metric*) is defined, reflecting an assessment based on the rationale for inclusion.

Tables 1a and 1b provide a list of DQIs relative to the documentation quality to which they are related and illustrate the extent to which each definition is complete. For example, Table 1a lists *Percentage of Missing References* as an indicator of Documentation Completeness. Table 1a also indicates that all four of the above mentioned steps have been completed for this particular DQI.

Several DQIs listed in the Tables have a "contingency footnote" associated with them to indicate that the current state of development is contingent on the resolution of a specific condition. Additionally, the "automation feasibility" column contains a rating reflecting the anticipated level of difficulty in automating the evaluation of the corresponding DQI. Although yet to be completed, a scale similar to the following is anticipated:

- F = Fully automatable,
- S = Semi-automatable,

- NC = Not currently automatable, and
- U = Automatability undeterminable at the current time.

In determining the scale value for each DQI, the measurement approach is examined to identify the required primitive data items. The feasibility of automatically extracting each item is examined and then, based on those findings, an automatability rating is determined. Assignment of automation feasibility ratings is currently being performed.

In summary, as indicated by Tables 1a and 1b, 80 percent of the DQIs are completely defined; the remaining DQIs are in various stages of refinement. Appendix C provides an expanded discussion of each individual DQI, including the measurement approach and the rationale for each approach.

Phase 2: Integrating the DQIs into the OPA Framework.

The primary task in Phase 2 is to identify the relationship between the DQI hierarchy and the OPA (Objectives/Principles/Attributes) framework for software quality assessment. Currently, the *qualities* of adequate documentation are being considered analogous to the *attributes* defined in the OPA framework. Identifying *principles* guiding the development of quality documentation still remains a goal of this research effort. Several principles defined by the current OPA framework, however, appear applicable to documentation, e.g. Concurrent Documentation and Abstraction. Nonetheless, they too need to be re-examined relative to their explicit contribution to the realization of quality documentation. The task of integrating DQIs within the OPA framework is currently being pursued.

Phase 3: Developing an automated system for assessing the DQIs.

Once the DQIs are defined, steps to implement an automatic documentation assessment procedure can begin. From a pragmatic perspective, the measurement of several DQIs relies on project specific details, usually related to

Table 1a: Developmental Status of Documentation Quality Indicators

Documentation Quality	Contingenicies	Documentation Quality Indicator	Undefined	Measure Defined	Rationale Complete	Measurement Approach in Progress	Measurement Approach Complete	Metric Development in Progress	Metric Complete		Automation Feasibility
ACCURACY	1	Requirements Supported by Design	▨	▨	▨	▨	▨	▨	▨		
	1	Design Supported by Code	▨	▨	▨	▨	▨	▨	▨		
		Design Utility	▨	▨	▨	▨	▨	▨	▨		
		Code Utilization	▨	▨	▨	▨	▨	▨	▨		
	2	Factual Consistency	▨	▨	▨	▨	▨	▨	▨		
		Invarience of Concept	▨	▨	▨	▨					
COMPLETENESS		Percentage Domain Coverage	▨	▨	▨	▨	▨	▨	▨		
		Adherence to Standards	▨	▨	▨	▨	▨	▨	▨		
		Refinement Enunciation	▨								
		TBD / TBS	▨	▨	▨	▨	▨				
		Percentage Missing References	▨	▨	▨	▨	▨	▨	▨		
		Percentage Appropriate References	▨	▨	▨	▨	▨	▨	▨		

Contingenicies

1 The correspondence between life-cycle stage and the metric parameter j has yet to be defined.

2 Two possible metrics exist for this indicator, involving computation (a) per section or (b) per fact. A decision needs to be made regarding which metric is more appropriate.

Table 1b: Developmental Status of Documentation Quality Indicators

Documentation Quality	Contingenicies	Documentation Quality Indicator	Undefined	Measure Defined	Rationale Complete	Measurement Approach in Progress	Measurement Approach Complete	Metric Development in Progress	Metric Complete		Automation Feasibility
USABILITY		Keyword Consistency	▨	▨	▨	▨	▨	▨	▨		
		Acronym Usage	▨	▨	▨	▨	▨	▨	▨		
		Abbreviation Usage	▨	▨	▨	▨	▨	▨	▨		
		Completeness of TOC	▨	▨	▨	▨	▨	▨	▨		
		Correctness of TOC	▨	▨	▨	▨	▨	▨	▨		
		Format of TOC	▨	▨	▨	▨	▨	▨	▨		
		Bottom-Up Traceability	▨	▨	▨	▨	▨	▨	▨		
		Top-Down Traceability	▨	▨	▨	▨	▨	▨	▨		
		Completeness of Index	▨	▨	▨	▨	▨	▨	▨		
		Correctness of Index	▨	▨	▨	▨	▨	▨	▨		
		Format of Index	▨	▨	▨	▨	▨	▨	▨		
		% Domain Coverage (intra-doc)	▨								
		Text Conciseness	▨	▨	▨	▨	▨	▨	▨		
		Formulated Readability	▨	▨	▨	▨	▨	▨	▨		
		Term Uniqueness	▨	▨	▨	▨	▨	▨	▨		
		Glossary Completeness	▨	▨	▨	▨	▨	▨	▨		
		Order of Glossary	▨	▨	▨	▨	▨	▨	▨		
		Adherence to Standards	▨	▨	▨	▨	▨	▨	▨		
		Simplicity / Modularity	▨	▨	▨	▨	▨				
		Redundancy Appropriateness	▨	▨	▨						
		Adequacy of Print	▨								
		Format Appropriateness	▨	▨	▨	▨					
		Format Consistency	▨	▨	▨	▨	▨	▨	▨	▨	
		Module Appropriateness	▨								

documentation format requirements. Whenever such cases arise, the formulation of a metric follows the assumption that the documentation products specified in DoD-STD-2167A apply. The authors note, however, that the theory and rationale underlying the DQIs is to be kept as *domain independent* as possible.

Research by other professionals supports the feasibility of the assumption that systems exist which permit automatic documentation assessment through currently proposed DQIs. For example, the assessment of formulated readability (as it relates to usability) has been studied by [FRYE68], [MCCG87], [VERC80], [BARJ80], and [DRUA 85]. Additionally, assessing DQIs that require keyword lists is discussed in [CHUK90] and [SALG89]. Based on preliminary work, the authors are confident that an automated assessment procedure is both feasible and tractable.

3.2.3 Future Goals

The current set of DQIs is based on measures that utilize data primitives directly related to the targeted concepts to measure. In addressing the pragmatic issue of computing effective DQIs, however, the authors recognize that some data primitives are difficult (if not impossible) to collect. Current research efforts focus on reassessing DQIs relative to approximation measures, i.e., formulations that approximate the originally defined metric or measurement approach. The authors do not anticipate extensive use of approximation measures, and perceive a clear advantage over the alternative of ignoring the DQI completely.

The timetable for the next six months of effort is shown in Figure 2. The definition of the DQIs (Phase 1), should be complete by the end of August 1991. Implementation of an automated system (Phase 3) is currently targeted for a late November completion. The integration of DQIs into an OPA framework is an ongoing effort and is tentatively targeted for a June completion.

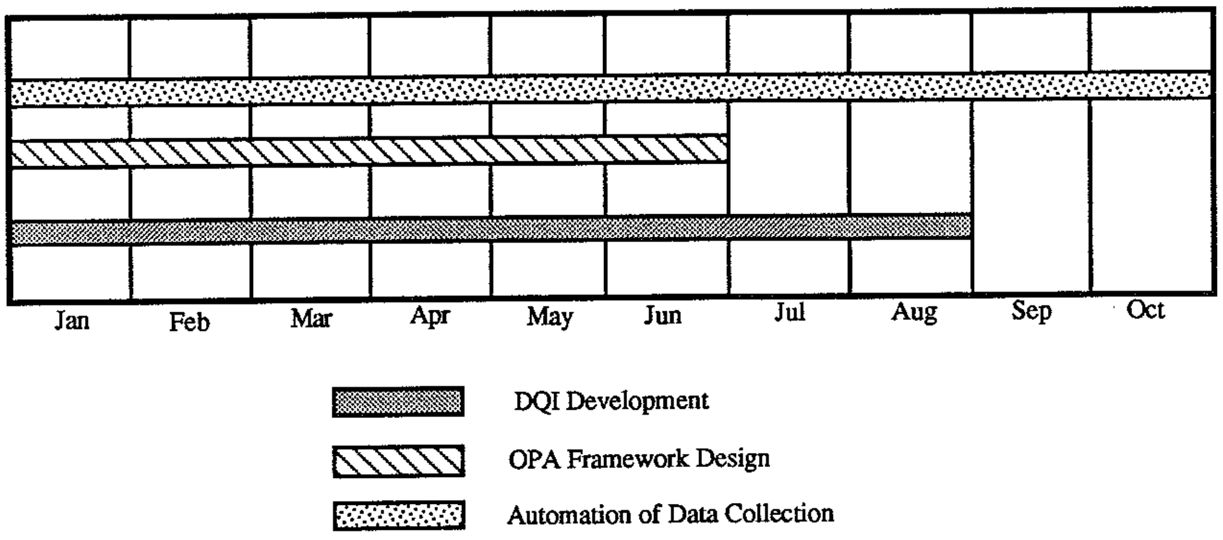


Figure 2:
Documentation Quality Assessment Timetable

3.3 Code Indicators

Developing measures of software (code) quality has been a continuous challenge in computer science and software engineering. A literature survey of metrics reveals that many metrics are available for measuring software. Some well documented metrics include Halstead's Software Science [HALM77], McCabe's Cyclomatic Number [MCCT76], and Henry and Kafura's Information Flow [HENS81]. A major criticism of many of these metrics is the lack of a "clear specification of what is being measured" [KEAJ86]. Another author notes that a desirable attribute, missing from most metrics, is that software metrics should "empirically and intuitively describe software behavior" [EJIL87].

A first step in addressing such criticisms can be found in Arthur and Nance's Objectives, Principles, Attributes (OPA) framework for software quality assessment [ARTJ87]. The OPA framework defines a set of linkages that relate the achievement of software engineering objectives to the use of principles, and the use of principles to the presence or absence of desirable attributes. This section describes the results of applying the OPA framework to Ada and the subsequent derivation of metrics that support a quality assessment of Ada code.

Similar to the DQI development, the synthesis of code indicators also follows a well-defined set of steps:

- identifying, classifying and categorizing Ada constructs,
- understanding the rationale for including specific language components in the definition of Ada,
- assessing the importance of each language component from a software engineering perspective,
- identifying the impact of component usage on desirable software engineering attributes, and finally
- identifying code properties and attribute pairs that serve as indicators, and formalizing metrics based on measurement approaches that reflect individual property/attribute relationships.

The research presented below describes the first step relative to the complete Ada language. For purposes of illustration and brevity, the remaining four steps are applied to one significant Ada component: the package.

3.3.1 Background and Research Results

The five steps outlined above provide a basis for tailoring the OPA framework to reflect characteristics particular to specific programming languages. Application of all five steps is completed for an Ada-based effort. The following sections individually discuss each of those steps, starting with the identification of critical Ada language components and ending with a discussion of the metrics directly related to the software engineering attributes defined by the OPA framework.

Step 1: *Identifying, Categorizing and Classifying Crucial Language Components*

The first step in defining an OPA-based procedure for assessing the quality of an Ada-based product is to identify those language components deemed necessary and crucial to the assessment process. Such first steps often involve a categorization scheme that permits a language to be analyzed at the individual component level and then to be viewed from analytical perspectives based on aggregated components. In concert with this approach, the initial categorization scheme employs partitioning criteria proposed by Ghezzi and Jazayeri [GHEC82]; that is, the partitioning of language components along specific functional boundaries. In particular, an Ada program can be viewed as possessing data types, statement level control structures, and unit level control structures.

Based on suggestions of Wichmann [WICB84b], Ada language constructs can be further partitioned relative to constructs defined in Pascal. That is, within functional boundaries an Ada construct can be further delineated based on whether it has a Pascal counterpart; and if not, whether it can be easily added to Pascal, or represents a new language feature having a significant influence on the language design issues. Data aggregates, user-defined types, looping, and decision constructs are members of the first set. Partial array assignment, exit statements, and named loops are representatives of the second set. Packages, generics, tasking, and exception handling are each members of the third set. The Pascal oriented categorization is particularly significant because it allows extension of previous research results reported by Farnan [FARM87] and Dandekar [DANA87], so that research focuses on those language constructs and semantic components found in the Ada language but not in Pascal.

Assuming that Farnan and Dandekar have necessarily and sufficiently analyzed conventional language constructs, the critical Ada language constructs requiring additional examination are:

- Data Types
  - Strings
  - Record Discriminants

- Statement Level Control Structures
  - Partial Array Assignments
  - Exit Statements
  - Named Loops with Exits

  - Block Structures

- Unit Level Control Structures
  - Subprograms

    Default Parameters, Name Overloading, Parameter Passing
  - Packages

    Specification

    Body
  - Generics
  - Tasking

    Concurrency Specification
  - Exception Handling.

The authors recognize that the above categorization does not cover all Ada specific language components, but stress that the intentions are to examine only those that are most prominent from a software engineering perspective. Bundy [BUNG90] offers a more detailed explanation of identifying, categorizing, and classifying Ada language constructs with respect to software quality assessment within the OPA framework.

Step 2: *Understanding the Rationale for Component Inclusion*

Before employing code structure analysis as part of a software quality procedure, one must acquire a firm understanding of why particular language constructs have been included in a language definition. In some cases, the rationale might simply be that a specific capability is needed, e.g., looping. From the perspectives of software engineering and software quality assessment, however, of particular interest is the rationale for including constructs like generics, packages, and block structures that are purported to support desirable product design and development capabilities. For Ada, the language designers have provided the *Rationale for the Design of the Ada Programming Language* [ADAR84]. Published papers describing research and development efforts and books describing usage techniques provide additional insights into the proposed uses of Ada language components. Using packages as a representative example, the next paragraph outlines the type of information the authors have sought in synthesizing an adequate understanding for including particular language elements in the definition of Ada.

According to [ADAR84] packages are one mechanism through which the programmer can group constants, type declarations, variables, and/or subprograms. The intent is that the programmer will use packages to group related items. From a software engineering perspective, this particular use of packages is appealing because it promotes code cohesion [ROSD86]. Packages are also a powerful tool in supporting the specification of abstractions. The ability to localize implementation details and to group related collections of information is a prerequisite for defining abstract data types in a language. Again, from a software engineering perspective, the capability to specify abstract data types and to force the use of predefined operations to modify data structures promotes reliability, portability, and maintainability.

Step 3: Assessing Component Importance from a Software Engineering Perspective

To exploit the OPA framework one must determine each individual component's contribution to the achievement of desirable software engineering objectives, its support in the use of accepted software engineering principles, and/or its ability to impart desirable software engineering attributes to the encompassing product. The authors note that the impact of a component on product quality can be beneficial or detrimental. For example, operator overloading generally enhances program readability [WICH84a, GHEC82]. If used indiscriminately, however, it can have the opposite effect [GHEC82].

From an Ada standpoint, the literature abounds with citations attesting to the "software engineering goodness" of Ada language constructs. In particular, Ada packages are extremely important in achieving a quality, software engineered product. Ada packages support four definitional abstractions: named collections of declarations, subroutine libraries, abstract state machines, and abstract data types. One particular abstraction, abstract data types, is fundamental to supporting the software engineering principle of information hiding [ADAR84]. That is, packages defining abstract data types provide the type declaration for an abstract data type and methods for manipulating the data type. What is hidden from the user is the sequence of coded instructions supporting the manipulative operations. Also, the user is forced to modify the abstract data type through the specified operations. This form of information hiding is particularly beneficial when maintenance is required because it tends to minimize the "ripple effect" that change can have. As also discussed by Booch [BOOG83, BOOG87], packages are crucial in supporting modularity, localization, reusability, and portability, all of which are highly desirable from a software engineering perspective.

Step 4: *Identifying the Impact of Component Usage on Desirable Software Engineering Attributes*

In the third step described above language components are associated with rather abstract software engineering qualities like maintainability, reliability, information hiding, and modularity. To implement an assessment procedure within the OPA framework, however, those language components must be aligned with less abstract entities, i.e. the software engineering attributes. This fourth step in the metric development process is crucial in that it establishes such linkages by identifying the impact(s) of each language construct on one or more (less abstract) software engineering attribute. This fourth step is illustrated below by considering the impact of packages relative to selected software engineering attributes.

As a basis, the authors examined the four proposed uses of packages in linking package properties to software engineering attributes. For example, packages that contain only type declarations indicate code cohesion [ROSD86]. The other three proposed uses are packages to define abstract data types, packages to define abstract state machines and packages to define subprogram units. Although all four of these uses induce desirable attributes in the developed product (see [GANJ86, EMBD88, BOOG87], respectively), improper use of packages can also have a negative impact on the desirable product attributes. For example, the use of packages to group type declarations has diminishing returns when too many type declarations are exported. This misuse hinders ease of change because program units must be unnecessarily checked for possible impacts caused by changes to declaration packages.

