PROVIDING A FRAMEWORK FOR EFFECTIVE SOFTWARE QUALITY ASSESSMENT

A FIRST STEP IN AUTOMATING ASSESSMENTS

Robert A. Martin and Lawrence H. Shafer

Presented At

The First Annual Software Engineering & Economics Conference

"Software Systems Modernization"

2-3 April 1996
MITRE Hayes Auditorium
McLean, Virginia

MITRE

The MITRE Corporation
202 Burlington Road
Bedford, MA 01730-1420

1. INTRODUCTION -- ASSESSMENT IN PERSPECTIVE

Software quality assessment is a field that is coming into greater focus as the global drive for systemic quality assurance continues to gather momentum. Many forces are at play, from the pressures of consolidations, mergers, and down-sizing, to the emergence of new technologies which are sparking renewed looks at re-engineering in business and government, particularly the advances in user interface capabilities endemic to the PC and distributed processing economic explosions. These developments and forces have led to our creation of a quality assessment methodology which harnesses the results of program analysis techniques to generate a timely analysis that can be productively used to focus management attention on the fundamental quality issues within their software systems.

This paper will review the historical background of software quality assessment and show that there is substantial consensus on the framework needed to measure the quality of software artifacts, including documentation and many levels of program code. We will show that this framework has been appropriately justified from methodological and core semantic grounds to give assurances that what it is purported to quantify is strongly related to basic quality criteria. Finally, we will describe our design and implementation of a system for applying this framework in practice to a small but rapidly growing sample of real systems in a variety of application areas. We contend that even without fundamental advances in any of the source technologies, an assessment system that produces reliable and useful results across various stages of the software life-cycle is now practical.

Over the last couple of years, a group charged with developing tools to promote software reuse and re-engineering at MITRE has explored the definition and prototyping of a Software Quality Assessment Exercise (SQAE) technique. The goal is to produce an assessment system that satisfies the objective of producing the above reliable and useful results across all stages of the life-cycle of software systems. As part of this effort a set of tools and evaluation methods to provide *repeatable* and *consistent* measures of the quality of software products has been designed and partially developed. This approach has been used successfully for more than 75 system evaluations as part of source selection or project management studies. It has been adapted to include artifact evaluation as part of process evaluation, and it is being improved to allow distributed assessment and to address additional needs of specialized applications, such as systems with transaction processing components.

2. STRUCTURES FOR QUALITY ASSESSMENT

The original analysis of a structure for software quality assessment was done by B. Boehm and associates at TRW (1 - Characteristics of Software Quality) and incorporated by McCall and others in the Rome Air Development Center (RADC) report (2 - Factors in Software Quality) in 1978. The basic structure involves quality attributes related to quality factors, which are decomposed into particular quality criteria and lead to quality measures. This framework has been studied and discussed in detail, most recently in complete form by Kitchenham and various co-authors (3 - The Architecture of Software Quality, 4 - Towards a Constructive Quality Model, 5 - Quality Factors, Criteria, and Metrics) in connection with the ESPRIT work that was input to the standards work producing the IEEE Standard (6 - A Standard for Software Quality Metrics Methodology). Work establishing the rela-

tionship of the quality factors suggested by these works to real-world system concepts, such as reliability, and to software-specific concepts, such as maintainability, has been done incrementally over the years. This work is in various stages of low level statistical analysis for experimental or trial projects, and seems to be converging in recent work such as Schneidewind (7 - Methodology for Validating Software Metrics), Grady (8 - Practical Results from Measuring Software Quality), and Stark et. al. (9 - Using Metrics in Management Decision Making).

The model originally proposed in 1973 by Boehm and his coworkers is shown in Figure 1. Factors such as portability, reliability, understandability and modifiability are basic static properties. Each factor is composed of several criteria which contribute to it in a structured manner. Testability and efficiency depend on the behavior of specific interpretations and are thought of as dynamic properties.

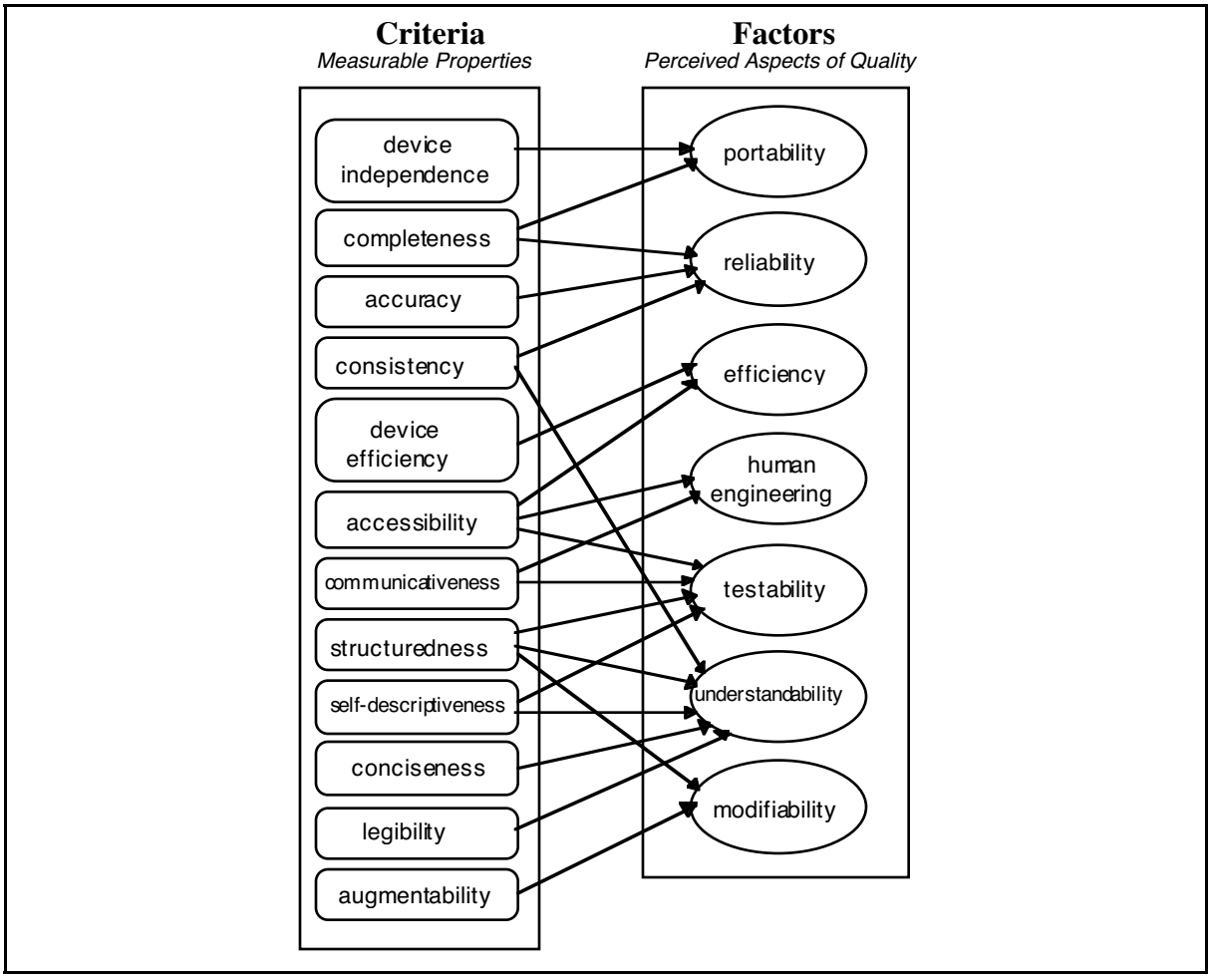


Figure 1: Boehm et. al. Original Quality Factors and Criteria

The later RADC work by McCall et. al. included additional criteria such as generality, modularity, data commonality, communications commonality, access control and simplicity.

It also decomposed device independence into system and machine independence. The diagram corresponding to this additional work is shown in Figure 2.

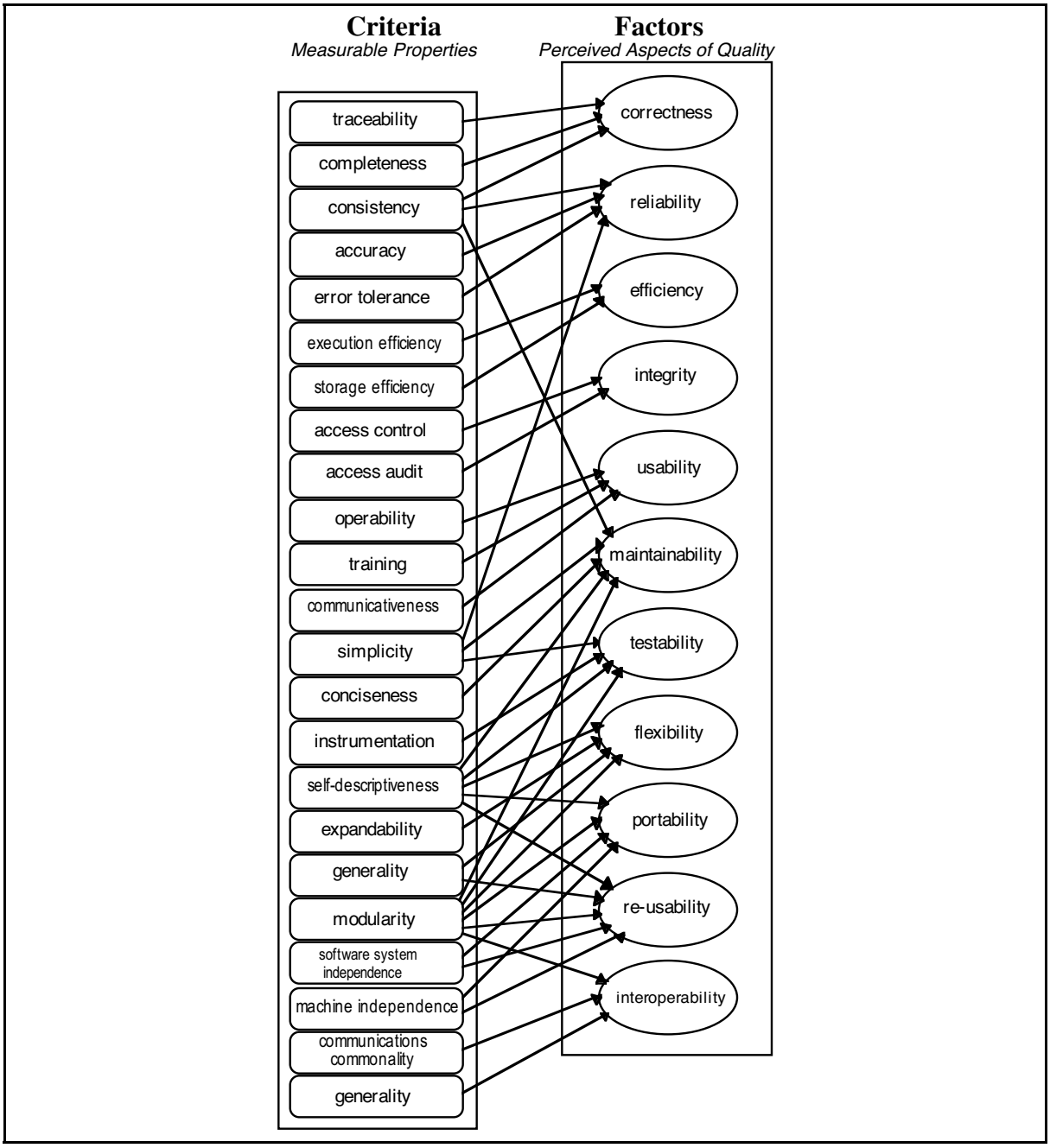
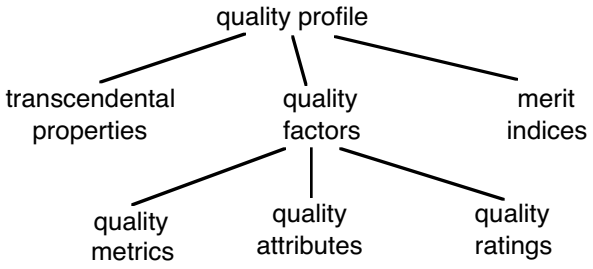


Figure 2: McCall et. al. Modified Quality Model

Further work at RADC since the early eighties has produced an extended evaluation framework that is now partially productized as the Quality Evaluation System (QUES) (10 - Software Quality Framework, for the RADC Technology Transfer Consortium). This framework includes 13 factors and decomposes them into 29 underlying criteria using an interconnected matrix of measurable qualities. Evaluations are conducted using more than 500 questions, and analysis includes an extensive suite of statistical programs. Many of the criteria involve analysis related to dynamic program behavior.

The RADC model was expanded and modified by Kitchenham and her various coauthors as part of a coordinated effort (called the European Strategic Programme for Research in Information Technology or ESPRIT consortium) of European researchers on quality, as reported in a series of papers in 1987. Their analysis was superficially somewhat more complex but attempted to clarify the relationship between quality factors and criteria in terms of values that can be measured in ways that more clearly distinguished between the kind of quality measure and overall goals. In particular, they suggested an important distinction between objective and subjective measures. They expanded "quality factor" to include three kinds of properties, contributing to an overall quality profile. The three categories they suggested were "transcendental properties" (not measurable), quality factors (to include objective metrics and binary attributes), and subjective merit indices.

Their overall model can be described by the simple diagram:



The functions that determine quality factors and merit indices can be formulated either algorithmically or statistically. The composition of these functions can be derived from a theoretical understanding of the relationship between the directly measurable parameters and the derived properties of the system. The best examples of quality factors in software are the degree of "structuredness" of some code. Metrics, such as McCabe Cyclomatic Complexity, are the objective parameters of the control structure of a program, for example. On the other hand, a quality attribute for a comparable program property would be a Boolean value that indicated whether or not all structures in a program are consistent with some predefined standards.

Quality factors and merit indices described the engineering concerns with defining the characteristics of an assessment methodology. A property such as quality must surely have to do with both the structure of a system and perceived performance in achievement of its objectives. A subjective quality factor is one that has no adequate theoretical support. It may still reflect well-formulated, explicitly stated value judgments, or even arbitrary functions over objective parameters that nevertheless express some essentially qualitative aspect.

For example, readability may be purely a judgment issue, or may be modeled using some combination of apparently arbitrary objective measures such as the length of identifiers, the length of statements, and the total number of words, statements, and paragraphs in a document. Axiomatic formulations of such subjectively determined functions are actually of the greatest value in just such cases. Merit indices are similarly functions of quality ratings that can be formulated algorithmically or statistically. Since ratings are inherently subjective, all merit indices are of this sort.

Attempted standardization work over the intervening years resulted in the standard ISO-9126 in 1991 (11 - Software Product Evaluation Standard). This model was fairly closely patterned after the original Boehm structure, with the six primary quality attributes: functionality, reliability, usability, efficiency, maintainability, and portability. Each of these was broken down into secondary quality attributes: maintainability, for example, into analyzability, stability, testability, and modifiability. While a good descriptive approach to defining quality, this approach was not designed with the objective of facilitating measurement, and in effect ignored attempts, such as Kitchenham's, to build profiles directly from metrics and ratings. Dromey (12 - A Model for Software Product Quality) reemphasized this distinction and reported significant repeatable results with his PASS (Program Analysis and Style System) methodology, based on a model consisting of three bottom-up levels:

- a set of quality components
- a set of quality-carrying properties
- a set of high-level attributes

Dromey suggested quality-carrying properties be categorized into four areas:

- correctness properties reflecting external consistency/validity
- structural properties reflecting intra-module design
- modularity properties reflecting inter-module design
- descriptive properties reflecting internal consistency/validity

The first category has to do with overall functional specifications, and would probably include traditional terms such as basic functionality and interoperability. Testability and analyzability would probably involve the first three categories in various ways. Overall code quality could be involved with aspects of all the last three categories in specific ways. The last category could include the quality of various forms of documentation concerned with mapping overall specifications into detailed design.

Dromey proposed a carefully constructed matrix relating these properties to the six ISO-9126 first-order attributes, plus an additional attribute he calls reusability. This matrix, shown in Figure 3, presents a mapping from measurable and rankable product characteristics to the high-level attributes. This mapping is used in actual software product evaluations by the PASS assessment system. He suggests that repeatability is an important characteristic of this kind of system, suggesting

> ... a primary requirement for the proposed model of software product quality was that it should possess a classification procedure [such that]... two people familiar with the model should arrive at the same classification...