Consider as a detailed illustration of the above, the use of packages to define abstract data types (the authors will refer to such packages at ADT packages). The benefits (relative to the inducement of desirable software engineering attributes) of ADT packages are enhanced cohesion (functional and logical), a well-defined interface to the ADT, and enhanced ease of change for program units "withing" the ADT package. The improved cohesion results from the grouping of the ADT declarations and access operations within one package. A well-defined ADT interface is achieved by using the package specification to house the subprogram specification for each ADT and then using private or limited private types to restrict access to the ADT. From a different perspective, because of the capabilities provided by packages, the use of ADTs has additional beneficial effects in terms of reduced code complexity and improved readability. Without further elaboration, it suffices to say that the definition of ADTs through packages embraces the use of abstractions that hide superfluous details from the ADT user.

Step 5: *Identifying Properties, Defining Indicators, and Formulating Measures and Metrics*

The fourth step of the metric development procedure describes the impact that component uses and abuses have on the software engineering attributes. Step 5 identifies and formally links product properties (language elements) to software engineering attributes. Because each identified property undeniably reflects either the presence or absence of a specified attribute, the authors refer to the property/attribute pair as an indicator. Building on the relationship between the property/attribute pair, a measurement approach and supporting metric is defined. These three activities are being discussed together, as a single step, because they are intrinsically tied together. To illustrate Step 5 of the metric development procedure the remainder of this section focuses on the identification of properties indicative of the presence of the attribute cohesion relative *to the use of packages in defining groups of subprograms.*

To begin the process one identifies those properties associated with the use of packages to define subprogram units and the attribute(s) that usage affects. In the cohesion example, the task is to identify characteristics that a cohesive package would exhibit. One such characteristic is the utilization of subprograms defined within a package. In particular, each program unit that "withs" the package of subprograms utilizes a percentage of the subprograms. A very low utilization suggests that the subprograms grouped by the package are not as closely related (or functionally cohesive) as they should be. A very high utilization suggests that the subprograms are closely related or functionally cohesive.

The description presented in the previous paragraph suggests the identification of a property, the establishment of a link between a particular property and attribute, a measurement approach and a supporting metric. In particular, the property/attribute indicator is the "definition of packages that export subprograms relative to its positive impact on code cohesion." Hence, to effectively measure the cohesiveness of packages that export subprograms, one must examine the utilization of the subprograms by "withing" units. Intuitively, if the subprograms are sufficiently related, any unit that "withs" the package will use a majority of the subprograms. The indicative metric, calculated on a per package basis, is given with the following formula:

$$\text{Sub Package Utilization} = \frac{\displaystyle\sum_{\substack{\text{"Withs"}\\ \text{to a Sub}\\ \text{Package}}} \text{package subprograms referenced}}{(\text{total \# of "withs"}) * (\text{\# of subprograms in the package specification})}$$

(Note: Sub Package refers to a package that exports subprograms)

The analysis of packages that define abstract data types and of packages that define abstract state machines provides similar results. Appendix D lists all proposed code indicators and their measurement approaches. For additional detail the authors refer the reader to [BUNG90].

3.3.2 Code Indicators: A Summary

The Objectives, Principles, Attributes framework provides a formal basis for defining a software quality assessment procedure relative to Ada-based products. Currently the authors have identified 66 automatable indicators:

8 are based on data type information, 12 exploit properties of statement level structures, and 46 reflect characteristic assessments of unit level constructs like packages, subprograms and so forth.

Because Ada is designed to support software engineering activities, evaluating Ada code for the achievement of objectives, use of principles, and presence of attributes is important to both developers of Ada-based products and the purchasers of Ada-based products. Instrumental to the proposed assessment procedure are the indicators and metrics reflecting the impact of Ada and the development of the Ada analyzer to facilitate the automated assessment process. The latter topic is discussed below.

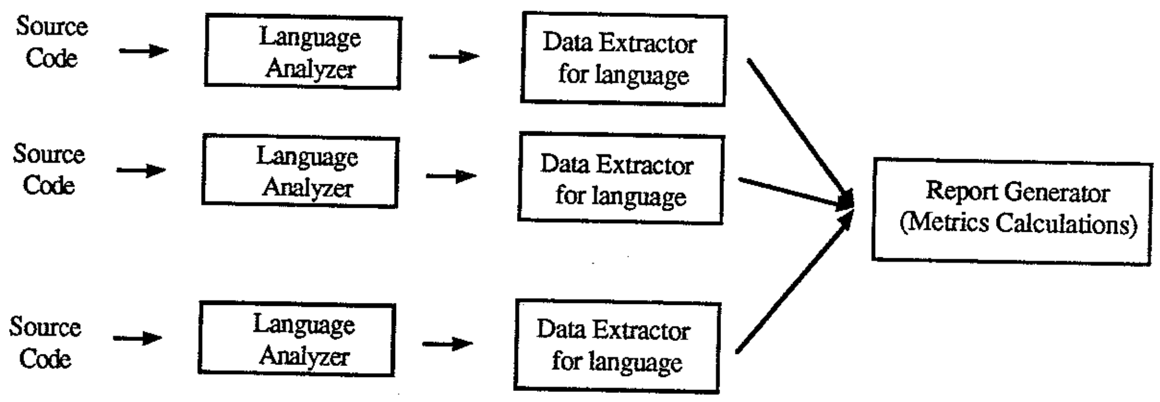
4.0 Extending the Ada Analyzer and Report Generator

Experience has taught that manually collecting the data necessary to compute the defined metrics is labor intensive and error prone. Correspondingly, a major research thrust has focused on the development of an automated system for assessing the quality of Ada products. The existing report generation system consists of three software tools that examine Ada code and produce a report detailing the presence of desirable software engineering attributes in the product, the use of principles, and a projected achievement of software engineering objectives.

As implied above the system has been intentionally designed to separate language particulars from the actual metrics values being calculated. The intent of this design decomposition is to promote extensibility. As illustrated in Figure 3, one can collect data from several sources and use the same report generator to compute metric values and to analyze the data. The three components of the current system are:

- the Ada Analyzer - ADALYZE,
- the Data Extractor - DEX, and
- the Report Generator - RGEN.

ADALYZE is a language dependent analyzer that accepts Ada programs as input and produces data items that support metric computations. Effectively, ADALYZE is a "compiler" that generates several language sensitive data files. In particular, these data files contain information reflective of Ada language characteristics, e.g. the number of packages exporting subprograms. The language sensitive data files are then passed to the data extractor (DEX). The function of DEX is to transform the language sensitive files into a more universal format, i.e. one that is primarily language independent. Because the data file created by DEX is in a language independent format (structured by metric data needs), only one metric computation program is needed regardless of the language being analyzed. Figure 3 illustrates how the assessment process is performed when components of more than one language are to be analyzed.



Source Code
Language Analyzer
Data Extractor for language
Source Code
Language Analyzer
Data Extractor for language
Report Generator (Metrics Calculations)
Source Code
Language Analyzer
Data Extractor for language

Figure 3
Overall View of Information Flow

Structured to exploit knowledge of required metric computations, the report generator (RGEN) reads the language independent files created by the data extractor and produces (a) a summary of the data elements and (b) a report reflecting the OPA hierarchy among attributes, principles, and objectives. Reflecting the current system architecture, Figure 4 illustrates a more detailed view of the relationship among the computational components and depicts how information flows between those components. For additional details about the analyzer and report generator, the authors refer the reader to the user manual in Appendix E.

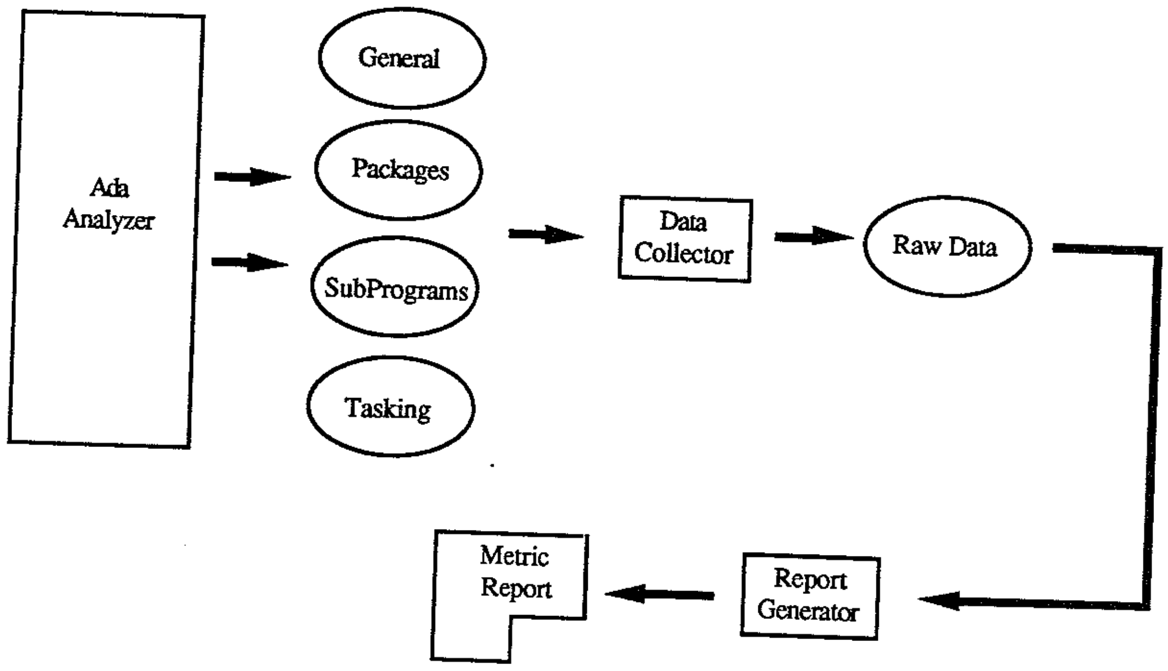


Figure 4
Detailed Diagram of Information Flow in the Ada Analyzer and Report Generator

5.0 Site Selection for the Validation Project

The set of desirable site characteristics, prepared on April 1 1990, is shown in Appendix F. The set includes five (5) project level criteria, eleven (11) site specific criteria, and thirty-six (36) criteria related to software development activities. Project level criteria focus on desirable characteristics of the proposed software system to be used in the validation effort. Sample project level criteria include the desire for the system development effort to be a new initiative, Ada-based, deployment within 18 months, and consisting of 10,000 to 50,000 source lines of code. Site specific criteria address capabilities and characteristics that are desirable of and generally applicable to a software development effort. For example, formal review processes, employment of a configuration management system, and an established provision for electronic document preparation are among the desirable site specific characteristics. Finally, categorized according to software development phases, the criteria associated with software development activities include desirable process activities like formal requirements specifications and review, preliminary design and review, and the production of expected products from such activities, e.g., review documents.

The set of site selection criteria is to be used as a basis for assessing the adequacy of a site and project for supporting the proposed validation effort. The authors recognize and stipulate that no one site is expected to meet all of the stated criteria. Site selection is achieved through an examination of criteria that each proposed site meets and a tradeoff assessment relative to those criteria.

6.0 Summarization and Future Plans

6.1 Year One: Completed Work

The process indicator effort focuses on the entire software life-cycle and utilizes data items reflecting development activity trends and characteristics of products stemming from those activities. An examination of requirements activities and associated products is complete. The authors are currently examining activities associated with the design process and several phase-independent characteristics like personnel profiles and staffing levels. Concurrently, the authors are investigating a modified software life-cycle model that supports a characterization of the relationships among process and product characteristics within and among the software development life-cycle phases.

From the perspective of document quality indicators, an initial set of 36 indicators and 33 supporting metrics have been defined. Current research is addressing the computational tractability of the proposed indicators and the use of approximation measures for those indicators that require data items which are difficult (or impossible) to collect. Additionally, the authors are re-examining each DQI relative to its linkage to OPA framework. This re-examination is also forcing the authors to broaden the current definition OPA entities and linkage set.

As discussed earlier, the formulation of code indicators is complete. Although more than 100 have been identified, only 66 are automatable. The Ada analyzer and report generator exploits the definition of these 66 code indicators in their task to produce reports that attest to the quality of Ada code. The code analyzer and report generator are complete and have been tested on several data sets, each comprising several thousand source lines of code. One error has been found; it is currently being addressed. A independent verification and validation of the analyzer and report generator is deemed desirable and will be pursued if time and monies permit.

6.2 Year Two: Current Research

Research directions for the current year include an extension of current work as well as the initiation of new efforts.

- *Process Indicators:* The authors continue to work toward the refinement of currently identified process indicators and search for new ones. In particular, efforts are focused on high- and low-level design activities, development trends and phase independent characteristics that impact product quality.
- *Valid Data Acquisition:* Recognizing the criticality of necessary, sufficient and valid data acquisition has prompted a preliminary investigation of data collection methods. Research efforts continue in the exploration of various data acquisition methods, with particular attention being given to studies emphasizing the collection of software engineering and software development data.
- *Automated Document Analysis:* The formal identification and definition of documentation quality indicators are nearing completion. Subsequent research efforts must address the collection of necessary data elements and the synthesis of a document analyzer. The authors expect to complete both the definition of DQIs and a corresponding document analyzer by the end of year two.
- *Site Selection and Examination:* Crucial to the validation effort is the selection of a validation site in the near future. Following the site selection, the authors anticipate several site visits to examine the software development process and to identify potential instrumentation points for data collection.
- *Instrumentation of the Development Process:* Following site selection and examination, an SRC/VPI team is expected to initiate an instrumentation and indicator refinement phase. This effort will require the on-site presence of SRC/VPI personnel for extended periods of time.
- *Initiation of Data Collection:* the collection of selected validation data is anticipated during the latter part of year two.

6.3 Years Three and Four: Future Plans

Based on the characteristics of the potential validation site being considered, year three is expected to yield sufficient validation data to initiate preliminary statistical studies. The authors anticipate the substantiation (or refutation) study of selected validation hypotheses during the latter part of year three and the first part of year four.

The primary efforts during year four, however, will focus on further refinement of predictive indicators and the collection of additional management indicators that support and solidify validation results.

References

[ABDT89] Abdel-Hamid, T., "The Dynamics of Software Project Staffing: A System Dynamics Based Simulation Approach," *IEEE Transactions on Software Engineering*, Vol. 15, No. 2, February 1989, pp. 109-119.

[ADAR83] *Reference Manual for the Ada Programming Language*, Ada Join Program Office, ANSI/MIL-STD-1815A, 1983.

[ADARF84] *Rationale for the Design of the Ada Programming Language*, Minneapolis, MN: Honeywell Systems and Research Center, 1984.

[ARTJ86] Arthur, J., Nance, R. and Henry, S., "A Procedural Approach to Evaluating Software Development Methodologies: The Foundation," Technical Report SRC-86-008, Systems Research Center, Virginia Tech, 1986.

[ARTJ87] Arthur, J. and Nance R., "Developing an Automated Procedure For Evaluating Software Development Methodologies and Associated Products," Technical Report SRC-87-007, Systems Research Center and Department of Computer Science, Virginia Tech, 1987.

[BARJ80] Barry, J. "Computerized Readability Levels," *IEEE Transactions on Professional Communication*, vol. PC-23, no. 2, June 1980, pp. 88-90.

[BASV84] Basili, V. and Weiss, D., "A Methodology for Collecting Valid Software Engineering Data," *IEEE Transactions on Software Engineering*, Vol. 10, No. 6, November 1984, pp. 728-738.

[BOEB81] Boehm, B., Software Engineering Economics, Prentice-Hall, 1981.

[BOOG83] Booch, G., *Software Engineering with Ada*, Menlo Park, CA: The Benjamin/Cummings Publishing Company, 1983.

[BOOG87] Booch, G., *Software Components with Ada*, Menlo Park, CA: The Benjamin/Cummings Publishing Company, 1987.

[BROF75] Brooks, F., The Mythical Man-month, Addison-Wesley, 1975.

[BUNG90] Bundy, G., "The Objectives, Principles, Attributes Approach for Measuring Software Quality in Ada Based Products," M.S. Thesis, Computer Science Department, Virginia Tech, July, 1990.

[CHUK90] Church, K. and Hanks, P., "Word Association Norms, Mutual Information, and Lexicography," Computational Linguistics, vol., 16, no. 1, March 1990, pp.22-29.

[DANA87] Dandekar, A., "A Procedural Approach to the Evaluation of software Development Methodologies," M.S. Thesis, Computer Science Department, Virginia Polytechnic Institute and State University, September, 1987.

[DRUA85] Drury, A. "Evaluating Readability," *IEEE Transactions on Professional Communication*, vol. PC-28, no. 4, December 1985, pp. 11-14.

[EJIL87] Ejiogu, LEM O., "The Critical Issues of Software Metrics--Part 0. Perspectives on Software Measurements," *SIGPLAN Notices*, Vol. 22, No. 3, March 1987, pp. 59-64.

[EMBD88] Embley, D. and Woodfield, S. "Assessing the Quality of Abstract Data Types Written in Ada," *Proceedings: 10th International Conference on Software Engineering*, April 1988, pp. 144-153.

[FARM87] Farnan, Mark A., "The Automation of a Set of Code Metrics for Pascal," M.S. Project, Computer Science Department, Virginia Polytechnic Institute and State University, September, 1987.

[FRYE68] Fry, E. "A Readability Formula That Saves Time," *Journal of Reading*, vol. 11, no. 7, April 1968, pp. 513-516, 574-578.

[GANJ86] Gannon, J. D., Katz, E. and Basili, V., "Metrics for Ada Packages: An Initial Study," *Communications of the ACM*, Vol. 29, No. 7, July 1986, pp. 616-623.