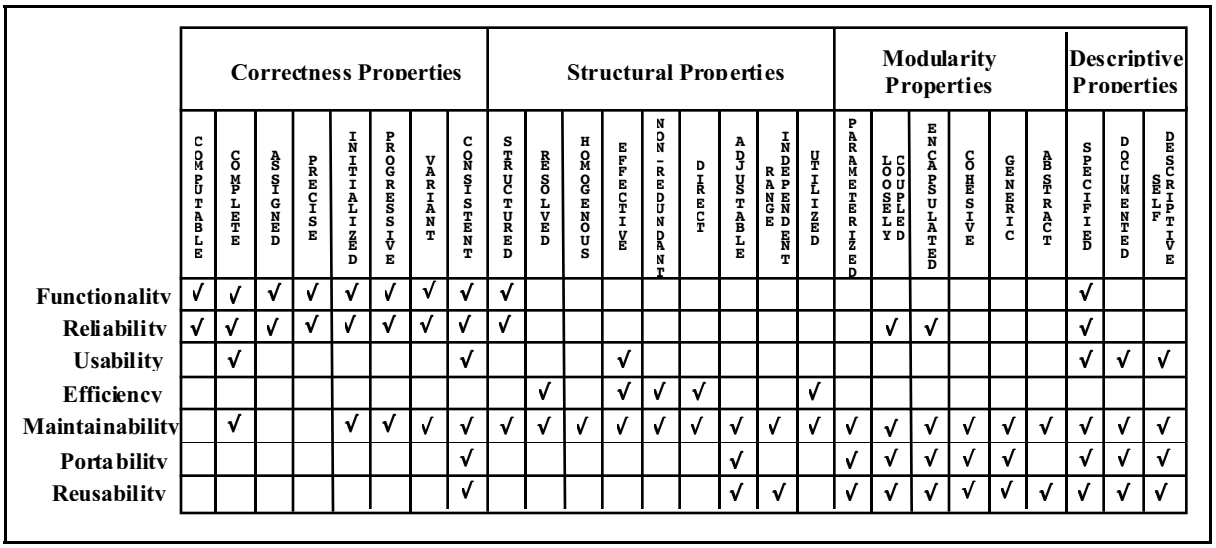


Figure 3: Dromey Matrix of Quality Factors

Further work of the ESPRIT project included work on measurement methodology called SCOPE, reported on most recently by Robert Dick at the University of Strathclyde (13 - Subjective Software Measurement). He discusses use of objective techniques for combining subjective and objective measures in a controlled way. One of the most illustrative examples of the need to do this is the ISO-9126 breakdown of "maintainability". Each of the four secondary quality attributes of the primary term has both kinds of metrics:

Secondary Quality Attributes of Maintainability	Examples of Objective Measures	Examples of Subjective Measures
Conciseness	Number of lexemes	Use of idiomatic terms
Legibility	Number of comments	Use of white space
Modularity	Number of statements	Cohesion of blocks
Structuredness	Depth of nesting	Simplicity of flow

While tools exist and techniques abound (more about some of these trends later), an important practical issue is how to compose these metrics meaningfully into useful evaluation techniques to apply to real maintenance and other software life-cycle problems. He writes of a similar kind of measurement problem, wine quality,

> [Even if] The metrics could be objectively determined by measuring equipment and the results presented neatly by tables and graphs, ... who would accept this as a rating of quality? In practice, an expert, or more likely a panel of experts, sample the wine and give a learned but nonetheless subjective assessment of the wine quality.

Similarly, Dick suggests, distinguishing good and bad features by combining expert intuitions is a way software quality judgments can be validated. To handle this, Dick writes,

> The SCOPE project developed the notion of an evaluation checklist. Checklists are encapsulations of the experience of a number of experts with knowledge of assessment techniques and consist of questions which guide the human expert - known as the assessor - through the evaluation.

Dick describes three aspects of this process: evaluation lists, scoring via ratings, and a well-engineered user interface to present a sequence of criteria to the assessor and record the results. An additional issue in doing the statistics is using the results, which is another kind of problem, much discussed in the literature. For the purposes of the current work, as well as for SCOPE, this was not regarded as crucial to designing and demonstrating a useful framework for getting meaningful data. SCOPE concentrated on a framework for organizing both a windowed version of a question-answer system and source program browser. The browser was for Pascal, and provided such special capabilities as a menu for searching on various declaration and statement types. This was interesting, but even when done was probably not state-of-the-art in static program analysis approaches, even menu-driven window-based ones.

The most transferable part of the SCOPE work is its way of organizing the queries, rankings and evaluation results and the use of on-line documentation and backup information about the questions, the definition of the ranking choices, and the purpose of the ranking approach. A typical example of a ranking is that for the checklist item:

> Are comments generally useful?

for which the rankings are:

0 - No, never	3 - More than half the time
1 - Rarely	4 - Mostly
2 - Less than half the time	5 - Always

The low values represent undesirable rankings and high values desirable ones. Each checklist item is similarly quantified, so that a score for a factor such as maintainability can be given by adding up the rankings related to it. Dick describes some of the useful considerations involved with this approach:

> Questions need not be answered in strict numerical sequence; in fact, they can be answered in any order at all. To allow this, … a browsing facility … consists of two command buttons 'PREVIOUS' and 'NEXT'. When the assessor clicks on either of these, the data for the current question is replaced by that of the previous or next one. This feature means, for example, that the assessor can leave difficult questions to the last, or can inspect the questionnaire before answering any question.

The result of the questionnaire can be saved for later analysis or analyzed immediately to generate any user-programmed report as provided. The evaluation can be checked by other assessors or combined with independent evaluations, as appropriate. Dick concludes with,

> An extension of the automated subjective technique may choose to apply assessment-aiding tools to other software process outputs. Any deliverable which has formal syntax and semantics is a [good] candidate … Such tools might lead to better quality specifications and would … allow problems to be caught early in the software life-cycle.

Removing subjective effects of human judgment can be dealt with by rules and conventions. Typically, using several evaluators with some kind of averaging employed to determine ratings is the favored methodology. This can be done with various voting schemes, and introduces very interesting areas of statistical analysis. Simple rules would include using extra evaluators and removing scores with high variance from the average. This is actually done in some standard DOD exercises, e.g., the Air Force Operational Test and Evaluation Center (AFOTEC) maintainability evaluations (14 - Software Maintainability Evaluation Guide). The AFOTEC quality evaluation system uses straightforward static analysis and human judgment of conformance to around 150 quality criteria based on attributes related to maintainability and portability. This scheme requires agreement by six evaluators using six levels of conformance from complete conformance to non-conformance. An average score for all items, using the rule of removing the highest and lowest for each attribute, is accumulated for three areas, documentation, code and implementation. This kind of scheme is the subject of some statistical and operations research analysis, and ideally could be quantified or parameterized experimentally to get the best overall validity or long term fit. This could be one approach in a SCOPE-like system, which could easily keep data for several evaluators on successive evaluations.

3. METRICS BACKGROUND

What objective measures to include is itself a judgment decision based largely on the same kind of expert consensus. The history of assessment and standards suggests there are ways to make useful decisions in this regard. The range of possibilities runs the gamut from traditional program analysis to very modern static evaluation techniques that are the basis of current research in diverse areas such as parallel programming and structural testing. The SCOPE project's use of browsing as a prototype activity is just the barest indication of the possibilities of the use of the combination of evaluation and assessment framework ESPRIT could eventually approach. The Rome study and subsequent reworking of it suggest standards work related traditionally to software development is one very useful source of quality considerations. These are more or less quantitative, but many of the issues with traditional methods persist.

The most traditional program analysis techniques are McCabe Cyclomatic Complexity (15 - A Complexity Measure), Halstead Software Science (16 - Elements of Software Science) volumetric techniques, and a wide range of tailored structural and graph-based approaches developed over the last two decades. McCabe's pioneering metrics involved the "cyclomatic" complexity based on the Euler winding number of the control structure graph of a program, and "essential" complexity based on a reduced graph of subprogram entrances and exits. To this day, his work remains the standard for analyzing program complexity and the values included in most assessment systems are a baseline for identifying the most difficult areas of software systems. What McCabe metrics miss is expression complexity and some detail about the complexity of nested control structures. Halstead's metrics address some of these concerns, including computing the average number of occurrences of operands and operator symbols in each subprogram. For dealing with other issues, Halstead introduces notions of the average lengths of identifiers and statements and uses heuristics based on the idea of "psychological difficulty" in dealing with many different items. Control structures are dealt with by suggesting comparisons and selection operations are also related

to the number of "distinctions" to be made in understanding the program. The Halstead program volume is a weighted average of these statistics and provides a measure of the effort needed to make use of or modify the program.

Less well known is a variety of refinements and extensions of these approaches developed using graphical manipulation and static analysis techniques applied to basic program structures. The branch most related to McCabe's work is the use of trees and symbolic expressions to represent cyclomatic behavior of programs. The work of Tamine (17 - Using Tree-like Structures to Simplify Measures of Complexity) and McCabe (18 - Structured Testing Using Cyclomatic Complexity) involved applying these notions successively to issues of complexity, maintenance, and testing. Work on this area continues to yield models of local graphical behavior, parameterized to combine structural factors similar to Halstead volumes, using polynomial combinations of directly measurable source properties (19 - Metrics for Assessing Software System Maintainability). While quite interesting research, these suggestions still have not been accepted by enough experts to be useful in improving or supplanting the earlier metrics for purposes of quality assessment. What they probably show is that for special purposes, certain more local properties than what is measured by McCabe or Halstead metrics are useful, but do not help in understanding overall program characteristics, at least of large systems.

What is more common is the use of similar program-related modeling ideas to deal with specific areas of software management and process description. This is so common these days that it is well to note that although methods are often quite similar (as noted above for program metrics), the intent is to develop metrics related to particular software system application areas and even particular life-cycle stages of those applications.

A related kind of extension is to areas incorporating the results of more dynamic analysis. This is more like the direction taken by continued work at RADC on the QUES system, which is very similar to all the above efforts in areas based on static analysis, but also incorporates results of testing and performance analysis in a more comprehensive evaluation. This allows the development of more extensive metrics, particularly in the case of latter stages of the software development cycle. There is no question these are valid and important additions to software assessment when sufficient long term data or dynamic analysis facilities are available. Incorporating this kind of information into a data base for predicting software behavior is an active research area. The QUES system includes this kind of data in the form of quality factors such as verifiability, reliability, integrity, efficiency, and usability.

4. DESIGNING AN EFFECTIVE QUALITY MEASURE

Given the variety of on-going efforts attempting to scope the assessment of software we were faced with a choice of directions to proceed with our own work. Following the ESPRIT effort's emphasis on defining a repeatable and well-documented assessment method that would provide useful information to management, we decided to start our work with a working definition for quality as the *minimalization of the life-cycle risks of a system*. This approach is in concert with the emergence of an industry-wide emphasis on the life-cycle characteristics of systems and is consistent with managing a system throughout its lifetime so that trade-offs between its design, development, and maintenance issues can be made from an

integrated perspective. Software systems are expected to be modified over time, so it is of utmost concern that they support ease of modification and evolution. In this context we reward systems which minimize risk. Such risk areas include: the risk of introducing errors during the development and maintenance phases of the system, the risks associated with rehosting the system from one machine to another or from one version of an operating system to another, the risks associated with making enhancements to the system, the risk inherent in staff changes in the development or maintenance organization, and the risks to project schedules associated with testing and deployment pressures.

With this as a working definition, we first explored the types of issues that have been considered in previous efforts to obtain a measure of system quality. In doing this we drew mostly upon the pioneering RADC work and the SCOPE experiments with table-driven scoring and analysis. We also sought to capitalize on the existence of industry and Government open system standards and we wanted to include static analysis assessment areas that would have been impractical to evaluate before the arrival of today's powerful computers and modern software reverse-engineering analysis methods. To this end, we explored some non-traditional approaches to query-based analysis that have not received as much attention as they probably should.

Of many current activities in static program analysis, some of the work that is most closely related to software assessment needs is efficient query research, applied to program data bases. A compelling indication of the value of this approach came in a short paper (20 - Rapid Software Assessment), where B. D. Rosenblum describes how doing simple non-contextual searches for features such as:

- use of non-standard OS services
- use of special file formats or embedded path names
- unusual usage of externals and globals

will produce results that can be used to determine major points of interest in programs. This information then can help to organize additional hypotheses and queries helpful in gaining insight into the program's strengths and weaknesses. Very high on his agenda is efficiency of queries and timely support for human intuition in formulating and evaluating hypotheses.

Using queries to organize the acquisition of information about programs is an active research area. This is related to code-level re-engineering and common theories about machine assistance for program understanding in maintenance and testing. In the area of assessment, querying a software system database can provide:

- information about characteristics that indicate special areas of concern (as in Rosenblum's work) and can be easily, or at least straightforwardly, related to quality-carrying properties

- support for correlating information from one medium (e.g., design documentation) to another (actual code) and making use of contextual data to establish viewpoints for interpreting models and sequencing operations

- information about relationships of parts on an incremental basis, so that searching for queriable instances or meaningful samples can follow logical or at least understandable deduction paths

The real problem being addressed is the weakness of traditional metrics as possibly objective measures. Many problems with cyclomatic or volumetric metrics have been identified over the years (21 - Evaluating Software Complexity Measures), involving essentially their inability to distinguish significant differences in programs with the same complexity values. Concepts such as cohesion and coupling with inherently specialized definitions have been the main compensatory formulations; however they become objective only under controls similar to the above list of reasons for using query mechanisms. The assessment problem becomes somewhat of a hostage to progress in the basic objectives of query management.

Areas of research that are important to making progress in assessment are efficiencies related to:

- formulating queries clearly with understandable relationships to quality-carrying properties
- keeping contextual information in convenient and accessible ways to facilitate meaningful use in constraining searches
- managing query processing so that partial results are useful and can be used meaningfully to assist in understanding results

Encouraging results appear occasionally which suggest that SCOPE-style browsing might be a model for improving assessment. The notions of context maintenance and reporting of partial results are consistent with the menu driven approach to analysis. Complication of the internal data base is a natural concern. Some results have been reported of re-engineering approaches using query-oriented structured code analysis (22 - Reverse Engineering to the Architectural Level) to evaluate conformance of code to semi-formal specification information. This is relevant to evaluating attributes such as simplicity/ understandability, independence/portability as well as to the intended goal of reusability (which could be thought of as either a metric or attribute in the context of such an analysis).

Techniques for improving search efficiency directly follow in the tradition of fast pattern matching that dates at least from the Boyer-Moore (23 - A Fast String Matching Algorithm). Of equal significance is work on organizing contextual information that has roots in very early program trace research such as the work of Habermann and Campbell (24 - The Specification of Process Synchronization By Path Expressions) in the mid-seventies. This comes together in such work as Parnas' formulations of trace semantics in tables and rewrite sequences as early as the late seventies (25 - Using Traces to Write Abstract Specifications for Software Modules) and as late as the end of 1994 (26 - Precise Documentation of Well-structured Programs). Both these trends are contributions to work such as that of Paul and Prakash on a tailored language and pattern matching system, which they have experimented with in competition with traditional analysis approaches to produce favorable results. The working principle of their system is code pattern automata written using sub-tree traversal idioms that can operate directly on the results of parsing the source code. Currently they

save the parsed source in intermediate form, but this does not appear to be necessary to the success of their system. What is needed is a way to save the state of a query in progress, in order to allow some programmable way to control exploration of likely interesting matches. This is mentioned as a possible future direction by Paul and Prakesh, who suggest a concept of "query-pipelining" (27 - A Framework for Source Code Search using Program Patterns).

An even more direct relationship to software assessment needs is suggested by Howden and Wieland in describing possible applications of an assertion checking system (28 - Quick Defect Analysis (QDA)). Their work involves an extension of assertion checking to analyze comments in source code. They describe a method and tool for incrementally verifying assertions about use of global data using "path splitting" to examine the effects of usage. The objective is to determine where object invariants are violated. Their analysis of operational flight code discovered 480 potential defects or problems, but just as interesting was one of their final suggestions:

> … [This] analysis provides a record of a code reading process … Analysts are required to document working hypotheses that occur during code reading, and then … held accountable for the failure to find some fault … Suppos[ing] … the analyst's goal is to determine if a piece of software has some general property … It may be possible to conclude that the property is correct provided some more specific, program dependent set of properties holds. These problem dependent lists are like QDA hypotheses, whose verification will require the creation of appropriate facts, which may lead to new lists of hypotheses.