[GHEC82] Ghezzi, C. and Mehdi Jazayeri, *Programming Language Concepts*, New York, John Wiley & Sons, Inc., 1982.

[HALD88] Hale, D.R., and Haworth, D.A., "Software Maintenance: A Profile of Past Empirical Research," *IEEE Conference on Software Maintenance*, Vol. PC-23, No. 2, June 1980, pp. 87-88.

[HALM77] Halstead, Maurice H., *Elements of Software Science,* New York: Elsevier North-Holland, Inc., 1977.

[HAMC85] Hammons, Charles and Paul Dobbs, "Coupling, Cohesion, and Package Unity in Ada," *Ada Letters,* Vol. 4, No. 6, May/June 1985, pp. 49-59.

[HENRS81] Henry, Sallie and Dennis Kafura, "Software Structure Metrics Based on Information Flow," *IEEE Transactions on Software Engineering,* Vol. 7, No. 5, September 1981, pp. 510-518.

[HUMW87] Humphrey, W., "A Method for Assessing the Software Engineering Capability of Contractors, " Technical Report CMU/SEI-87-tr-23. Software Engineering Institute, Carnegie Mellon University, September 1987.

[HUMW89] Humphrey, W., Managing the Software Process, Addison-Wesley, 1989.

[KEAJ86] Kearney, J., Sedlmeyer, R., Thompson, W., Gray, M., and Adler, M., "Software Complexity Measurement," *Communications of the ACM,* Vol. 29, No. 11, November 1986, pp. 1044-1050.

[MCCG87] McClure, G.M. "Readability Formulas: Useful or Useless?," *IEEE Transactions on Professional Communication*, vol. PC-30, no. 1, March 1987, pp. 12-15.

[MCCT76] McCabe, Thomas J., "A Complexity Measure," *IEEE Transactions on Software Engineering,* Vol. 2, No. 4, December 1976, pp. 308-320.

[PARD85] Parnas, D., "Software Aspects of Strategic Defense Systems," *Communications of the ACM,* Vol. 28, No. 12, December 1985, pp. 1326-1335.

[ROSC88] Rosson, C., *Management Indicators: Assessing Product Reliability and Maintainability*, Technical Report SRC-88-011, Systems Research Center, Blacksburg, Virginia.

[ROSD86] Ross, Donald L., "Classifying Ada Packages," *Ada Letters,* Vol. 6, No. 4, July/August 1986 , pp. 53-65.

[ROSN90] Ross, N., "Using Metrics in Quality Management," *IEEE Software,* July 1990, pp. 80-85.

[SALG89] Salton, G., Automatic Text Processing: The transformation, Analysis, and Retrieval of Information by Computer, Addison-Wesley Publishing Co., 1989.

[SHUK88] Shumate, Ken and Khell Nielsen, "A Taxonomy of Ada Packages," *Ada Letters*, Vol. 8, No. 2, March/April 1988, pp. 55-76.

[STEK88] Stevens, K.T., "A Taxonomy for the Evaluation of Computer Documentation," Master's Thesis, Virginia Tech, 1988.

[VERC80] Vervalin, C.H. "Checked Your Fog Index Lately?," *IEEE Transactions on Professional Communication*, vol. PC-23, no. 2, June 1980, pp. 87-88.

[WICB84a] Wichmann, B. A., "Is Ada too Big? A Designer Answers the Critics," *Communications of the ACM*, Vol. 27, No. 2, February 1984, pp. 98-103.

[WICB84b] Wichmann, B. A., "A Comparison of Pascal and Ada," *Comparing and Assessing Programming Languages*, Englewood Cliffs, NJ: Prentice-Hall Inc., 1984.

Appendix A

Measures for Process Indicators

(Metrics not included)

The Process Indicators presented in this Appendix have the following form:

Process Property:	The measurable characteristic of a development process that has an affect on the quality of the resulting product.
Impact of the Property:	The specific impact of the process property on the development process and the product is described.
OPA Entity Affected:	The specific objective, principle, product attribute or process attribute affected by the process property is stated in this section.
Justification:	The justification of why and how the stated process property impacts the OPA entity is presented in this section. A logical, intuitive argument is made for linking the process property to the OPA entity.
Measurement Approach:	This section presents a measurement approach that captures the affect a process property has on a product or process attribute.
Metric:	Although preliminary metrics for the following process indicators have been formulated, they are not included in this report because they are still being refined and, subsequently, are not finalized.

Process Property:	The addition and deletion of requirements following the Software Specification Review (SSR).
Impact of the Property:	Functionality added or deleted following the SSR adversely affects the decomposition decisions made during high level design of the system.
OPA Entity Affected:	Design Decomposition
Justification:	Requirements decomposed along functional or hierarchical boundaries are adversely affected by the addition or deletion of functionality following the Software Specification Review. New functionality must be integrated into a design along previously determined design boundaries. The added requirement is broken down along existing decomposition boundaries, negatively affecting the uniformity of decomposition at each level. The new functionality increases coupling among existing design modules because implementation of this new functionality requires the correctness of all modules which support the added requirement. Testing of the new requirement is also made more difficult. Deletion of requirements reduces the functionality of the design modules affected. This narrowing of functionality reduces the benefit of decomposition by leaving functional boundaries around design modules with newly restricted functionality.
Measurement Approach:	The measurement approach is to calculate the total number of design boundaries affected by the addition and deletion of requirements, tempered by the total number of design boundaries in the design.

Proposed Process Indicators

Process Property: The detection of requirements defects after the Software Specification Review (SSR).

Impact of the Property: The detection of requirements defects following the SSR is evidence that requirements errors were not detected early, before the design phase, in the development process.

OPA Entity Affected: Early Error Detection

Justification: One purpose of the Software Specification Review is to contribute to the detection of errors early in the development process. All errors present in the Software Requirements Specification are assumed to have been discovered during this Review. The detection of requirements defects after the Software Specification Review implies that requirements defects have propagated to design and code. Errors that have propagated in this way reduce the actual number of errors which should have been detected earlier (before design and coding). The extent to which Early Error Detection has been achieved can be judged by (a) the number of defects found following the Software Specification Review and (b) when those defects were found

Measurement Approach: The measurement approach is to weight number of defects detected per month by the number of months elapsed since the SSR. Heavier weight is given to those defects which have been present in the requirements for a longer period of time. The weighted number of defects is tempered by the total number of requirements specified. The monthly metric values are then averaged to obtain the quantitative (time-sensitive) impact of requirements defects on early error detection.

Proposed Process Indicators

Process Property:	The recording of requirements that are not testable.
Impact of the Property:	Untestable software requirements do not allow early error detection to be achieved in the product.
OPA Entity Affected:	Early Error Detection
Justification:	Requirements should be sufficiently defined in order to minimize the number of requirements that are not testable. Requirements that are not testable cannot be verified until they are deployed. The software realization of these requirements must be implemented and installed before any testing can be performed. This clearly decreases the overall amount of early error detection possible in the software product.
Measurement Approach:	Divide the number of requirements which are not testable by the total number of requirements.

Proposed Process Indicators

Process Property: Design review completeness

Impact of the Property: The completeness of the design review process significantly impacts the extent to which early error detection is reflected in the product.

OPA Entity Affected: Early Error Detection

Justification: All design modules should be reviewed during the critical design review. Those design modules which are not reviewed, due to time shortage, oversight or intentional omission, detract from the extent of early error detection achieved in the product. These units have not been examined thoroughly for the presence of errors. This type of incompleteness results in portions of the product where detection of errors has not been fully achieved.

Measurement Approach: The measurement approach is to reduce the level of early error detection achieved in the system relative to the number of references to design modules not reviewed during the design review. That is, for each reviewed module, reduce the level of early error detection imparted on the product by the percentage of unreviewed modules referenced.

Proposed Process Indicators

Process Property: The updating of software development folders (SDFs).

Impact of the Property: SDFs are a repository containing change documentation for each module over the lifetime of the module. The recording of changes documents the functional history of the module.

OPA Entity Affected: Visibility of Behavior

Justification: Software development folders should be updated each time a design or code module is changed. Design or code modules that do not have this one to one update correspondence exhibit a gap in the historical documentation needed for understanding the functionality and internal behavior of the design or code module. Updates to a design or code module that are not documented in the module's SDF reduce the ability to retrace the functional and structural change history of the design or code module over the module's lifetime.

Measurement Approach: The measurement approach is to assess the extent to which the system exhibits visibility of behavior by calculating the percentage of updates to each module that are documented in the module's corresponding SDF.

Proposed Process Indicators

Process Property: The creation of Software Development Folders (SDFs)

Impact of the Property: A SDF should exist for a module when the module is added to the existing system in order to record the need for creating the module.

OPA Entity Affected: Traceability

Justification: Software development folders should exist when a design or code module is initially included in the configuration management system. This immediate creation of a SDF satisfies the necessary condition for recording the motivation for creating the module. Modules added to design without corresponding SDFs do not have the repository for information critical to tracing the module's history. Effectively, there should never exist a design or code module without a corresponding SDF. Design or code modules that do not have an SDF do not satisfy the necessary condition for recording the historical information needed to insure module traceability.

Measurement Approach: The measurement approach is to reduce the amount of traceability present in the product by the number of weeks each design or code module does not have a corresponding SDF. That is, traceability is reduced by the percentage of a module's existence during which no corresponding SDF existed.

Proposed Process Indicators

Process Property: The involvement of unacclimated personnel in the development process.

Impact of the Property: Development personnel which are not acclimated to the development process and the target product are more error-prone.

OPA Entity Affected: Introduction of errors/Addition of unacclimated personnel

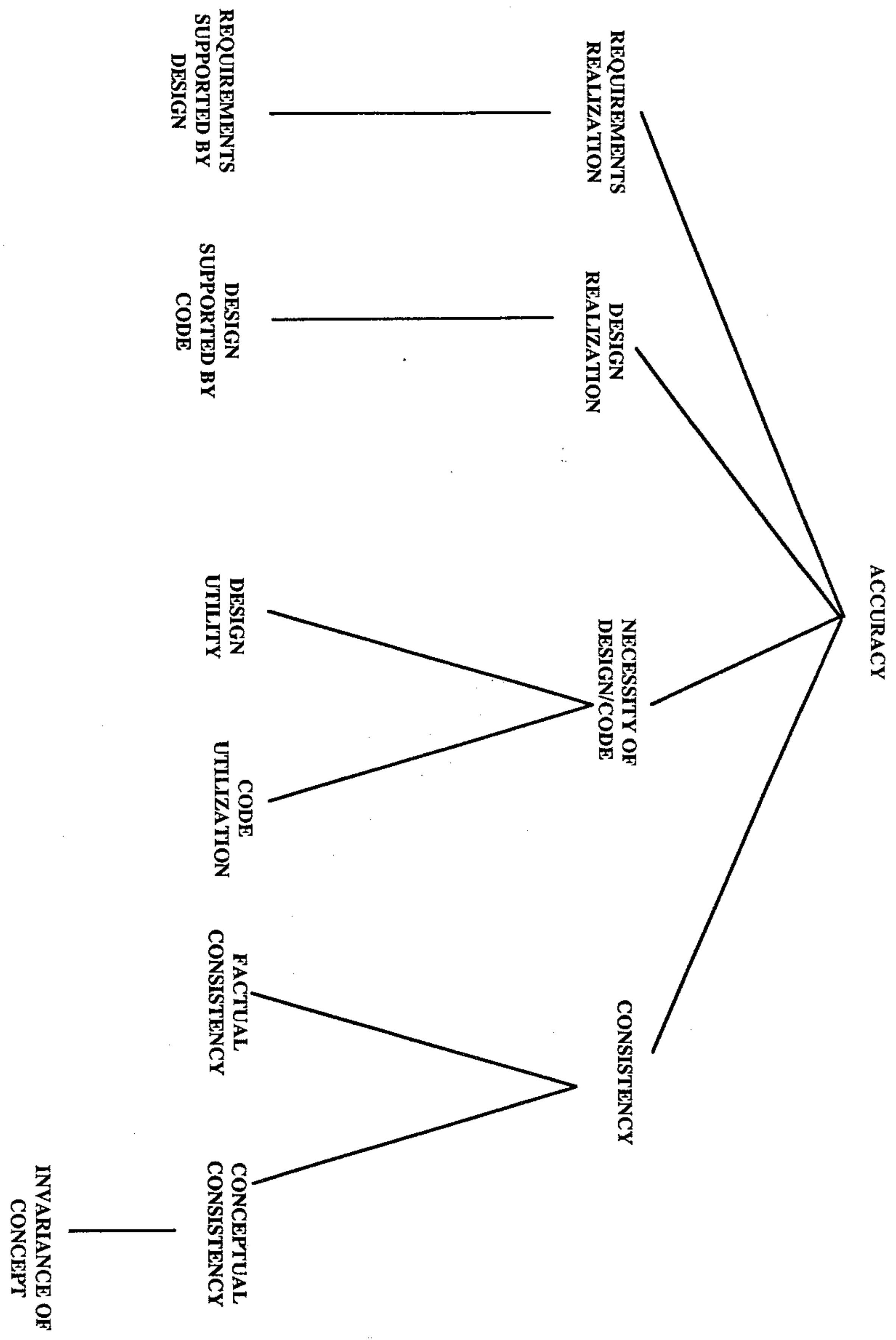
Justification: A key part of the principle of Life-Cycle Verification is the reduction of the number of errors in the product during the early phases of the software development. This reduction is achieved in part by construction of the intermediate products by technical staff familiar with the historical background of the process and product. Unacclimated personnel involved in an ongoing project require a four to twelve week period to become acclimated to product requirements, the development process and the existing organization. During this time the unacclimated personnel are engaged in the production and review of the product. Unacclimated personnel are more error-prone than acclimated personnel, introducing errors in the intermediate products. Unacclimated personnel are less effective than acclimated personnel in recognizing errors in the product during production

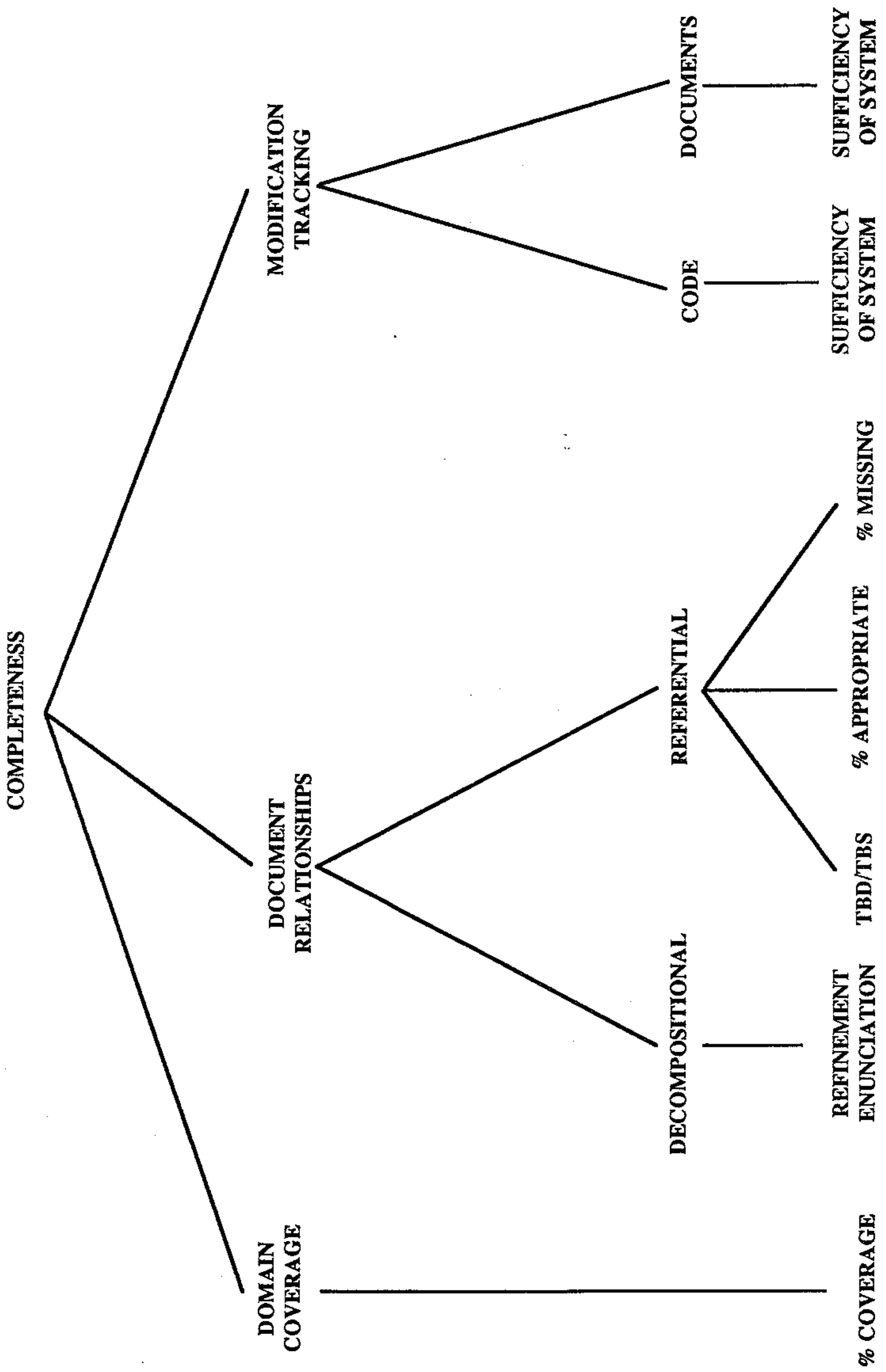
Measurement Approach: The measurement approach is to increase the number of errors introduced in the product proportional to the number of unacclimated personnel involved in the development process.

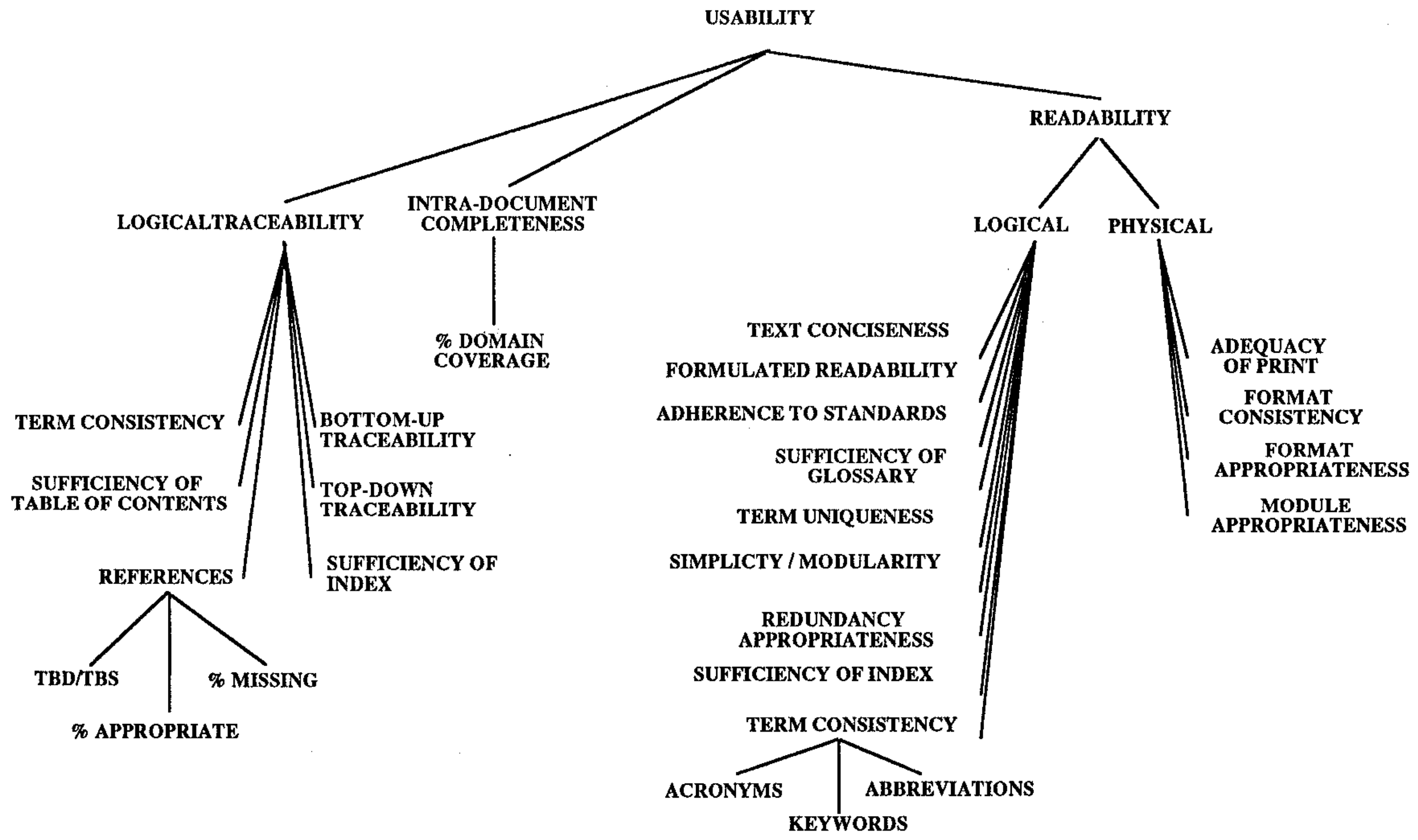
Appendix B

Documentation Quality Indicator Hierarchy

(Preliminary Version)


ACCURACY
REQUIREMENTS REALIZATION
DESIGN REALIZATION
NECESSITY OF DESIGN/CODE
CONSISTENCY
REQUIREMENTS SUPPORTED BY DESIGN
DESIGN SUPPORTED BY CODE
DESIGN UTILITY
CODE UTILIZATION
FACTUAL CONSISTENCY
CONCEPTUAL CONSISTENCY
INVARIANCE OF CONCEPT


COMPLETENESS
DOMAIN COVERAGE
DOCUMENT RELATIONSHIPS
MODIFICATION TRACKING
DECOMPOSITIONAL
REFERENTIAL
CODE
DOCUMENTS
% COVERAGE
REFINEMENT ENUNCIATION
TBD/TBS
% APPROPRIATE
% MISSING
SUFFICIENCY OF SYSTEM
SUFFICIENCY OF SYSTEM


USABILITY
LOGICALTRACEABILITY
INTRA-DOCUMENT COMPLETENESS
READABILITY
TERM CONSISTENCY
SUFFICIENCY OF TABLE OF CONTENTS
REFERENCES
TBD/TBS
% APPROPRIATE
% MISSING
BOTTOM-UP TRACEABILITY
TOP-DOWN TRACEABILITY
SUFFICIENCY OF INDEX
% DOMAIN COVERAGE
LOGICAL
PHYSICAL
TEXT CONCISENESS
FORMULATED READABILITY
ADHERENCE TO STANDARDS
SUFFICIENCY OF GLOSSARY
TERM UNIQUENESS
SIMPLICTY / MODULARITY
REDUNDANCY APPROPRIATENESS
SUFFICIENCY OF INDEX
TERM CONSISTENCY
ACRONYMS
KEYWORDS
ABBREVIATIONS
ADEQUACY OF PRINT
FORMAT CONSISTENCY
FORMAT APPROPRIATENESS
MODULE APPROPRIATENESS

Appendix C

Measures for Document Quality Indicators

(Metrics not included)

Each document quality indicator presented in this Appendix has the following format:

Property: The name given to the property of the documentation which is assessed and evaluated using the measurement approach described under the *measure* heading.

What is measured: A brief description of the property being measured.

Rationale: The reason(s) the assessment of this particular property is germane to the assessment of documentation quality in general.

Measurement Approach: The approach taken to evaluate this documentation property, including the data collected, the methods used to collect the data, and the way(s) the data is used to render an evaluative measure of the property in question.

Metric: Although preliminary metrics for the following documentation quality indicators have been formulated, they are not included in this report because they are still being refined and, subsequently, are not finalized.

Proposed Document Quality Indicators

Property: Percent Appropriate References

What is measured: The proportion of references made within the documentation which are deemed appropriate or necessary.

Rationale: Unnecessary references are a waste of the user's time because time spent tracking down irrelevant or insignificant reference could be better spent pursuing important information. A low score in this indicator could be indicative of:

- a "jumbled" or disorganized documentation process, or
- overkill on the part of the documentation team (how much is too much?).

Measurement Approach: References made throughout the documentation are collected in a table containing the subject and the location(s) referenced. The referenced locations are then assessed for appropriateness by determining the amount of relevant information present at the referenced location. Such an assessment can be implemented by consulting the linkages derived through a keyword indexing procedure to determine if two referencing or co-referencing subjects are "related" to one another. The ratio of appropriate references to the total number of references is evaluated.

The references can be collected by noting section numbers not corresponding with the current section.

Proposed Document Quality Indicators

Property:	Percent Missing References
What is measured:	The proportion of missing references to the total number of existing references in the documentation. A "missing" reference is defined as: • a referenced section that doesn't contain the subject referenced, or • the lack of a reference where one would be appropriate. This measure shall take into account only the first type of missing reference; determining where a reference *should* be is not automatable.
Rationale:	Documentation that has sections which are inaccurately linked together is not only less useful, but less complete than documentation which is linked literally as well as hierarchically.
Measurement Approach:	A table of references is constructed, including the referenced locations. Each cited location is checked to determine if the topic it is purported to cover is actually present at the location. The measure is the ratio of the number of missing references to the total number of references.

Proposed Document Quality Indicators

Property: Abbreviation Usage

What is measured: The degree to which abbreviations are used in a well-defined and consistent manner.

Rationale: Abbreviation usage is helpful to the user of a document only if the abbreviations are well-defined and used in a consistent manner.

Measurement Approach: Each abbreviation used in a document will be checked for the following:

- definition at initial usage, and that
- each usage of the abbreviation is in a similar context (i.e., has the same meaning).

The measure is computed by combining the scores obtained by evaluating the above characteristics.

The meaning of an abbreviation is simply its verbose form. For example, m.p.h stands for "miles per hour", and should not be used in the same document as meaning "multiple pay days". This would be an instance of inconsistent usage of an abbreviation.

Abbreviations can be identified by the following characteristics:

- words ending in a period (.) that are not followed by 2 spaces and capitalized words. (problems: typing errors and proper nouns can cause a preceding word to appear as an abbreviation), and
- *groups* of letter-period-letter-period sequences (e.g. m.p.h.).

Proposed Document Quality Indicators

Property: Acronym Consistency

What is measured: The degree to which acronyms are used in a well-defined and consistent manner.

Rationale: Acronym usage is helpful to the user of a document only if the acronyms are well-defined and used in a consistent manner.

Measurement Approach: Each acronym used in a document is checked for the following:

- definition at initial usage,
- definition in the glossary or acronym list accompanying the documentation, and that
- each usage of the acronym is within a similar context (referred to below as a "proper use").

The measure consists of a combination of the evaluations of the above characteristics.

The meaning of an acronym is its verbose form. For example, SRC "means" Systems Research Center. Once SRC has been used in this context, it should not appear anywhere in the same document(s) intending another meaning, e.g. State Regulatory Commission. This would be an instance of inconsistent usage of an acronym.

Proposed Document Quality Indicators

Property: Adequacy of Print

What is measured: The adequacy of the display techniques used in the documentation.

Rationale: The readability of a document is affected by its physical attributes as well as the way in which the content is presented. Line length, character size, highlighting and use of white space are all influential in the physical readability of a document.

Measurement Approach: For each of the categories mentioned in the rationale, a measure is computed as appropriate. Each of these measurements is compared against accepted standards from the literature and ranked accordingly. These measures will remain separate so that the nature of a deficiency can be readily identified.

Optimal character size has been stated to be 8-10 points, depending upon the font used. A judgement as to where in this range the employed font lies can be made on a per document basis with little loss of generality.

Highlighting techniques, such as bold typeface, italics, and underlining will be noted, but a means to determine the appropriate use of such techniques has yet to be developed.

Proposed Document Quality Indicators

Property:	Adherence to Standards
What is measured:	The degree to which documentation is in accordance with the specified documentation standards. This measure is similar to Domain Coverage, but the emphasis is more focused on format than content.
Rationale:	Consistent use of standardized methods enhances the ability of user(s) of the documentation to comprehend the information presented.
Measurement Approach:	A list of required items (e.g. TOC, index, acronym list) is compiled by referencing the particular documentation standard in use. The documentation is then examined to determine the extent to which each item is present. By using a weighting scheme to assign relative importance to each item being considered, an overall score is calculated.

Proposed Document Quality Indicators

Property: Bottom-Up Traceability

What is measured: The extent to which the claimed functionality of a code component is designated by successively higher levels of specifications through design to requirements.

Rationale: From any level, a path of designated components at successively higher levels of specification should be defined to promote ease-of-change. This applies particularly to the objective of maintainability. In order to update/correct/change a code module, the maintainer might need to reference the design specification that motivated the code module's creation. Further, reference to the requirement specification might also be necessary.

Measurement Approach: This measurement approach uses three sets as its domain. These sets are:

Base sets (determined from raw data):

r = requirements (r_k) set forth in system specification,
d = design specifications (ds_j) from design documentation, and
c = code modules (cm_i) used to implement design.

Derived sets (calculated from Base sets):

cd = code modules that support some design specification. Sets of (cm_i, ds_j) pairs, and
dr = design specifications that support some requirement specification. Sets of (ds_j, r_k) pairs
cdr = cd $\text{JOIN}_{cd.ds=dr.ds}$ dr, or (cm_i, ds_j, r_k) triples, where (cm_i, ds_x) is a member of cd, (ds_y, r_k) is a member of rd, and $ds_j = ds_x = ds_y$, where x = y.

The set cdr represents code modules reflected in both design and requirements.

Proposed Document Quality Indicators

Property: Code Utilization

What is measured: The degree to which the existing code modules are necessary to fulfill the design specifications set forth in the design documentation.

Rationale: Code that supports no design specification is superfluous, possibly the remnant of a previous design specification which has since been deleted or a 'utility' module constructed in advance and never used. Such code may actually be useful, but appears to be superfluous as a result of insufficient documentation. Poorly documented code modules can create difficulties during maintenance. Also, new errors might appear as a result of repairs made to a code module that has an undocumented influence on another portion of the product. For these reasons, the utilization of code modules must be both maximized *and* documented.

Measurement Approach: A set-based approach is used to assess this property. First, a set of all code modules, C_{total} is constructed. Next, a subset of C_{total}, C_{used} is constructed. C_{used} contains only those code modules that support the implementation of some design specification(s) (i.e., a code module must support a *valid* design module to be a member of C_{used}). Perhaps D_{used} (see Design Utility) could be used as the set of valid design modules.

The cardinalities of these two sets are used to calculate the proportion of necessary code modules relative to all extant code modules. Note that a low value could indicate either poorly documented code, *or* the existence of unnecessary code modules.

Proposed Document Quality Indicators

Property: Completeness of Index

What is measured: The extent to which the index includes all keyword occurrences in the documentation.

Rationale: A good index enables the user to selectively read a document by noting the location of important topics within the text of the document. In order to provide a complete coverage of the important topics contained within a document, the index must include as many keywords as possible.

Measurement Approach: Completeness of the index is assessed by listing the number of entries in the index and comparing this list to a list of topics that *should* be covered in the index. The list of "should be's" might be compiled automatically by extracting terms from a document that are deemed *important* by some evaluation criteria (e.g., frequency of occurrence, occurrence in the TOC). Deductions are made for each omitted entry, using a weighting scheme reflecting the importance of the omitted term. Similar deductions are made for entries that do not list all occurrences of the topic.

Proposed Document Quality Indicators

Property: Completeness of Table of Contents

What is measured: The extent to which all important topics contained in the documentation are included in the table of contents.

Rationale: If a table of contents is to be of assistance to the user of a document, it must be complete. This means that the table of contents should include all topics that are of sufficient importance that have a section of the documentation has been dedicated to them. If important topics are not included in the TOC, then the table of contents loses effectiveness as a reference tool.

Measurement Approach: Completeness is measured by evaluating the extent to which all sections and subsections are contained within the table of contents. The reasoning behind this is that if a topic is important enough to warrant a separate section within the documentation, then an entry in the table of contents is warranted as well.

Sections and subsections can be counted by parsing the document set and counting each section/subsection heading. If possible, a hierarchy should be established, consisting of sections, followed by subsections, followed by sub-subsections, etc.

The measurement approach combines the degree of inclusion of sections and subsections with an applied weighting scheme to give more importance to more essential sections.

Proposed Document Quality Indicators

Property: Correctness of Index

What is measured: The accuracy with which the index cites locations of terms.

Rationale: A good index enables the user to selectively read a document by noting the locations of important topics within the text of the document. If an index misleads a user with respect to the location of a term within the text of a document, the user suffers losses of both time and confidence in the documentation.

Measurement Approach: Correctness is an assessment of the accuracy of the entries (i.e., if the index says **missile warning system** appears on page 22, does it?). Deductions are made for erroneous entries, perhaps accounting for the degree of error committed in each case.

A more important facet of correctness is the significance of each term presented at the location(s) specified by the index (i.e., is the term actually discussed, or does it just make a cameo appearance?). This can be assessed by determining whether the observed term is used as the subject of a sentence on the page noted within the index.

Each entry is checked for accuracy of location and citation significance at the location cited.

Proposed Document Quality Indicators

Property: Correctness of Table of Contents (TOC)

What is measured: The accuracy with which the TOC cites the section locations and titles.

Rationale: If a table of contents is to be of assistance to the user of a document, it must correctly guide the user to the location of the desired section. Inaccuracies of this nature cause wasted time and effort, and thereby diminishes the effectiveness of the TOC as a reference tool.

Measurement Approach: The following criteria for TOC correctness are evaluated:

- Does the section begin at the location specified by the TOC?
- Does the section heading concur with the title cited in the TOC?

These criteria are assessed by comparing the entries of the TOC with the document body for accuracy.

Property: Design Supported by Code

What is measured: The degree to which the design specifications are realized by the code (implementation).

Rationale: To accurately reflect the product it describes, each design specification must be realized by some code module(s).

This property can impact the objective of correctness in either a positive or a negative manner. If the design is inaccurately represented by the code, then the requirements that the design represents are not met, thereby negatively affecting product correctness. In the reverse direction, if all of the design specifications are realized by the code, then the resultant code should be as correct as the design.

Measurement Approach: The following sets are constructed in order to measure the existence of design support present in the code:

D_{des} = the set of design specifications that support some requirement(s), extracted from the design documentation. (See D_{used} in the description of the Design Utility indicator), and

D_{code} = the set of design specifications found to be supported by the code.