They specifically suggest this approach is similar to "browsing" and that their hypothesis language could be extended to allow expressions with quantization over paths. The feasibility of a SCOPE-like user interface to control these aspects of specification-style query satisfaction seems apparent. What this would add to a system for software assessment is the ability to make adequacy of subjective rankings a realistically measurable objective. The goal is simply that subjective metrics be quantifiably validatable, with repeatability or incremental provability a kind of guarantee of objectivity. The architecture of the assessment tool should enforce this property, to the extent possible in the current assessment world.

4.1 Designing The Framework For Measuring Quality

In designing our methodology, we have focused our attention on the development of a language- and system-independent evaluation framework, scoped to address the specific issues of life-cycle stability. Furthermore, we have modernized and generalized many of the criteria and concepts presented in the earlier works to fully capitalize on the existence of open standards and powerful software development and analysis tools. The resulting Software Quality Assessment Exercise (SQAE) methodology relies on two fundamental practices to produce meaningful and repeatable results: a rigid scoring criteria imposed to constrain the subjective nature of the non-metric evaluation criteria and the integrated use of analysts' observations and annotations rather than a simple numerical score in the creation of the total risk profile for the evaluated system.

The SQAE methodology is built around the concept of establishing a hierarchy

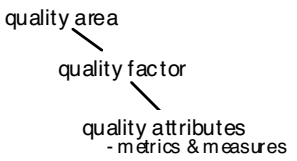


Figure 4: Hierarchical Area, Factor, Attribute Model

for abstraction in which high level concepts relating to life-cycle risk are composed of more tangible and measurable factors. The factors establish a context within which metrics, documentation, and coding standards may be used to generate a profile of strengths and weakness imbedded in the design and implementation of a given system. These interrelationships are shown in Figure 4.

4.2 Quality Areas And Quality Factors

The result of our integration of previous work with our own ideas is discussed in the following paragraphs. Here we introduce seven "quality factors" that serve as a measurable foundation for the definition of the four quality areas of maintainability, evolvability, portability, and descriptiveness. The factors: consistency; independence; modularity; documentation; self-descriptiveness; anomaly control; and design simplicity are less abstract than the quality areas mentioned above and will provide a better framework in which to measure the quality of a system. The relationships between these seven quality factors and the four quality areas are shown in Figure 5 and echo the types of relationships between quality areas and quality factors that have appeared in the software quality frameworks since the first RADC work.

We use the term *quality area* to define the high level concepts of life-cycle risks. Although most will agree that these features are good for a system to possess, few people will agree on what the actual terms mean and fewer still can agree on how to demonstrate the existence of a given feature. To address the hidden subjectivity of the terminology, we express our quality areas as the weighted sum of the various factors that comprise the aspects of the concept we seek to measure. Since we decided to limit our activities to static analysis of actual source code and related artifacts, we chose maintainability, evolvability, portability, and descriptiveness as our basic quality areas. These are discussed below, and are rigidly defined by the attributes and ratings used in the assessment process. The use of percentages of quality factors to derive overall scores for particular quality areas is one of the unique outcomes of our work. These percentages were initially empirically derived to reflect evaluators' perceptions of software quality for sample programs, and have been used satisfactorily for a large number of programs since being established.

For each quality factor shown in Figure 5 we have defined a mapping of that quality factor's contribution to one or more quality areas. Each quality factor is itself further defined by a set of attributes. The quality attributes are deliberately worded at the conceptual level

and avoid use of subjective terms and assumptions regarding the language(s), architecture, or target platform for the system being assessed. The factor attributes are distinct, measurable questions or metrics that address the various ways the concept of the factor may be implemented in the code.

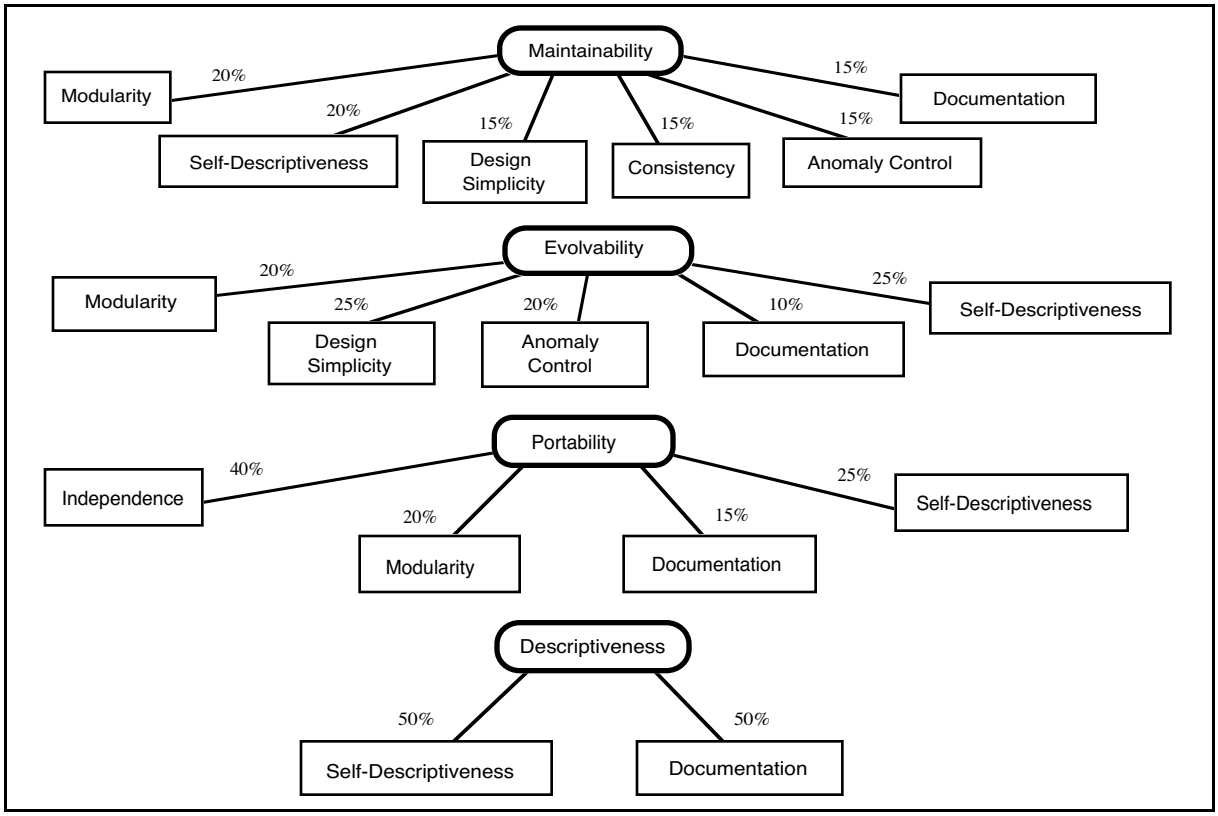


Figure 5: Quality Areas to Quality Factors Mapping

For example, Figure 5 shows that the factor Self-Descriptiveness contributes to all four quality areas. As defined below, it is composed of attributes that measure the information content of the prologues and comments, use of meaningful naming conventions, white space management, etc. By enumerating a list of distinct and measurable questions or metrics that can be evaluated we can construct a structured, repeatable method for measuring the quality of an application system and its supporting documentation. Summarized representative questions showing the general intent and focus of the seven core quality factors of our assessment framework are as follows.

- Consistency: Have the project products (code and documentation) been built with a uniform style to a documented standard?
- Independence: Have ties to specific systems, extensions, etc. been minimized to facilitate eventual migration, evolution, and/or enhancement of the project?

- Modularity: Has the code been structured into manageable segments which minimize gross coupling and simplify understanding?
- Documentation: Is the hard copy documentation adequate to support maintenance, porting, enhancement and re-engineering of the project?
- Self-Descriptiveness: Does the embedded documentation, naming conventions, etc. provide sufficient and succinct insight into the functioning of the code itself?
- Anomaly Control: Have provisions for comprehensive error handling and exception processing been detailed and applied?
- Design Simplicity: Does the code lend itself to legibility and traceability where dynamic behavior can be readily predicted from static analysis?

Before we can structure a fully repeatable measurement instrument that is independent of the evaluator and applicable across the full gamut of languages, environments, and architectures, we need to consider the features that will provide these types of capabilities and we need to think through the way we will communicate our findings to management.

Each of the quality factors is decomposed into numerous quality attributes which explore the different facets of the quality factor in question. Finally, for each attribute of a factor we define a scope for evaluation, an attribute weight, a rigid scoring criteria with evaluation guidelines, and a supplemental evaluation context.

The scope is defined as that portion of the delivered product that will be examined in the course of a particular assessment. For many attributes (especially those that can be evaluated in a fully automated way, such as complexity metrics) the scope is the entire body of code while others, (the more intellectually demanding tasks) are restricted to a "representative sample."

With the above working definitions of the factors underlying the maintainability, evolvability, portability, and descriptiveness of the system in concrete terms we can move on to exploring the question of how to actually go about measuring the quality of a system.

4.2.1 The Quality Of Maintainability

We evaluate the maintainability of a software system by studying the implemented code and documentation and evaluating how each reflects on the overall architecture and design. In assessing maintainability, our concern with understanding the effort needed to locate and fix software failures and make minor changes leads to queries concerning overall code readability and the quality of information available to determine inter-module interactions. We evaluate the documentation at the system and architectural level, and assess any information about overall and unit level data flow. We evaluate uses of formatting to encourage understanding of local control flow and the content and utility of documentation and comments in the code itself. Consistency is revealed by standard approaches in naming, organizing code into modules, and providing prologue information and commentary that provide reliable guidance to how things are done. Modularity is measured by the structure of the software using common and simple metrics. Simplicity of error detection design and adequate han-

dling of anomalies are evaluated in terms of their contribution to understanding the effects of a portion of code or subsystem interaction.

4.2.2 The Quality Of Evolvability

We measure evolvability by looking at the system in terms of the operations required to perform effective modification to adapt to changing environments. We examine documents and code with the main concern being the relative ease of changing the system to accommodate changes in requirements and need for enhanced capabilities. Simplicity of design is reflected in clarity of design documentation and accessibility of information of documentation and code. Low complexity of code, concern with modularity of decomposition and implementation, and effective control of errors in the software are the important implementation considerations. Appropriateness and independence of languages and tools, their amount of standardization, are also factors in enhanceability. Overall informative and accurate description in documentation and code are the more general factors in making judgments of evolvability.

4.2.3 The Quality Of Portability

Portability is a function of the hardware platform or any operating system or other dependencies on the target environment. Both the developed software and the commercial software products within the system, the language they are written in, and the platforms on which they are available need to be evaluated for independence. We also assess use of I/O protocols, graphics standards, the existence of any direct hardware calls, and the machine/system software specificity of any environmental parameters or control scripts of commercial software. Independence is the major factor in this quality area. Modularity is also important and can be measured by a combination of traditional metrics and judgments on cohesion and coupling. The extent to which the system is well documented and its use of platform unique features is described and explained is also judged using all available information.

4.2.4 The Quality Of Descriptiveness

Descriptiveness is evaluated by studying external printed material about the system and the source code resident documentation. Both high level functional descriptions and characterizations of the system, as well as low level design details, affect the understandability of the system's functionality as well as its consistency. Modules are evaluated for standard formatted prologue sections and formatting. Naming conventions are judged for their contribution to the legibility and understandability of the code. Comments are evaluated on the basis of their contribution in highlighting special features and clarifying aspects of functionality, error handling and esoteric processing.

4.3 Defining A System Quality Measure

Our fundamental approach depends on widely held and mutually agreed upon notions of basic software engineering. In some ways it now seems (with some advantage of historical hindsight) to be a kind of epiphany of common sense. How to relate this to a valid and reli-

able measurement approach is the greater problem. Again basic computer science comes to the rescue. What is needed is a model of what might be called coordinated, inferential processes. Understanding how to make this a coherent and repeatable process is the issue. Many of the past and existing efforts to define quality do not make any claims to producing a full set of metrics. This is true of the original RADC work, and of most of the direct derivatives, including the latest version in the QUES system. The function point analyses used for transaction processing systems (29 - Software Function, ... A Software Science Validation) is the effort that is closest to a goal of providing insight into life-cycle issues using a purely static analysis. Other evaluation systems, including the AFOTEC maintainability assessment, provide a large amount of intentionally focused assessment data, that is not easily related to facilitating management decision making based on such considerations as relative risk. What is unique about the Software Quality Assessment Exercise approach is its intent to provide this kind of information about current risk, based on current artifacts (both documents and code) available at any stage of the software cycle in a very time efficient manner. Dealing with this kind of problem is crucial to effectively managing software.

When assessing a multi-language system, we are applying a coordinated set of measurements and evaluations of the artifacts that make up the system. We look at each of seven quality criteria factors and assess them using a variety of examination techniques that make use of focused tools, common sense metrics, and narrowly based human judgments. Then we relate them to quality areas that have a commonly accepted impact on concrete software goals. Thus all our analysis and judgments are related to the traditional problems and widely accepted perspectives of software development and assessment. Then looking at the interpretations of the results as the continuation of an inferential data base problem brings current information management methods to rolling up individual metrics into displays and reports that have easily interpreted and meaningful statistical results. These results are in terms of standard reports and graphics that present such data as comparisons of the system being assessed to all previously assessed systems.

We base this investigation on the seven quality factors, and measure or judge the quality according to 76 specific attributes derived from our lists of basic concerns and from our synthesis of past research. The focus of the quality issues covered by each of the seven quality factors is described in the following paragraphs.

4.3.1 Measuring Consistency

Consistency is a direct reflection of the policies and procedures for all aspects of the development process. Consistent software is built when a development standards document exists and development conforms with this document. Any potential issues on documentation, I/O protocols, data definition and nomenclature are addressed here. The Consistency factor is mapped to the maintainability quality area. The full list of Consistency attribute questions is contained in Table 1 in the Appendix.

4.3.2 Measuring Independence

Measurement of independence requires attention to two broad areas: software system independence and machine independence. Here the issue is to avoid tying the system to a host environment that would make it difficult or impossible to migrate, evolve, or enhance the

system. The Independence factor is mapped to the portability quality area. The full list of Independence attribute questions is contained in Table 2 in the Appendix.

4.3.3 Measuring Modularity

Modularity consists of several facets, each of which supports the concepts of organized separation of functions and minimizes unnoticed couplings between portions of the system. The Modularity factor is mapped to the maintainability, evolvability, and portability quality areas. The full list of Modularity attribute questions is contained in Table 3 in the Appendix.

4.3.4 Measuring Documentation

Documentation refers to the supplemental material about the system that is external to the code. The documentation must be adequate to support the maintenance, porting, and enhancement activities that will occur throughout the system's life. High-level functional descriptions, characterizations of the system, and low-level design details are needed to support the maintainers' need to understand the system's functionality as well as identify where to make changes and corrections. Note that this differs from Self Descriptiveness, which is a quality factor for the documentation embedded within the code itself. The Documentation factor is mapped to the maintainability, evolvability, portability, and descriptiveness quality areas. The full list of Documentation attribute questions is contained in Table 4 in the Appendix.

4.3.5 Measuring Self-Descriptiveness

Modules should have standard formatted prologue sections. These sections should contain module name, version number, author, date, purpose, inputs, outputs, function, assumptions, limitations and restrictions, accuracy requirements, error recovery procedures, commercial software dependencies, references, and side effects. White space and naming conventions should be used to help the legibility and understandability of the code. The judicious use of comments to highlight special features and to clarify the code's functionality is also desired. The Self-Descriptiveness factor is mapped to the maintainability, evolvability, portability, and descriptiveness quality areas. The full list of Self-Descriptiveness attribute questions is contained in Table 5 in the Appendix.

4.3.6 Measuring Anomaly Control

Comprehensive error handling and exception processing will enhance the operation, maintenance, and modification of the system. Interfaces to other packages, systems, and commercial software need extra attention to ensure that any upgrades to these external components do not corrupt the operation of the system. The Anomaly Control factor is mapped to the maintainability and evolvability quality areas. The full list of Anomaly Control attribute questions is contained in Table 6 in the Appendix.

4.3.7 Measuring Design Simplicity

Design simplicity is found when all modules are structured and implemented in a manner that lends itself readily to traceability and legibility. A static analysis of the code should

produce models that easily and accurately predict the code's dynamic behavior. The Design Simplicity factor is mapped to the maintainability and evolvability quality areas. The full list of Design Simplicity attribute questions is contained in Table 7 in the Appendix.

4.4 Measuring Quality Consistently

Before we can structure a fully repeatable measurement instrument that is independent of the evaluator and applicable across the full gamut of languages, environments, and architectures, we need to consider the features that will provide these types of capabilities.