These two sets are intersected to determine the extent to which the code accurately represents the design. Only utilized design specifications are considered for D_{des}; no credit is or should be given for code in support of superfluous design.

Proposed Document Quality Indicators

Property: Design Utility

What is measured: The extent to which design specification(s) fulfill a requirement specification.

Rationale: In the course of project development, requirements are changed, added, and deleted from the original set of requirements specifications. If a requirement is removed or changed, then the corresponding design specification(s) should be removed or changed to reflect this action. Design specifications that are not updated properly become a catalyst for confusion during both the implementation and maintenance stages of software development.

Measurement Approach: A set-based approach is used to assess this property. First, a set of all design specifications, D_{total} is constructed. Next, a subset of D_{total}, D_{used} is constructed. D_{used} contains only those design specifications that support the development of some requirement specification(s) (i.e., the requirement supported must currently be valid).

The cardinalities of these two sets are used to calculate the proportion of the existing design specifications that have utility with respect to fulfilling requirements.

Proposed Document Quality Indicators

Property:	Percent Domain Coverage
What is measured:	The degree to which the items required by the documentation standard (i.e., sections, modules, documents) are present within the documentation.
Rationale:	Standards are defined with a specific purpose in mind: to insure that those items deemed "necessary and sufficient" are present in a product. Such items are necessary to insure that (a) the product can be utilized in a straightforward manner, and (b) all needed information is present. Therefore, all items required by the documentation standard should be present if the product is to be considered complete.
Measurement Approach:	Given a standard (e.g. DoD-STD-2167A), evaluate the documentation set for completeness of coverage relative to the items required by that standard.

Proposed Document Quality Indicators

Property: Format Appropriateness

What is measured: The suitability of data presentation style or *layout*, i.e., the the most proper choice and use of charts, graphs, tables, and other graphic presentation formats.

Rationale: The presentation methods used to convey information directly impacts the user's comprehension level. Selection of the best (worse) method to display information can enhance (hinder) document comprehension.

Measurement Approach: The appropriateness of tables, graphs, and charts is difficult to determine. Nonetheless, one method is to note situations in conventional text (i.e., prose) that could better be represented in an alternate format. For instance:

List - long sequence of <phrase, phrase, phrase,...>,
Table - list of <phrase,...value, phrase,...value,...>,
Graph - multiple occurrences of phrases of the form <FROM x TO y> within a passage, where x and y are numerical in nature.

The text is examined for the presence of such instances, and the ratio of inappropriate representations to all such representations is examined.

Proposed Document Quality Indicators

Property:	Format Consistency
What is measured:	The application of a consistent format over all items within the same class (e.g. an index is in the class of indices) and within or among document set(s),
Rationale:	Once an item format has been observed, the reader has a priori knowledge of what to expect for additionally related items. Usability is enhanced by the consistent use of a format.
Measurement Approach:	Key re-occurring items of software documentation are sampled and an intra-class comparison is made. The proportion of those items using a consistent format is calculated on a per class basis; each class is weighted by its overall contribution to document usability. These figures are then summed to determine the overall consistency measure with respect to formatting.

Proposed Document Quality Indicators

Property: Format of Table of Contents (TOC)

What is measured: The application of a formatting scheme to the TOC that conveys the maximum amount of information about the document as possible, while maintaining ease of use.

Rationale: The user of a document should be able to determine the content, hierarchical organization, and component length(s) of a document by perusing the TOC. The format used in presenting the information contained within a TOC directly affects the user's ability to perform the above actions.

Measurement Approach: The TOC is evaluated for the presence of the following attributes:

- use of keywords in section titles,
- use of meaningful indentation,
- use of different typefaces (including sizes) to convey hierarchical organization of section,
- use of section numbers in section entries, and
- use of a single column for page numbers.

Each of the above attributes is assigned a significance value, reflecting its relative importance in the TOC formatting scheme. If an attribute is determined present, its significance value contributes to the Format of TOC metric value.

Proposed Document Quality Indicators

Property:	Factual Consistency

What is measured: The degree of consistency across documents with respect to "facts" (i.e., file names, enumerations of items, etc.). Webster defines a fact as:

- an assertion, statement, or information containing or purporting to contain something having objective reality.

For purposes of this indicator, a "fact" is a definitive or specific statement/term that has a specific value associated with it.

Rationale: Differing values across documents implies inaccuracy, and hence, misleads the user. Conflicting values across documents can lead to confusion, poor assumptions, and lack of faith in other documents.

Measurement Approach: For each document section, create a table of "facts", where each fact is represented as a <variable, value> pair.

One measurement approach is to compare the sets of facts among related sections of the documentation (i.e., sections pertaining to the same requirement) as follows:

<u>Facts In Common (FIC)</u> = the set of facts shared across related sections,
<u>Consistent Meaning (CM)</u> = the set of all facts in FIC that have the same value, and where
<u>Factual Consistency</u> = cardinality of CM divided by the cardinality of FIC.

"Relatedness" of sections is determined by traceability links calculated through the Top Down Traceability indicator.

Proposed Document Quality Indicators

Property:	Textual Clarity
What is measured:	The ease with which a document is comprehended by the user.
Rationale:	If users are to effectively use documentation, they must be able to comprehend it.
Measurement Approach:	The *reading grade level*, a measure of the amount of education necessary to acquire an understanding of the documentation, is calculated through the use of an established readability evaluation formula or combination of formulas. This readability level is then compared with the reading level of the anticipated user group of the documentation.

Proposed Document Quality Indicators

Property: Glossary Completeness

What is measured: The degree to which the glossary contains appropriate entries.

Rationale: A good glossary enables the user to find definitions of terms specific to the project. Accordingly, the reader can browse a document without being unduly concerned about missing the definition of a term/acronym at its first usage. A complete glossary contains all explicitly defined terms that a reader might need (project specific terms, acronyms, abbreviations).

Measurement Approach: Completeness of the glossary is determined by checking if all project-specific terms, abbreviations and acronyms are defined in the glossary. This list can be compiled manually or through an automated analysis of the documentation (One possible location where such terms might be collected is in the entries of the table of contents; another location is the first sentences of paragraphs.). Deductions are made for each omitted entry, augmented by a term frequency weighting scheme.

Proposed Document Quality Indicators

Property:	Invariance of Concept
What is measured:	The extent to which the idea or meaning of each requirement is preserved at the design/code levels of documentation.
Rationale:	Differing meanings across documents implies that the meaning of a requirement has possibly been misconstrued. This can lead to a requirement that is unsupported, even though the documentation states otherwise.
Measurement Approach:	Related sections of documentation (i.e., sections related across requirements, design, and code) are checked to determine if the central idea of the requirement is preserved at successively lower levels of abstraction. To effectively assess invariance of concept, an "invariance of concept" measure is constructed to evaluate the similarity of related sections based on the common use of keywords.

Proposed Document Quality Indicators

Property: Keyword Consistency

What is measured: The degree to which keywords are used consistently across *all* portions of a document set.

A keyword is any word specific to a project, or to computer science in general (acronyms and abbreviations excluded).

Rationale: The use of keywords in a documentation set should be consistent with respect to their meaning throughout the documentation. Usability is greatly diminished if such is not the case. Effectively, the user can easily become confused by conflicting uses of a keyword within the documentation.

Measurement Approach: A list of keywords is constructed. Associated with each keyword is a collection of the contexts in which the keyword is used throughout the documentation text. Each context is compared for consistency of meaning with the other contexts in which the same keyword is used.

Proposed Document Quality Indicators

Property: Module Appropriateness (Module Apparentness)

What is measured: The suitability of the physical division of text modules at varying levels of magnitude (i.e., sections, documents, chapters) within the documentation.

Rationale: Usability of documentation is enhanced if the reader can determine information about the structure/content of a document through simple perusal, as opposed to searching for detail.

Measurement Approach: Documents should begin on a new page, paragraphs and sections should be delimited by a consistent number of blank lines, indentation or possibly selected typefaces. The measure shall account for the most important of these, spacing. A general rule is that as module level increases, so should spacing. This rule is reflected in the following evaluation criteria:

Text: double-spaced,
Paragraphs: 2 * Text = 4-spaced,
Sections: 2 * paragraphs = 8-spaced (6-spaced acceptable), and
Documents, Large Sections: New page.

Proposed Document Quality Indicators

Property: Order of Glossary

What is measured: The use of an alphabetical ordering format in the glossary.

Rationale: A good glossary enables the user to find the definitions of terms specific to the project. This enables a reader to scan the document selectively without having to be unduly concerned about missing the definition of a term/acronym. The order of the glossary should be alphabetical because such a format greatly enhances the search process by paralleling the intuition of users.

Measurement Approach: The glossary is examined to determine if alphabetical ordering is employed. If some other ordering scheme is employed, the evaluation is altered accordingly.

Proposed Document Quality Indicators

Property:	Order of Index
What is measured:	The use of an effective ordering scheme in the index.
Rationale:	A good index enables the user to selectively read a document by noting the location of important topics within the text of the document. The format of an index should contribute to document usability by providing efficient and convenient access to important terms/topics within a document. The ordering scheme of an index assists in determining the extent to which effective access can be achieved.
Measurement Approach:	The order of the index should be alphabetical because an alphabetical format enhances the searching process. The use of an alphabetical scheme is evaluated, including the possibility of nested lists under general topics contained within the index. There may be other effective means of ordering, and if encountered, these means will be evaluated for their utility.

Proposed Document Quality Indicators

Property: Redundancy Appropriateness - Inter-section (stand-alone quality)

What is measured: The appropriateness of the amount of information repeated within a section of the document.

Rationale: Certain background information should be present within each section of a document. This characteristic enables the user to selectively read a document without needing to re-read each of the sections that hierarchically precede the selected section. An exception exists when the background is very lengthy or involved, in which case references to the appropriate sections should be present.

Measurement Approach: Throughout the documentation, topics are identified and recorded along with the locations of where the topics are discovered Topics are then examined to determine:

- the total number of times a topic is repeated within a group of related sections, and
- the degree of repeated coverage for each topic relative to the rest of the topics in the documentation.

Using the above information, the appropriateness of the redundant information is assessed.

Proposed Document Quality Indicators

Property: Specification Refinement (Refinement Enunciation)

What is measured: The refinement of specifications from high level documents to low level documents.

That is, as specifications progress from requirements through design to code, one expects abstractions to be refined to less abstract terms.

Rationale: As the documentation proceeds from one level of specification to the next, more detail is expected; as lower level documents are used to construct the next, a more concrete level of specification is realized (e.g., design is used to create code).

Measurement Approach: A larger amount of detail necessitates larger quantities of text. Therefore, an indication of the increase in detail is determined by assessing the amount of text dedicated to describing the same topic from one level of specification to the next. The increase in text is compared to an expected increase, and the ratio of actual increase to the expected increase is calculated.

Proposed Document Quality Indicators

Property: Requirements Supported by Design

What is measured: The degree to which the requirements set forth in the system specification document are present in the design documentation.

Rationale: In order for a document to *accurately* reflect the product it describes, all requirements must be represented in the design documentation. A design that does not include all requirements increases the probability of product defects because inaccurate design documents will later be used to implement the product in the form of code.

Measurement Approach: The following sets are constructed in order to measure the existence of requirements support present in the design:

R_{req} = the set of all requirement specifications, extracted from the requirements documentation.

R_{des} = the set of requirement specifications found to be supported by the design documentation.

These two sets are intersected to determine the extent to which the design accurately supports the requirements. The current stage of the development life cycle is also considered by weighting incomplete support more heavily as the development life cycle progresses.

Proposed Document Quality Indicators

Property: Simplicity/Modularity

What is measured: The appropriate division or decomposition of documentation such that a single theme is expressed by each section/subsection of a document.

This concept should propagate in a bottom-up manner. That is, the theme of a section should be an aggregation of its subordinate sections.

Rationale: In reading a document, comprehension is enhanced if the reader has only a limited, small number of related topics to consider at a time; ease of reading is enhanced, thereby decreasing reading time.

Measurement Approach: Each section is ranked by its hierarchical level, as determined by the number of periods in its section number (i.e., section 5 has level 0, while section 5.1.4 has level 2). The sections occupying the least abstract (i.e., the highest numbered) level(s) are analyzed for simplicity by evaluating the number of topics covered. (High level sections containing more than one theme should be reevaluated for partitioning into separate sections.) Lower level sections should be composed of topics related to the subordinate sections. Modularity evaluation determines the extent to which information in a given module can be attributed to topics in its subordinate modules.

Proposed Document Quality Indicators

Property: TBD/TBS (To Be Defined/To Be Specified)

What is measured: The amount of references to a later time in project development (e.g. "to be defined ...", "to be specified ...") that occurs within the documentation.

Rationale: The existence of TBDs and TBSs decreases the ability of a user to determine necessary information from the documentation because references leave gaps in the documents.

Measurement Approach: In order to assess this quantifier, phrases of the form "TBD/TBS" must be counted.

TBD/TBS statements are appropriate at many times during the development of software documentation, but the ones that remain unresolved over time detract from the accuracy, completeness, and usability of the documentation. The *persistence* of TBD/TBS phrases from one baseline to the next is an indicator of the non-resolution of information. Such persistence is measured by comparing the number of TBD/TBS phrases in the current documentation and comparing this to the number of TBD/TBS phrases in the corresponding sections from the previous baseline documentation.

Proposed Document Quality Indicator

Property: Top-Down Traceability

What is measured: The degree to which requirements are designated in successively lower levels of specification from design through code.

Rationale: If the requirements cannot be followed through design to implementation, the usability of the documentation is diminished.

Measurement Approach: This measurement approach uses three sets as its basis. These sets are:

Base sets (determined from raw data):

r = requirements (r_i) set forth in system specification,
d = design specifications (ds_j) from design documentation, and
c = code modules (cm_k) used to implement design.

Derived sets (calculated from Base sets):

rd = requirements supported by design documentation. (r_i, ds_j) pairs,
dc = design specifications supported by code documentation. (ds_j, cm_k) pairs, and
rdc = rd $\text{JOIN}_{rd.ds=dc.ds}$ dc, or (r_i, ds_j, cm_k) triples, where (r_i, ds_x) is a member of rd, (ds_y, cm_k) is a member of dc, and $ds_j = ds_x = ds_y$, where $x = y$.

The set rdc represents requirements reflected by both design and code documentation.

Proposed Document Quality Indicators

Property:	Term Uniqueness
What is measured:	The extent to which (significant) terms are used in a document to convey the same meaning in similar contexts.
Rationale:	The association of multiple meanings with a term given similar contexts invites reader confusion, particularly when the term is used in an atypical (i.e., project specific) or unfamiliar manner.
Measurement Approach:	Each instance of a term is recorded, along with its meaning and a representation of its context (context here applies in a general sense, perhaps encompassing an entire section of a document in some cases). Comparisons across term instances to determine uniqueness of meaning within a context are made, and a calculation is derived. These calculations are then combined to render an overall assessment, reflecting the consistency with which a term is used in a particular context.

Proposed Document Quality Indicators

Property: Text Conciseness (Succinctness)

What is measured: The succinctness of the documentation text; or, "does the author use just enough text to completely cover the topic at hand, and no more?"

Rationale: Each document should state the necessary information as briefly and as succinctly as possible so that the effort required to retrieve information from the documentation is minimized. Verbosity hinders this process because the user must "cut through the fat" to comprehend the meaning of the passage.

Measurement Approach: A succinct passage of text is characteristically rich in meaningful terms. Such terms are usually keywords of a document. Thus, the proportion of keywords to "noise" words indicates the succinctness of a passage.

A further indication of succinctness (or lack thereof) is the frequency with which passive voice is used. Passivity in a document detracts from succinctness, and should be avoided.

The two characteristics mentioned above are measured and combined to provide an indication of the succinctness of the documentation.

Appendix D

Measures for Ada Specific Indicators

(Metrics not included)

Each code indicator presented in this Appendix has the following format:

Property:	The measurable code characteristic used in computing a measure reflective of its direct relationship to desired software engineering attribute.
Attribute:	The software engineering attribute being assessed.
Rationale:	The reason(s) the assessment of this particular property is germane to the assessment of the software engineering attribute.
Measurement Approach:	The approach taken to evaluate the code/attribute relationship. Discussion includes the data collected, the methods used to collect the data, and the way(s) the data is used to render an evaluative measure of the property in question.
Metric:	Although preliminary metrics for the following code indicators have been formulated, they are not included in this report because they are still being refined and, subsequently, are not finalized.

Property:	Use of Record Discriminants
Attribute:	Complexity (+)
Rationale:	A factor in reducing complexity is the ability to use abstractions. Record discriminants encourage the accurate modeling of data types and allow more succinct abstractions to be made than are possible using "conventional" records.
Measurement Approach:	The measure for this indicator is the number of discriminant types defined relative to the number of user defined types defined. (Per Unit)

Property:	Use of Record Discriminants
Attribute:	Early Error Detection (+)
Rationale:	Implementing record discriminants requires detailed data-structures earlier in the life cycle, exposing weaknesses in previous documents.
Measurement Approach:	The measure for this indicator is the number of discriminant types relative to all user defined types. (Per Unit)

Proposed Code Indicators

Property: Use of Block Statements

Attribute: Cohesion (+)

Rationale: Block statements are an additional control structure provided by Ada. By binding code into control structures (block statements are one such control structure), the code cohesion is improved because such bindings usually imply that statements are functionally related.