4.4.1 The Scope Of Quality Factor Attributes

For each attribute we must define the scope under which the attribute will be evaluated and define a rigid scoring criteria with evaluation guidelines. The scope is defined as that portion of the systems being evaluated which will be examined in evaluating a particular attribute. For many attributes (especially those that can be evaluated in a fully automated way, such as complexity metrics) the scope is all of the code, while others (the more intellectually demanding tasks) are restricted to a "representative sample." An example of attributes that make use of sampled scopes are the questions with respect to the actual information content of embedded documentation. In these cases the scope has been defined as the seven largest, seven smallest, and seven average-sized files for each language found within the system. Thus if a system is composed of Ada, C, shell scripts, UIL, SQL, and 4-GL programs the assessment will review 126 files to assess the attributes that require a more manual assessment approach. If the results of this sampled examination are inconsistent across the sample set we quickly expand the sample to determine the source of the variance.

4.4.2 The Scoring Of Quality Factor Attributes

The evaluation guidelines used for instructing the analysts on how to assign a risk level scoring are deliberately worded at the conceptual level and avoid use of subjective terms and assumptions regarding the language(s), architecture, or target platform for the system being assessed. This allows the same framework of risk areas to be explored whether the system is on a UNIX platform using Ada and C or in an IBM MVS environment using COBOL and PL/1.

The scoring criteria and evaluation guidelines defined for each attribute use a normalized grading curve. The guidelines given to the analysts explicitly state to what level a system must address a particular feature to receive a given level of risk rating for that attribute. The rigidity of this scoring technique has been implemented and enforced to enhance the overall repeatability of the methodology attribute questions with their scoring criteria and evaluation guidelines as shown in Figure 6. This figure also displays a dialogue box from the automated Code Assessment Toolset (CAT) which was developed to help integrate the results from individual evaluators when assessments are conducted with teams. This level of automated support is not necessary to conduct an evaluation but its use accelerates the assessment process.

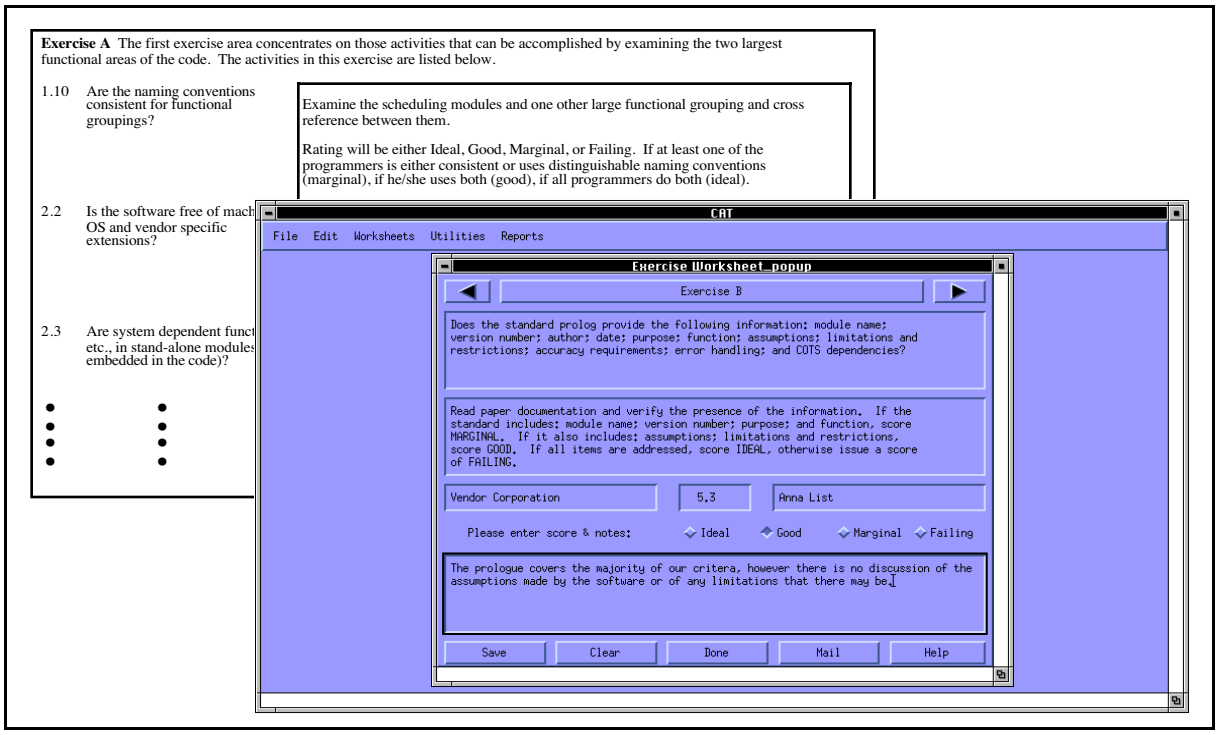


Figure 6: Example Attribute Questions with Sample Screen from the Automated Questionnaire

As stated earlier, the evaluation guidelines used for scoring are deliberately worded at the conceptual level and avoid use of subjective terms and assumptions regarding the language(s), architecture, or target platform for the system being assessed. To assess a system fairly and address a system specific context, the evaluator needs to determine how to answer the question for a particular language within a particular operating system. To aid in this a knowledge-base is used that contains supplemental evaluation context data. This data provides the analyst with lists of known "symptoms" and analysis tools that are appropriate for a given operating environment and language mix. An on-line version of this knowledge-base is made available to analysts through the help button on the CAT questionnaire screen (see Figure 6).

5. MEASURING QUALITY EFFICIENTLY

We have already made allusions to aspects of the automated support used during assessments of systems in the previous pages of this paper. In fact, a great deal of the effort that went into the selection and wording of the factor attribute questions was aimed at maximizing the level of instrumentation and automation possible within the assessment process. While it was not always easy to create a question about a factor in a manner that allows us to use an automated method to answer it, we were able to do this for a very large percentage of the assessment questions.

5.1 Use Of Automated Tools

The majority of our automation is based on programs that traverse complete directory structures while examining each of the files encountered. These programs are basically enhanced pattern matching search routines. A UNIX environment is used for our assessment and the paradigm of the UNIX "grep" functionality was chosen as the basis for these tools. To make our assessment process equally usable by those who understand UNIX and those who do not, our tools are intuitive to use and hide the platform dependencies and UNIX-ness of the tools from the analyst. A simple dialogue box appears for the analyst to select the type of search desired and to indicate within which directory the tool should search. While we have capitalized on our own knowledge of the file and directory searching capabilities of UNIX for our tools, similar capabilities could be assembled on whatever platform a group decided to use as their working environment.

5.2 Commercial Automated Tools

One may ask why we did not make use of commercial reverse-engineering and code analysis tools as the basis of our assessment automation. Early attempts were made to use such tools but the wide variety of languages used in the systems under analysis made this infeasible. Most of the commercial analysis tools are written for one language. While the vendors have multiple versions of the same tools (one can buy either an Ada, C, FORTRAN, or COBOL analysis tool), vendors do not offer a single tool that can be used to analyze multiple languages. Another problem with the commercial analysis tool offerings is that most are focused on only mainstream languages. Script languages, GUI code, 4-GLs, SQL, and variants of languages are not supported. Since our use of software assessments is for a wide audience of customers with systems in an unknown set of languages and platforms, it would be short sighted to base our automation support on products that are constrained to only mainstream languages.

The final problem with using commercial analysis tools lies in their assumption that all of the code for a system will be available to the tool. While this is not an insurmountable problem, it does severely impact the ability to conduct assessments quickly when we are given only part of the code for a system, as is frequently the case. To use the commercial tools in that situation one must create dummy routines for each routine the code calls that is not available or one must comment out the calls themselves. This can consume considerable time and will also alter the results of some portions of the analysis itself.

5.3 Our Own Automated Tools

Since our intention is to perform assessments against an unknown number of systems that are using an unknown number of languages, we decided it would be more efficient to concentrate our automation efforts on the creation of easily-used language-independent tools to which we could adapt new languages with language profiles for any language-specific capabilities needed. To date this approach has proven itself to be very successful. Currently our tools have been used on over four dozen different languages.