Measurement Approach: This indicator utilizes global program measures as well as measures at the subprogram level. All subprograms are examined to determine an average lines of code per blocking structure across the entire program. From this the expected number of blocking structures for individual subprograms can be computed. The measure uses two ratios: (1) the expected number of blocking structures divided by the actual number of blocking structures tempered by (2) the total lines of code enclosed by blocking structures divided by the total lines of code of the subprogram. (Per Unit, excluding Tasks)

Property: Use of Block Statements

Attribute: Complexity (+)

Rationale: Block statements are a control structure provided by Ada. By using block statements to break code into smaller units, the complexity of the code is reduced.

Measurement Approach: This indicator utilizes global program measures as well as measures at the subprogram level. All subprograms are analyzed to determine an average lines of code per blocking structure across the entire program. From this measure, the expected number of blocking structures for a particular subprogram can be computed. The measure uses two ratios: (1) the expected number of blocking structures divided by the actual number of blocking structures tempered by (2) the total lines of code enclosed by blocking structures divided by the total lines of code of the subprogram. (Per Unit, excluding Tasks)

Property: Use of Block Statements

Attribute: Readability (+)

Rationale: Block statements are an additional control structure provided in Ada. Block statements aid readability by partitioning code into smaller related units. This partitioning results in smaller blocks of code that need to be considered as a unit by the reader, hence improving the understandability and readability of the code.

Measurement Approach: The measurement approach for this indicator utilizes a measure across the entire program as well as measures at the subprogram level. All subprograms are analyzed to determine an average lines of code per blocking structure across the entire program. From this measure, the expected number of blocking structures for a particular subprogram can be computed. The measure uses two ratios: (1) the expected number of blocking structures divided by the actual number of blocking structures tempered by (2) the total lines of code enclosed by blocking structures divided by the total lines of code of the subprogram. (Per Unit, excluding Tasks)

Property: Use of Both Default Parameters and Positional Notation

Attribute: Complexity (-)

Rationale: A parameter list in positional notation that has utilized a default parameter gives the reader the impression that something has been left out. This adds to confusion, and hence, complexity.

Measurement Approach: The measure for this indicator is the number of omitted parameters relative to the total number of parameters possible in all calls using positional notation. (Per Unit)

Proposed Code Indicators

Property:	Use of Both Default Parameters and Positional Notation
Attribute:	Readability (-)
Rationale:	It is more difficult to read a program containing calls utilizing positional notation that have omitted parameters. It appears to the reader that something has been left out.
Measurement Approach:	The measure for this indicator is the number of omitted parameters relative to the total number of parameters possible in all calls using positional notation. (Per Unit)

Property:	Definition and Use of Default Parameters for Stable Values
Attribute:	Complexity (+)
Rationale:	Parameter values that remain relatively stable for the entire program can be given default values. By defining and using a default value for a parameter, there is less information for the programmer to keep up with, hence, reducing complexity.
Measurement Approach:	The measure for this indicator is the number of subprogram calls where defaults are defined and used, relative to the number of subprogram calls where default values are possible. (Per Unit)

Property:	Definition and Use of Default Parameters for Stable Values
Attribute:	Readability (+)
Rationale:	Reducing the amount of superfluous information benefits the reader of a program. By using a default parameter value for stable values, the reader need only consider the parameter when necessary.
Measurement Approach:	The measure for this indicator is the number of subprogram calls where defaults were defined and used relative to the number of subprogram calls where default values were possible. (Per Unit)

Property:	Definition of Default Parameters
Attribute:	Well-Defined Interface (+)
Rationale:	When default values are defined, it conveys additional information about the module's interface This improves the clarity of the interface.
Measurement Approach:	The measurement approach for this indicator is the number of parameters defined with defaults relative to the total number of parameters. (Per Unit)

Proposed Code Indicators

Property: Use of Parameters with Name Notation

Attribute: Complexity (+)

Rationale: Because of the self documenting feature of subprogram calls using name notation, the overall complexity of the program is reduced.

Measurement Approach: The measure for this indicator is the average number of parameters from subprogram calls using name notation and having more than five parameters. (Per Unit)

Property: Use of Parameters with Name Notation

Attribute: Readability (+)

Rationale: The self documenting feature of subprogram calls using name notation inherently improves the readability of the program.

Measurement Approach: The measure for this indicator is the average number of parameters from subprogram calls using name notation with more than five parameters. (Per Unit)

Property:	Mixing the Order of Parameter Lists
Attribute:	Complexity (-)
Rationale:	Mixing the order of the parameter lists contributes to incomprehensibility and confusion. This adds to complexity.
Measurement Approach:	The measure for this indicator is the number of subprogram calls where the order differs from the subprogram specification relative to the total number of subprogram calls. (Per Unit)

Property:	Mixing the Order of Parameter Lists
Attribute:	Readability (-)
Rationale:	Mixing the order of the parameter lists is confusing to the reader and hence, reduces the readability of the program.
Measurement Approach:	The measure for this indicator is the number of subprogram calls where the order differs from the subprogram specification relative to the total number of subprogram calls. (Per Unit)

Proposed Code Indicators

Property: Use of Both Positional and Name Notation in one Subprogram Call

Attribute: Complexity (-)

Rationale: Using both forms of parameter notation in one subprogram call is confusing and hence, increases the overall complexity of the program.

Measurement Approach: The measure for this indicator is the number subprogram calls using mixed notation relative to the total number of subprogram calls. (Per Unit)

Property: Use of Both Positional and Name Notation in one Subprogram Call

Attribute: Readability (-)

Rationale: Using both forms of parameter notation in one subprogram call is confusing to the reader and hence, reduces the overall readability of the program.

Measurement Approach: The measure for this indicator is the number subprogram calls using mixed notation relative to the total number of subprogram calls. (Per Unit)

Property: Overloading of Subprogram Names
(Does not Include Overloading through Generics)

Attribute: Readability (-)

Rationale: Overloading of subprogram names reduces readability as the number of overloaded names increases. One subprogram name that refers to two or more subprogram definitions causes understandability problems in two main areas: (1) differences in the parameters of the subprograms and (2) differences in the semantics and functionality of the subprograms.

Measurement Approach: The measurement approach for this indicator is the number of unique overloaded subprogram names relative to the total number of unique subprogram names. Intuitively, readability is extremely low if half of the subprogram names are overloaded. Therefore, a factor of two is used in the measure. (Global)

Property: Definition of Declaration Packages

Attribute: Cohesion (+)

Rationale: One use of packages is to group related constants and types (i.e. declarations). This grouping is logically cohesive (related items) and procedurally cohesive (physically in one package).

Measurement Approach: Program units that "with" a declaration package actually reference a percentage of the declarations. A high percentage (or utilization) is indicative of very related items because they were necessary and used in the same program unit. The measure for cohesion is the average utilization for all declaration packages in the program. (Per Declaration Package)

Proposed Code Indicators

Property: Definition of Declaration Packages

Attribute: Ease of Change (+)

Rationale: The isolation of declarations into one cohesive package aids ease of change because units that may be impacted are easily identified.

Measurement Approach: The measurement approach for this indicator is to measure the reduction of the scope (percentage of units that must be checked) and weight the reduction by the significance of the scope reduction (percentage of code that must be checked). (Per Declaration Package)

Property: Insufficient Decomposition of a Declaration Package

Attribute: Ease of Change (-)

Rationale: Insufficient decomposition of declaration packages results in more work during maintenance activity. Program units must be unnecessarily checked for possible impacts caused by changes to declaration packages.

Measurement Approach: Declaration package utilization is a measure for the degree to which declarations in a package are utilized. It is also a measure for the percentage of items that must be checked for possible changes that actually need to be checked. Temper this measure by the significance of the package (total number of declarations relative to all declaration packages) to obtain a measure for ease of change. (Global)

Proposed Code Indicators

Property:	Definition of Packages that export Subprograms
Attribute:	Cohesion (+)
Rationale:	The isolation and localization of related subprograms into one package provides for logical, functional, and procedural cohesion. Functional cohesion is the strongest, and hence, the most important.
Measurement Approach:	To measure the cohesiveness of packages that export subprograms, relate the utilization of the subprograms by "withing" units. Intuitively, if the subprograms are sufficiently related, the majority of "withing" units will need to use most of the subprograms. (Per Subprogram Package)

Property:	Definition of Packages that export Subprograms
Attribute:	Well-Defined Interface (+)
Rationale:	A package provides an interface to the subprograms it exports through the package specification. All "withing" units must use this interface to use any of the subprograms. Other than subprogram specifications, the package specification may contain constants, type declarations, variables, and exceptions.
Measurement Approach:	One factor in assessing the quality of a package interface is the number of objects being exported. The objects considered are variables and subprogram specifications. The clarity of the interface is directly related to the Hrair limit, i.e. five subprograms is the optimal number. Because of the damaging effect of global variables, zero is the optimal number of variables in the package specification. The second factor is global variable references external to the package. It is through these references that information flow occurs that is not documented in the package specification. Due to the major impact of these references, their weight will be twice that of the first factor. (Per Subprogram Package)

Proposed Code Indicators

Property: Definition of Packages that export Subprograms

Attribute: Ease of Change (+)

Rationale: By grouping subprograms together in packages, it is possible to localize code for a group of functionally related subprograms. This creates modularity, resulting in an improvement for ease of change. The placing of the logically related subprograms benefits ease of change by isolating implementation details, thus reducing the "ripple effect."

Measurement Approach: The measurement approach for ease of change is to measure the total number of subprograms defined in subprogram packages relative to the total number of subprograms in the program. The significance of these subprograms is weighted by the significance of the subprograms (TLOC of the subprogram packages relative to TLOC). (Global)

Property: Units which "with" Packages that export Subprograms

Attribute: Complexity (+)

Rationale: "Withing" packages that export subprograms reduces the overall complexity of the program by reducing the amount of work associated with the program. Instead of having to look at the body of the package, a user need only look at the package specification to use any subprogram exported by the package.

Measurement Approach: The measure for the reduced work associated with a unit is the total lines of code of the "withed" package specifications relative to the total lines of code of the "withed" package bodies. (Per Subprogram Package)

Proposed Code Indicators

Property:	Units which "with" Packages that export Subprograms
Attribute:	Readability (+)
Rationale:	"Withing" packages that export subprograms benefits readability. This benefit occurs because there is simply a subprogram call rather than a subprogram definition and a subprogram call. Every distinct subprogram call to a subprogram from a package benefits the readability of the code.
Measurement Approach:	The measurement approach for this indicator is the number of distinct subprogram calls to subprograms from packages relative to the total number of distinct subprogram calls. (Per Subprogram Package)

Property:	Definition of Packages that are never "withed"
Attribute:	Complexity (-)
Rationale:	Defining packages that are never "withed" adds to the amount of code in the program without adding utility. This adds to the work associated with the program, or the program complexity.
Measurement Approach:	The measure for this indicator is the number of packages that are never "withed" relative to the total number of packages. (Global)

Property: Definition of Parameterless Generics

Attribute: Complexity (-)

Rationale: A parameterless generic provides a method of copying a group of objects. Because the generic has no parameters; multiple, duplicate copies of the objects are available. This is confusing and increases complexity.

Measurement Approach: The measure for this indicator is gives a value of three if the generic unit is parameterless and a five otherwise. (Global)

Property: Use of Subprograms as Generic Parameters

Attribute: Complexity (-)

Rationale: Ada permits subprograms to be passed as parameters to generic units. The actual generic unit is confusing because the programmer must remember that the formal subprogram name is really a parameter to be specified by the generic instantiation. The functionality of the subprogram is unknown and may have side effects. This makes programming the generic harder, hence, increasing complexity.

Measurement Approach: The measurement approach for this indicator uses the number of subprogram parameters relative to the total number of parameters to the generic. This ratio is multiplied by a factor of two because of the severe impact of this indicator. (Per Generic Unit)

Proposed Code Indicators

Property: Use of Subprograms as Generic Parameters

Attribute: Readability (-)

Rationale: Ada permits subprograms as parameters to generic units. The reader sees a subprogram name that is only a place holder for the real subprogram. Because of the unknown functionality and nondescript name, it is difficult for the reader to understand.

Measurement Approach: The measurement approach for this indicator uses the number of subprogram parameters relative to the total number of parameters to the generic. This ratio is multiplied by a factor of two because of the severe impact of this indicator. (Per Generic Unit)

Property: Definition of a Generic Unit (One Subprogram or One Package)

Attribute: Complexity (+)

Rationale: When identical algorithms differ only by type, it is less complex to abstract out the types and create templates of program units. Defining Ada generic units creates less complex units due to an additional layer of abstraction being made.

Measurement Approach: The measure for this indicator is the ratio of subprograms defined within a generic unit relative to the total number of subprograms defined. This ratio is multiplied by a factor of two because of the following argument. It is intuitive that if half of the subprograms are defined within a generic unit, the program has much of its complexity reduced because of the abstractions made through the definition of generic units. (Global)

Proposed Code Indicators

Property:	Definition of a Generic Unit (One Subprogram or One Package)
Attribute:	Readability (+)
Rationale:	Readability of the program is improved by abstracting out types from algorithms, that creates type independent algorithms. For example, an exchange algorithm can be written independent of the type. Readability and clarity is improved by having only one algorithm instead of multiple copies of similar algorithms.
Measurement Approach:	The measure for this indicator is the ratio of subprograms defined within a generic unit relative to the total number of subprograms defined. This ratio is multiplied by a factor of two because of the following argument. It is intuitive that if half of the subprograms are defined within a generic unit, the program's readability has improved because of the abstractions made through the definition of generic units. (Global)

Property:	Multiple Instantiations of a Generic Unit (One Subprogram or one Package)
Attribute:	Ease of Change (+)
Rationale:	The benefit to ease of change with regard to generics increases as the number of generic instantiations increases. A generic with only one instantiation has no benefit to ease of change, whereas a generic with five instantiations achieves beneficial results with respect to ease of change. Only one program unit need be changed instead of five.
Measurement Approach:	The measure for this indicator is the average number of instantiations over all defined generics. (Global)

Proposed Code Indicators

Property: Use of Tasking

Attribute: Complexity (-)

Rationale: Concurrent programs are more difficult to write and debug. This results in increased program complexity.

Measurement Approach: Each subprogram unit that communicates with a task will increase the overall program complexity. The measure for this indicator is the percentage of subprograms that have rendezvous. (Global)

Property: Definition of Tasks

Attribute: Readability (-)

Rationale: The readability of a tasks is directly related to its communication with subprograms and other tasks. The number of accept statements defined in a task gives the number of communicational entry points into the tasks. The more of these communicational entry points, the less readable the task becomes.

Measurement Approach: The measure for this indicator is the number of accept statements in the task body. The optimal value for this measure is three or less with an increase in the number of accept statements corresponding to a decrease in the measure for readability. (Global)

Proposed Code Indicators

Property:	Use of Rendezvous with In Out or In and Out Parameters
Attribute:	Coupling (-)
Rationale:	To achieve low coupling of modules, it is desirable to have minimal communication. The smallest feasible exchange of communication is in one direction per rendezvous. (Information is given to the task <u>or</u> information is retrieved from the task). When the information flow is in both directions during a rendezvous, the task and the unit requesting the rendezvous are excessively coupled.
Measurement Approach:	The measure for this indicator is the number of entries defined with In Out or In and Out parameters relative to all defined entries. (Per Task)

Property:	Use of Unprotected Global Data Areas
Attribute:	Coupling (-)
Rationale:	A change to the data area by any one of the accessing units can potentially cause a "ripple effect " in any other accessing module. This is a commonly accepted form of common coupling. This is further aggravated by the fact that these units can access the global data area concurrently.
Measurement Approach:	The measure for this indicator is the average number of units potentially concurrently coupled by a global data area. (Global)

Proposed Code Indicators

Property: Use of Unprotected Global Data Areas

Attribute: Ease of Change (-)

Rationale: A global data area is one that can be updated by concurrent units. This concurrent aspect is another hindrance that must be dealt with during a change to any one of the concurrent units. With a sequential program, behavior does not deviate without changes to input, a concurrent program's execution is nondeterminant, and hence, repeatability may not be achievable.

Measurement Approach: The measure for this indicator is the average number of units with concurrent access to global data areas. (Global)

Property: Use of Exception Handlers

Attribute: Complexity (+)

Rationale: The use of exception handlers separates the algorithm from error handling code. This produces two distinct code sections, an algorithm section and an error handling section. This clarity reduces the complexity of the product.

Measurement Approach: The measure for this indicator is the number of subprograms with an exception handling section relative to the total number of subprograms. This number is weighted by a factor of two because of the following argument: using exception handlers with at least half of the subprograms is extremely beneficial. (Global)

Property:	Relying on Upper Level Subprograms to Handle User Raised Exceptions
Attribute:	Coupling (-)
Rationale:	A user raised exception should have a corresponding exception handler in the same subprogram. If an exception handler does not exist, the exception is propagated to upper level subprograms. This couples the subprograms via the exception propagation.
Measurement Approach:	The measure for this indicator is the number of subprograms that raise errors, but do not handle them. This measure is not made relative to the total number of subprograms because the presence of the indicator is simply inherently detrimental. (Global)

APPENDIX E

Ada CODE ANALYZER

USERS MANUAL

(Preliminary Draft)

Ada Code Analyzer

Users Manual

(Preliminary Draft)

TABLE OF CONTENTS

1. Overview

Experience has shown that manually collecting data to compute metrics is labor intensive and error prone. In response to such experiences, this manual outlines the capabilities of an automated system for collecting data and computing quality metrics related to Ada products. The report generation system presented consists of three software tools that examines Ada source code and produces a report that details the extent to which desirable software engineering attributes are present in the examined code. The reported measures are based on metrics developed by Bundy [BUNG90] for the Software Quality Assessment project with oversight by Systems Research Center personnel at Virginia Tech.

The data collection and analysis system is designed to separately address language issues and metric computations. The intent of this design decomposition is to promote extensibility. As illustrated in Figure 1, one can collect data from several sources, and then use the same report generator to compute metric values and to analyze the data. The three components of the current system are:

- the Ada Analyzer - ADALYZE,
- the Data Extractor - DEX, and
- the Report Generator - RGEN.

ADALYZE is a language dependent analyzer that accepts Ada programs as input and produces data items that support metric computations. Effectively, ADALYZE is a "compiler" that generates several language specific data sets. In particular, these data sets contain information reflecting Ada language characteristics, e.g., the number of packages that export subprograms. The language specific data sets are then passed to a data extractor (DEX). The function of DEX is to transform the language specific data sets into a more universal format, i.e., one that is primarily language independent. Because the data sets created by DEX are language independent only one metric computation and analysis program is needed, regardless of the language being analyzed. Figure 1 illustrates how the data collection and computational processes proceed when components of more than one language are to be analyzed.

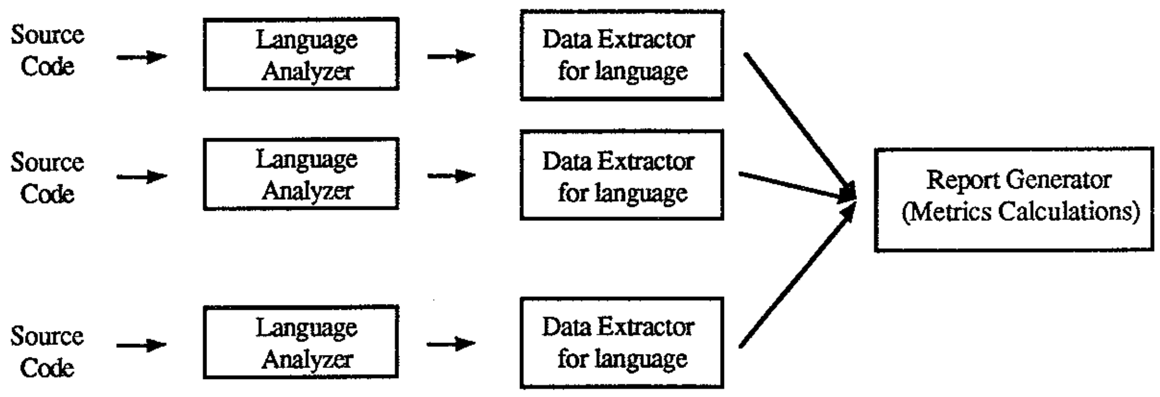

Source Code
Language Analyzer
Data Extractor for language
Source Code
Language Analyzer
Data Extractor for language
Source Code
Language Analyzer
Data Extractor for language
Report Generator (Metrics Calculations)

Figure 1
Overall View of Information Flow

Structured to exploit knowledge of required metric computations, the report generator (RGEN) reads the language independent data sets (or files) created by the data extractor and produces (a) a summary of the input data elements and (b) a report reflecting the extent to which desirable attributes are present in the product being examined. Figure 2 illustrates a more detailed view the relationship among the computational components and depicts how information flows between those components. The remainder of this manual provides a detailed discussion of the three system components.

2. Basic System Requirements

ADALYZE is constructed by using a parsing grammar designed for ALEX and AYACC (Ada versions of LEX and YACC). The output of ALEX and AYACC are Ada source files that, when combined with semantic routines (also written in Ada), comprise ADALYZE. Working copies of ALEX and AYACC are not essential to the operation of ADALYZE unless one intends to change its functionality. What is required, however, is an Ada compiler to translate the source code for all three programs (ADALYZE, DEX and RGEN) into target code for the

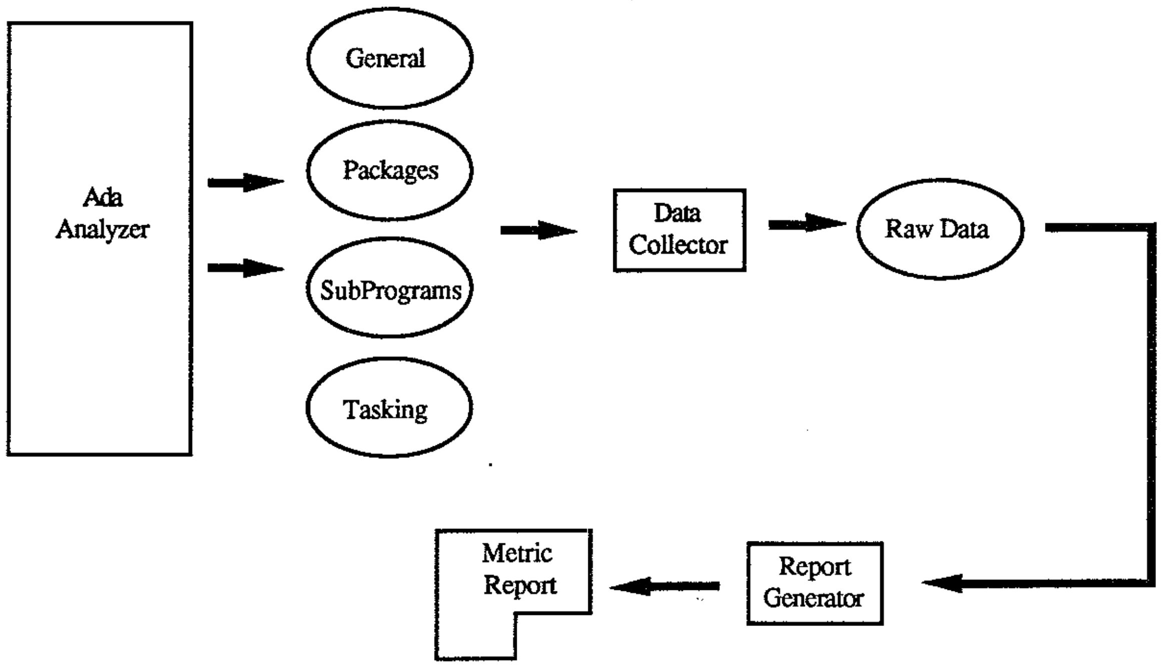


Figure 2
Detailed Diagram of Information Flow in the Ada Analyzer and Report Generator

host system.

The amount of virtual memory required by the current implementation of ADALYZE depends on the size of the source code being analyzed. In particular, to reduce execution time ADALYZE maintains a memory resident symbol table for the complete Ada program being analyzed. DEX and RGEN, on the other hand, do not depend on dynamic memory allocation, and subsequently, require only minimal amounts of memory. As an example, ADALYZE requires approximately 20 meg of virtual memory to process 30,000 lines of Ada source code. On a multiuser system, it took about three hours to complete. With the same input, DEX produces data files totaling approximately 62,000 bytes; the report generator, RGEN, produces a two megabyte report file.

For execution purposes, the authors recommend that the user identify an automated procedure for constructing a compilation order file. This file, which is the input to the Ada analyzer, specifies the compilation order for files containing the software system to be analyzed. For small systems, the compilation order file can be created by hand. For large systems, however, a manual process is extremely difficult, if not impossible. Most Ada compilers provide a utility for determining the compilation order of a system. The Ada Software Repository has public domain software that supports such capabilities.

Currently, all three programs used command line arguments supported by the use of a packages provided by VADS (Verdix Ada Development System). To maintain this feature during a software port, similar routines must be provided by the compiler on the target system under which the analyzer system is to be compiled and executed.

3. The Ada Analyzer - ADALYZE

When the analyzer is invoked it reads a compilation order file that contains a list of file names to be examined by the analyzer and the order in which the examination is to proceed. This file is called the compilation order file (CO file) and should have the extension **.co**. When specifying the name of the CO file the **.co** extension must be given. The CO file name can be given as a command line argument or as a response to a prompt from ADALYZE.

Four files are produced as output by ADALYZE, each containing data specific to General, Subprogram, Package and Task structures. The names of these files are constructed from the base name of the CO file and extensions **.gen**, **.subprg**, **.pkg** and **.task**, respectively. The format of these data files vary but have the same basic structure. Each output file is in ASCII format and contains a set of data items placed on a separate line. In most cases the data item is a number followed by a description; other times it is simply a name (e.g. a subprogram name). Each subprogram, package and task files contain a general section followed by one section of data for each unit being examined. For example, a subprogram file contains a general section recording global data, like the number of subprograms and the parameter passing methods, followed by separate sections that record information particular to individual subprograms. Both the contents and the naming of files generated by ADALYZE reflect an Ada bias, that is, they incorporate Ada terminology. They must , therefore, be preprocessed by the Data Extractor (DEX) before being read by the Report Generator (RGEN). Attachment A provides an outline of the files created by ADALYZE.

4. The Data Extractor - DEX

The purpose of the data extractor is to convert language specific data files into a language independent format. By using files that are language independent, only one report generator needs to be written. DEX accepts as input the files generated by the Ada analyzer and reorganizes them into four distinct files having the extensions: **.prg**, **.sub**, **.eu** and **.cu**, denoting Program, Subprogram, Encapsulation Unit, and Concurrent Unit, respectively. As illustrated in Attachment D, DEX also accepts two command line arguments: the base name of the input files and the base name of the output files. If a command line error is encountered, DEX responds by displaying the proper syntax for the command line and terminates execution.

DEX produces output files that have the same information as its input files, but in a re-organized format. These output files consist of unit names and decimal numbers. Attachment C provides a sample input file to DEX and a sample output file.

5. The Report Generator - RGEN

Using the language independent files produced by DEX, RGEN creates a report that summarizes the input data, computes and prints metric computations, and provides set of measures reflecting aggregated metric and attribute values. RGEN expects two command line arguments: (1) the base name of the files created by DEX and (2) the name of an indicator file. Contents of the indicator file are text strings that describe data elements used to compute metrics, e.g. number of if statements. Attachment D illustrates the proper syntax for the command line arguments.

As illustrated in Figure 3, when RGEN initiated, it first prompts the user for a name for the output file. RGEN also solicits four (4) options. The first option asks if the input data is to be echoed to an output file. The second option allows the user to specify whether the reporting is to use Ada specific terminology or semantically equivalent generic terminology, e.g., package rather than encapsulation unit. The third option asks the user if metrics that have been given default values are to be included in the aggregation report. (Default metric values are assigned when

```
> a.out data indicators.txt

Enter name for output file
report1.out
File already exists.  Overwrite? (y or n) y

OPTIONS:        enter y or n after each
----------

Echo raw data --> y
Use Ada-specific terminology --> n
Include default metric values in analysis --> n
Change weights for subprg, encap unit, and concur unit --> n
```

Figure 3
A Sample RGEN Run

computations require a data item that is missing.) Finally, the fourth option allows the user to modify the default weightings given to the subprogram, encapsulation unit, and concurrent unit parts when total program statistics are calculated. For example, when computing program statistics, the subprogram part, encapsulation unit part, and concurrent unit part are weighted by the number of unit types. The user can specify an additional weighting by responding 'y' to the fourth option and, as illustrated in Figure 4, by providing the appropriate augmented weightings.

Currently, the output of RGEN is organized into three major sections. As illustrated in Attachment C, the first section is an echo of the input data, organized by unit type (e.g. subprogram data first, followed by encapsulation unit data and then concurrent unit data.) The second output section lists individual metric computations and their contributions to the achievement of desirable software engineering attributes. Note, reporting in this second section is still on a per unit basis within unit type. The third sample format provides attribute values reflecting an average of contributing metric values.

The three output forms shown in Attachment C are selected samples and represent only part of the information that is actually provided by RGEN. Figure 5a provides an outline of the different levels at which data is echoed.

```
> a.out data indicators.txt

Enter name for output file
report1.out
File already exists.  Overwrite? (y or n) y

OPTIONS:        enter y or n after each
---------

Echo raw data --> y
Use ADA-specific terminology --> n
Include default metric values in analysis --> n
Change weights for subprg, encap unit, and concur unit --> y
      Enter weights as decimal numbers
           Subprogram --> 1.5
           Encapsulation Unit --> 1.0
           Concurrent Unit --> 0.5
```

Figure 4:
A Sample RGEN Run Changing Unit Weights

That is, echoed data is reported by individual units within each of the unit types: subprogram, packages, tasks, and global data. Similarly, Figure 5b conveys the same organization relative to the reporting of individual metrics, while Figure 5c provides an outline of the how the attribute values are reported. Note in Figure 5c that computational values are presented on a per unit basis within each unit type, but then are aggregated to form unit type averages and finally, a program average. Lines of code are used as a weighting factor to average together units within unit types; the number of units within each type is used to weight the contribution of each unit type relative to the total program attribute values.

```
Per Subprogram Data
Per Subprogram Data Totals
Per Encapsulation Unit Data
Per Encapsulation Unit Totals
Per Concurrent Unit Data
Per Concurent Unit Data Totals
Global Data
     Subprogram
     Encapsulation Unit
     Concurrent Unit
     General Program
```

Figure 5a
Organization of Echoed Data

Per Subprogram Metrics
Metrics on Per Subprogram Totals
Per Encapsulation Unit Metrics
Metrics on Per Encapsulation Unit Totals
Per Concurrent Unit Metrics
Metrics on Per Concurrent Unit Totals
Global Metrics
- Subprogram
- Encapsulation Unit
- Concurrent Unit
- General Program

Program Averages

Figure 5b
Organization of Metrics

Per Subprogram Analysis
Analysis on Per Subprogram Totals
Per Encapsulation Unit Analysis
Analysis on Per Encapsulation Unit Totals
Per Concurrent Unit Analysis
Analysis on Per Concurrent Unit Totals
Global Analysis
- Subprogram
- Encapsulation Unit
- Concurrent Unit
- General Program

Contributions by all Subprograms
Contributions by all Encapsulation Units
Contributions by all Concurrent Units
Contributions by Entire Program

Figure 5c
Organization of Attribute Value Reporting

On a final note, the authors point out that some metrics will have non-numeric values because of special circumstances. In particular, many metric calculations involve division. If the computed denominator has a value of zero, an "N/A" appears next to that metric's name to imply that the particular metric could not be calculated and is not included in any of the attribute computations. Additionally, several metrics have a default condition. For example, when computing the metric associated with "the number of parameters passed," a default value of five is given if no subprogram calls are detected. This case is denoted in the report as "5.00 DEF". Through a solicited response from RGEN, the user can choose to include or exclude the default metric values from attribute value computations. Finally, some metric values fall outside the range 0 - 10. When this occurs, either a "0.00*" or a "10.00*" is respectively printed; for statistical analysis purposes, the values of 0.00 and 10.00 are then used, not the actual values.

ATTACHMENT A

ADALYZE OUTPUT FILES

General Information - ADALYZE OUTPUT FILE (filename.gen)

- number of discriminant types defined
- number of user defined types
- TLOC
- total number of units (subprograms, packages,tasks)
- number of parameterless generics defined
- number of generics with subprogram parameters defined
- total number of generics
- total number of generic parameters
- total number of subprograms defined within generics
- total number of generic instantiations

Subprograms - ADALYZE OUTPUT FILE (filename.subprg)

General

- total number of subprogams which rendezvous (call an entry point)
- number of subprograms with an exception handler
- number of subprograms which raise exceptions not handled in that subprogram's exception handlers
- number of unique overloaded subprogram names
- total number of unique subprogram names

Subprograms - ADALYZE OUTPUT FILE (filename.subprg)

Per Subprogram

- Full Name
- number of formal parameters
- number of formal parameters with discriminants
- number of formal parameters with defaults
- Is a generic? (0=NO 1=YES)
- number of generic parameters
- number of generic parameters that are subprograms
- TLOC
- number of of Block Structures (BS) at level 0
- TLOC enclosed by BS at level >= 0
- TLOC enclosed by BS at level >= 3
- Maximum BS level
- total number of if statements
- number of unique non-local variables referenced
- number of unique local variables referenced
- number of unique parameters referenced
- number of parameters in calls made
- number of unique parameters passed in calls
- total number of parameters in all calls using only positional notation
- total number of parameters omitted in all calls using only positional notation
- total number of parameters from calls using positional notation with > 5 parameters
- total number of calls using positional notation with > 5 parameters
- number of parameters from calls using named notation with > 5 parameters
- number of calls using named notation with > 5 parameters
- total number of calls using positional notation
- total number of calls using named notation
- total number of calls using both parameter notations
- number of unique parameterless calls
- number of calls with defaults possible
- number of calls with defaults used
- number of unique calls
- total number of calls with ordering different from their subprogram specification
- number of calls made

Subprograms - ADALYZE OUTPUT FILE (filename.subprg) - CONTINUED

- number of unique parameters passed
- number of user defined types
- number of discriminant types defined
- number of structured data type (user defined) parameters
- number of unique structured data type references
- number of unique variable references
- number of gotos
- number of block comments
- number of block comment lines
- formal parameters used as switches
- total number of comment lines
- total number of single line comments
- number of distinct calls to subprograms in subprogram packages
- total number of distinct calls to all subprograms
- number of non-symbolic constants referenced
- number of subprogram packages "withed/used" by this the subprogram
- calculate the sum of (1 - (TLOC of the pkg spec) / (TLOC of the pkg)) for all subprg packages withed/used by this subprogram
- number of rendezvous requested (calls to entry points)
- number of calls made which contain switches
- number actual parameters used as switches

Packages - ADALYZE OUTPUT FILE (filename.pkg)

General

- number of declaration packages (compilation unit) which are never "withed"
- number of subprogram packages (compilation unit) which are never "withed"

Per Package

- Package Type (0=declaration; 1=subprogram)
- Full Name
- number of subprograms in the package body
  - listing of subprograms in package body. One per line.
- Is a generic? (0=NO 1=YES)
- number of generic parameters
- number of "withing/using" units to this Package
- TLOC of "withing/using" units
- total number of declarations
- total number of user defined types
- total number of discriminant types defined
- total number of referenced package declarations summed over all units that with/use this decl pkg
- number of generic parameters that are subprograms
- TLOC
- number of subprograms in the package specification
- number of global variables declared in the sub package specification
- number of distinct external global variable references found in the subprogram package
- TLOC of the package spec
- TLOC of the package body
- total number of referenced package subprograms summed over all units that with/use this subprogram package

Tasking - ADALYZE OUTPUT FILE (filename.task)

General

- number of accept statements for all tasks
- sum (the number of distinct global data areas accessed by a task) over all tasks
- total number of distinct global data areas accessed

Per Task

- Full Name
- TLOC
- total number of entries with **In Out** or both **In** and **Out** parameters
- total number of entries
- number of unique entry points (entries)

ATTACHMENT B

SAMPLE INPUT AND OUTPUT FOR DEX

SAMPLE INPUT FILE TO DEX
(Output file of Ada analyzer)
(Encapsulation Unit File)

```
  0  Number of declaration packages (compilation units never withed)
  0  Number of Subprogram packages (compilation units never withed)
  1  Packaged type; decl - 0   subprg - 1
 11  Total number of subprograms in package body
  0  Is a Generic; 0=no 1=yes
  0  Total number of generics parameters
  1  Total number of withing units to this pkg
432  TLOC of withing units
 54  Total number of declarations (spec)
 17  Total number of user defined types (spec).
  8  Total number of disc types defined
 66  Total number of refed pkg decls summed over all units that withed the decl pkg
  0  Total number of generic parameters that are subprograms
114  Total lines of code for package
  5  Total number of subprograms in package spec
  0  Total number of global variables declared in the spec of a subprogram package
  5  Total number of external variable refs in subprogram package
 54  Total lines of code in package spec
 60  Total lines of code in package body
  9  Total number of refed pkg subprograms summed over all units that withed the subprg pkg
```

SAMPLE OUTPUT FILE FROM DEX/ INPUT FILE TO RGEN
(Encapsulation Unit File)

```
 0.00
 0.00
 0.00
11.00
 0.00
 0.00
 1.00
50.00
 0.00
 0.00
 0.00
 0.00
62.00
 0.00
 0.00
60.00
 0.00
```

ATTACHMENT C

SAMPLE DATA OUTPUT FROM RGEN

Input Data Echoing
(Selected Data Items)

```
***** BEGIN SUBPROGRAM KIND_OF

number of formal parameters.................................................................... 1
number of formal parameters with discriminants.................................................. 0
number of formal parameters with defaults....................................................... 0
Is an instantiable object? (0=NO 1=YES)......................................................... 0
number of instantiable object parameters........................................................ 0
number of instantiable object parameters that are subprgs....................................... 0
TLOC............................................................................................ 3
number of BS at level 0......................................................................... 2
TLOC enclosed by BS at level >= 0............................................................... 2
TLOC enclosed by BS at level >= 3............................................................... 0
Maximum BS level................................................................................ 1
number of if statements......................................................................... 1
number of unique non-local variables referenced................................................. 0
number of unique local variables referenced..................................................... 0
number of unique parameters referenced.......................................................... 1
number of parameters in calls made.............................................................. 1
number of parameters in all calls using only positional notation................................ 1
number of parameters omitted in all calls using only positional notation........................ 0
number of parameters from calls using positional notation with > 5 parameters................... 0
number of calls using positional notation with > 5 parameters................................... 0
number of parameters from calls using named notation with > 5 parameters........................ 0
number of calls using named notation with > 5 parameters........................................ 0
number of calls using positional notation....................................................... 1
number of calls using named notation............................................................ 0
number of calls using both parameter notations.................................................. 0
number of calls with defaults possible.......................................................... 0
number of calls with defaults used.............................................................. 0
number of unique calls.......................................................................... 1
number of calls with ordering different from their subprg specification........................ 0
number of calls made............................................................................ 1
number of unique parameters passed.............................................................. 1
number of user defined types.................................................................... 0
number of discriminant types defined............................................................ 0
number of structured data type (user defined) parameters........................................ 0
number of unique structured data type references................................................ 0
number of unique variable references............................................................ 1
number of gotos................................................................................. 0
number of block comments........................................................................ 1
number of block comment lines................................................................... 3
formal parameters used as switches.............................................................. 0
number of comment lines......................................................................... 3
number of single line comments.................................................................. 0
number of distinct calls to subprgs in subprg encap units....................................... 0
number of non-symbolic constants referenced..................................................... 0
number of subprg encap units imported by this subprg............................................ 0
number of synchronizations requested (calls to entry points).................................... 0

***** END SUBPROGRAM KIND_OF
```

Per Unit Metric Computations
(Selected Metrics)

```
***** BEGIN SUBPROGRAM KIND_OF

***** ANALYSIS OF COUPLING

Number of global variables referenced.......................................    5.00
Number of parameters passed.................................................    9.00
Types of parameters passed..................................................   10.00
Number of structured data types passed as parameters........................    5.00

***** ANALYSIS OF COHESION

Number of blocking structures...............................................    2.56
Division of code into logical units performing single specific functions...    8.33

***** ANALYSIS OF WELL DEFINED INTERFACES

Number of global variables..................................................    5.00
Use of parameters in subprogram and entry calls.............................   10.00
Use of parameterless procedure calls........................................    5.00
Num of data structures used with respect to intra-routine communication....    0.00
Use of "excessive" number of parameters.....................................    5.00
Definition of default parameters............................................    5.00

***** ANALYSIS OF COMPLEXITY

Length of subprograms.......................................................   10.00*
Number of blocking structures used..........................................    2.56
Number of goto statements used..............................................    5.00
Number of block comments....................................................    1.00
Number of single line comments..............................................   10.00*
Number of structured data types used........................................    5.00
Number of if statements.....................................................    8.10
Use of both default parameters and positional notation......................    5.00
Definition and use of default parameters for stable values..................    N/A
Use of parameters with name notation........................................    5.00 DEF
Mixing the order of parameter lists.........................................    5.00
Use of both positional and name notation in one subprogram call.............    5.00
Units which "with" packages that export subprograms.........................    N/A
Use of record discriminants.................................................    N/A
Use of subprograms as generic parameters....................................    N/A

***** ANALYSIS OF READABILITY

Use of blocking structures..................................................    2.56
Use of goto statements......................................................    5.00
Use of block header comments................................................    1.00
Use of single line comments.................................................   10.00*
Subprogram length...........................................................   10.00*
Use of symbolic constants (literals)........................................   10.00
Use of both default parameters and positional notation......................    5.00
Definition and use of default parameters for stable values..................    N/A
Use of parameters with name notation........................................    5.00 DEF
Mixing the order of parameter lists.........................................    5.00
```

```
Use of both positional and name notation in one subprogram call.................................................. 5.00

***** ANALYSIS OF EASE OF CHANGE

Use of global variables........................................................................................................ 5.00
Use of distinct functions within a single module.......................................................................... 8.33

***** ANALYSIS OF EARLY ERROR DETECTION

Use of record discriminants.................................................................................................. N/A

***** END SUBPROGRAM KIND_OF
```

```
***** BEGIN SUBPROGRAM KIND_OF

ATTRIBUTE CONTRIBUTIONS:                                      LOW        HIGH
-------------------------------

COUPLING.............................................  7.25    5.00      10.00
COHESION.............................................  5.45    2.56       8.33
WELL DEFINED INTERFACES..............................  5.00    0.00      10.00
COMPLEXITY...........................................  5.67    1.00      10.00*
READABILITY..........................................  5.86    1.00      10.00*
EASE OF CHANGE.......................................  6.67    5.00       8.33
EARLY ERROR DETECTION................................  N/A     N/A        N/A

***** END SUBPROGRAM KIND_OF
```

ATTACHMENT D

COMMAND LINE ARGUMENTS FOR DEX AND RGEN

COMMAND LINE FORMATS - DEX AND RGEN

DEX:

```
> exec filename1 filename2
```

exec	-->	the executable file
filename1	-->	the base name of the 4 input files containing the data from the Ada analyzer
filename2	-->	the base name of the 4 output files created by DEX, which will supply each with the appropriate extension
EXAMPLE:		dex mp data

RGEN:

```
> exec filename1 filename2
```

exec	-->	the executable file
filename1	-->	the base name of the 4 input files containing the data from DEX
filename2	-->	the full name of the file containing the strings associated with the data
EXAMPLE:		rgen data indicators.txt

APPENDIX F

Site Selection Criteria

Prefatory Remarks

The first set of criteria is project related. The second set of criteria is somewhat general in nature, but represent highly desirable site capabilities from the standpoint of supporting an automated validation effort. The remaining sets of criteria are partitioned by phases within the software development life cycle, and then within each phase, further partitioned on a process/product classification.

(1) The attached list outlines an ideal set of criteria. We do not expect any one site to support all of the stated desirables. Moreover, the extent to which a site does or does not support our ideal set of criteria is not a measure of its capability to produce a quality product. It simply provides a yardstick by which we can assess a site's capabilities to support the SQA validation effort.

(2) We would like to stress that it is not our intention to interfere with the development process in any way. We hope to be as unobtrusive as possible in any setting and will work to minimize any negative impact that our presence might have.

(3) Although the investigative effort will focus on validating the predictive capabilities of the Objectives, Principles and Attribute framework, we anticipate significant benefits for all involved. Such benefits will stem from interaction with site personnel and through continuous feedback to the Project Manager.

(4) Although we have specified a project size of 10,000 to 50,000 lines of Ada code, size is not the crucial factor. More significant is project deployment within 18-20 months from the date that the project starts. By meeting this constraint the SRC/VPI personnel will have four to six months left (at the end of the projected three year effort) to instrument post-deployment activities and to collect additional validation data.

Site Selection Criteria (Ideal)

Project Level Criteria:

- ❏ New Initiative - We need to be involved from the start
- ❏ Ada Based
- ❏ Deployment within 18 months from project initiation date
- ❏ Relatively small effort (10,000 - 50,000 source lines of code)
- ❏ Logistically effective (Travel, On-Site Personnel)

Desirable Site Characteristics:

- ❏ Formal review processes (similar to those outlined in DOD-STD-2167A)
  - ❏ Requirements review
  - ❏ Preliminary design review
  - ❏ Critical design review
  - ❏ Code walkthroughs
- ❏ Employment of an (automated) configuration management system supporting
  - ❏ compilation of software development folders (SDFs)
  - ❏ tracking of trouble reports and specification changes
  - ❏ code modification (SCCS)
- ❏ Provision for and utilization of on-line documentation facilities
- ❏ A well defined procedure for transferring Configuration Management (CM) responsibilities at deployment time

Desirable Activities, Products and Participation Level by Software Development Phase

Requirements Specification Phase:

Desirable Process Activities:

- ❑ Formulation of software development plan
- ❑ Requirements specification
- ❑ Formal requirements review
- ❑ Activity logging for each (ideally, through an automated requirements specification system)

Desirable Products:

- ❑ Software development plan
- ❑ Requirements specifications
- ❑ Activity log

SRC/VPI Interaction Activities:

- Non-participatory (observer) involvement in major requirement specification activities
- Possible interviewing of personnel involved in process activity
- Access to activity log for data extraction (machine readable forms preferable)

Preliminary Design Phase:

Desirable Process Activities:

- ❑ Preliminary design
- ❑ Design review
- ❑ Logging of defects detected, corrective options, actions taken and justification

Desirable Products:

- ❑ Preliminary design documents
- ❑ Central repository of high level design activities (preferable computerized), e.g. design reviews and resultant actions
- ❑ The beginning of "Software Development Folders"

SRC/VPI Interaction Activities:

- Non-participatory (observer) involvement in major design activities
- Possible interviewing of personnel involved in design activities
- Examination of design documents and activity logs (machine readable forms preferable)

Detailed Design Phase:

Desirable Process Activities:

- ❑ Detail design
- ❑ Critical design review including design walkthroughs
- ❑ Logging of defects, modifications to design and specifications, modification options, and rationale for selected option

Desirable Products:

- ❑ Detail design document
- ❑ Log of actions taken in transforming high level design to detail design including critical design review, design walkthrough, etc. (an automated system to log such activities is preferable)
- ❑ Software verification plan
- ❑ Test requirements

SRC/VPI Interaction Activities:

- Non-participatory (observer) involvement in major activities surrounding the transformation of high level design to detail design
- Possible interviewing of personnel involved in activities
- Examination of design documents and activity logs (machine readable forms preferable)

Implementation Phase:

Desirable Process Activities:

- ❑ Identifying team(s) composition relative to implementation task (organizational structure)
- ❑ Code development in compliance with specified standards
- ❑ Structured code walkthroughs and code inspections
- ❑ Logging of changes to design and code (preferable through a configuration management system)

.Desirable Products:

- ❑ Source code
- ❑ History of source code modifications, detail design modifications, and rationale for changes.

SRC/VPI Interaction Activities:

- Non-participatory (observer) involvement in major activities surrounding implementation
- Examination of source code
- Examination of source code and design documentation modification histories

Testing Phase:

Desirable Process Activities:

- ❑ Test plan review
- ❑ Test procedures review
- ❑ Testing
- ❑ Logging of test results
- ❑ Acceptance review

Desirable Products:

- ❑ Test plan
- ❑ Test plan review
- ❑ Test procedure
- ❑ Test procedures review
- ❑ Results of test including unresolved issues, problems and generated software trouble reports

SRC/VPI Interaction Activities:

- Non-participatory (observer) involvement in major testing activities
- Examination of testing plans, procedures and results of the testing

Post-Deployment Maintenance Activity (may or may not be applicable to development site):

Desirable Process Activities:

- ❑ Recording of reported trouble reports as well as the tracking of open and closed trouble reports
- ❑ Configuration management activities logging code changes, documentation changes, and personnel time involved

Desirable Products:

- ❑ Activity log capturing elements of the above process activities

SRC/VPI Activities:

- Examination of activity log for the extraction of factors needed to validate predictive indicators

Post-Deployment Execution History (may or may not be applicable to development site):

Desirable Process Activities:

- ❑ Recording of reported trouble reports, and software execution history

Desirable Products:

- ❑ Activity log capturing elements of the above process activities

SRC/VPI Activities:

- Examination of activity log for the extraction of factors needed to validate predictive indicators

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE	Form Approved OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) Systems Research Center SRC-91-002	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Systems Research Center 6b. OFFICE SYMBOL *(If applicable)*	7a. NAME OF MONITORING ORGANIZATION Naval Surface Warfare Center
6c. ADDRESS *(City, State, and ZIP Code)* 320 Femoyer Hall Virginia Tech Blacksburg Virginia 24061-0251	7b. ADDRESS *(City, State, and ZIP Code)* Dahlgren, Virginia 22448-5000
8a. NAME OF FUNDING/SPONSORING ORGANIZATION JLC/CSM 8b. OFFICE SYMBOL *(If applicable)*	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS *(City, State, and ZIP Code)* Dr. Raghu Singh, Chair Space and Naval Warfare Systems Command Mail Code 31F1 Washington, DC 20363-5100	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. | PROJECT NO. | TASK NO. | WORK UNIT ACCESSION NO.

11. TITLE *(Include Security Classification)*
Software Quality Measurement: Validation of a Foundational Approach Final Report, Year One

12. PERSONAL AUTHOR(S)
James D. Arthur, Richard E. Nance, Gary N. Bundy, Edward V. Dorsey, Joel Henry

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT *(Year, Month, Day)*	15. PAGE COUNT
Final - Year One	FROM 1/1/90 TO 12/31/90	1991 May 1	143

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
FIELD	GROUP	SUB-GROUP	Simulation quality assessment and prediction; software engineering objectives, principles and attributes; software quality indicators, process indicators, code indicators, document quality indicators

19. ABSTRACT *(Continue on reverse if necessary and identify by block number)*

This report discusses the first year findings of a proposed three year investigation effort that focuses on the assessment and prediction of software quality. The research exploits fundamental linkages among software engineering Objectives, Principles and Attributes (the OPA framework). Process, code and document quality indicators are presented relative to the OPA framework, with elaboration on their individual roles in assessing and predicting software quality. The synthesis of an Ada code analyzer is discussed as well as proposed complementary tools comprising an automated data collection and report generation system.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE *(Include Area Code)* | 22c. OFFICE SYMBOL

DD Form 1473, JUN 86 *Previous editions are obsolete.* SECURITY CLASSIFICATION OF THIS PAGE