Handling multiple languages is partially a recognition problem. As mentioned above, when an assessment is arranged it is not certain what languages will be used in the system being reviewed, nor do we know how big the systems are or what percentage of the system are made up of each language. This can lead to a situation where the skills needed for a particular assessment are unknown in advance and the levels of effort are difficult to estimate. To improve the chances of successfully managing assessment efforts, a tool was developed that traverses the entire directory structure of a project, attempts to automatically recognize the types of languages it encounters, counts the numbers of files in each language, and counts the number of lines of code in each language. Any unrecognized files are also recorded and can be manually reviewed for identification purposes. The tool can be "taught" to recognize these new languages as well as count the lines of code in these new languages. By running this tool against the code of each new system we are asked to assess, we can quickly identify the languages and commercial tools used in the system as well as the level of effort that will be required for the assessment (recalling that we look at least 21 files from each language used in a system for the manually intensive questions).

5.4 Actual Assessment Techniques

We do our automatic and semi-automatic evaluation with all commonly used metrics where they are applicable, and then by computing densities of observed use of constructs with high assumed impact on risk for additional basic criteria. This approach is used for twenty of the criteria, shown listed below in the order of decreasingly automatic evaluation. The first six are automatically computed using locally developed tools. The remaining use one of the tools to produce data files that are reduced by (mostly) semi-automatic UNIX scripts or commands to give numerical inputs to density formulas that are simple to do by hand. Numbers in parentheses indicate the factor group and ordering in each group as shown for each question in the tabular lists in the Appendix. These 20 attributes are the following:

cyclomatic complexity (7.2)
essential complexity (7.1)
nesting complexity (7.9)
module size density (7.7)
embedded system dependencies (2.3)
effective standards usage (2.4)
communication channel consistency (1.8)
pathname isolation (2.1)
environmental variable consistency (4.11)
non-linear jump occurrence (7.4)
loop index modification (7.5)
communication isolation (7.6)
self modifying code occurrence (7.8)
Boolean parenthesization effectiveness (7.10)

duplicate condition avoidance (7.11)
system dependency isolation (3.3)
symbolic name usage (3.7)
symbolic name isolation (3.8)
global variable coupling (3.10)
dead code accounting (5.9)

The complexity and module size measures are evaluated using a single language sensitive construct counting program, with internal tables providing language specific support for Ada, C, C++, Pascal, PL/1 and some common 4-GL languages. The tool is also used, with tailoring for particular variants as needed, for evaluating FORTRAN and COBOL code. All the remaining risk factor densities are computed using a locally developed pattern matching tool that removes comments and string literals from pattern input to avoid false positive matches. The scripts and patterns can be made fully automatic for the first two criteria involving system dependencies for known standards. The remaining 14 have to be evaluated by tailoring patterns and performing reductions that often require special programming or analysis. The first six are marked with a code of (A) in the appropriate question table in the Appendix, and the remaining with a code of (B).

We do our less automatic evaluation of code quality using sampling and ranking based on 10% or 50% deviation from the ideal for the attributes addressed by the questions 1.7, 1.11, 1.12, 2.2, 2.5, 2.7, 3.5, 5.2, 5.3, 5.4, 5.5, 5.6, 5.8, 5.10, 5.11, 6.5 (indicated with code of C in the appropriate tables in the Appendix).

We measure documentation quality and some consistency issues using a qualitative judgment ranking scheme (excellent-good-marginal-failing) for the attributes addressed by the 31 questions marked with a code of D in the Appendix.

Our evaluation of some implementation consistency issues depends on a yes/no judgment for the attributes in the remaining 9 questions marked with a code of E in the Appendix.

6. SUMMARY AND FUTURE DIRECTIONS

When assessing a multi-language system, the SQAE applies a coordinated set of informational queries to the artifacts that make up the system and draws conclusions based on less than full interpretation. This enterprise is fraught with semantic peril. That this is not either hopeless or nonsensical is supported by the almost everyday (or at least every source selection) issues of system acquisition in the world of DOD. Life-cycle models and many more local (from requirements definition, specification to various levels of design) description activities have even greater issues related both to meaning and instrumentation. These problems are related to many traditional problems of software development and assessment. What looking at the issue as a logical data base problem does is introduce very promising applied information R&D methods to help solve a very important long term problem.

The SQAE methodology is most often used to provide an overall assessment of fielded systems that typically have not been well/fully understood or adequately measured to facilitate long term goals such as portability and maintainability. The SQAE provides this information quickly and with results that are in practice easily related to real issues for both software maintenance and project management personnel. The system as it stands is still highly

manual, but has achieved very useful practical acceptance in the DOD acquisition field. It is based on a conservative view of static analysis in a multi-language program data base, and its main innovations are from the point of view of multi-paradigm inferences that have somewhat non-first order properties. However, experience with the software quality assessment evaluation has convinced many natural skeptics of its validity and of the usefulness of the life-cycle quality insights that it produces.

As of February of this year (1996) we have used our assessment methodology and its support tools to assess over 31 million lines of code in over four dozen languages. Figure 7 indicates the major languages we have assessed. The "others" category contains over three dozen languages but only represents 2.2% of the code we have examined.

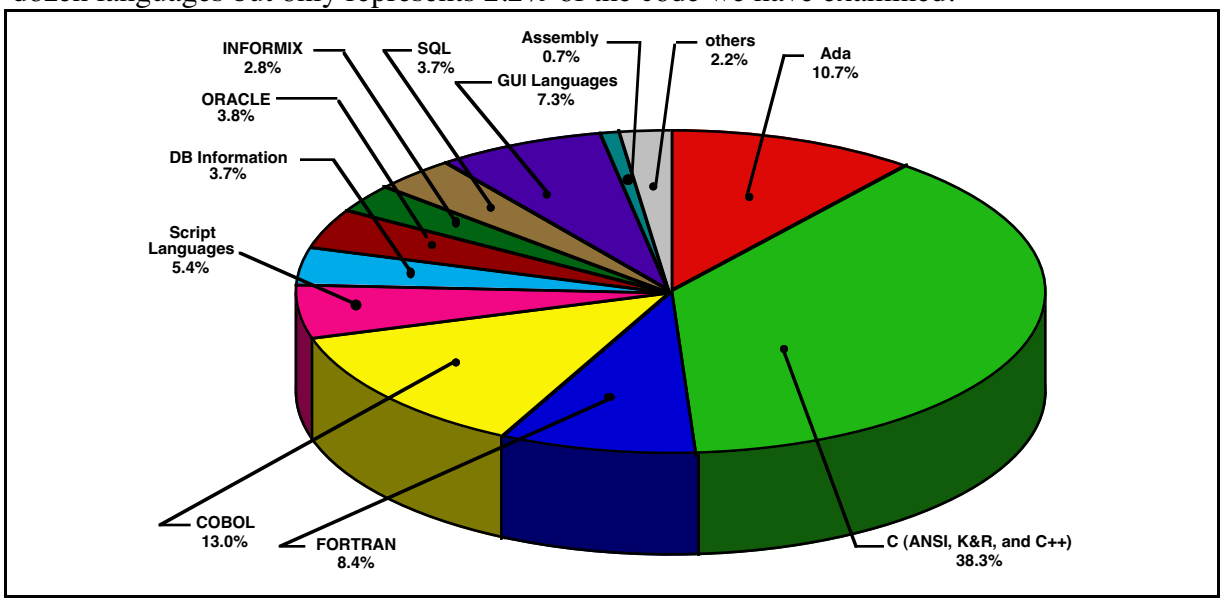


Figure 7: Distribution of Languages We Have Assessed

The SQAE approach uses logical data base facilities along with modern query-oriented views of knowledge acquisition to constructively address a serious problem is assessing software quality. The main insight is that iterative queries, building successively on the results of simple or quickly processed questions to harder and more complex information, can be used to control the assessment process and successively gather more and better facts about a software system. Advances in both methodology and computer hardware make this feasible. Human engineering using data base concepts make this usable.

This point of view relates to modern ideas of "conceptual software methodologies" such as object oriented and other ambitious design and re-engineering methodologies in terms of kind of "style of interaction". The hypothesis of many such tools is that consistency of a set of descriptions of functions and data over many cooperative validation activities will produce a self-contained system. What is needed to make this credible is the idea many partial orders will converge to some total order under appropriate constraints. These constraints are centered upon the fundamental concepts of persistence and coherence of interpreters (observers each with his/her own different perspective). In an activity like software quality assessment,

this has to do with querying from one set of assumptions and then making improved assumptions to determine some model that handles or incorporates all the partial results.

We see an evolution of rapid assessment following such a model as the apparent best hope of achieving valid quality assessment over time. Our current results have built up a sufficient data base over a variety of software applications and architectures to demonstrate the potential value of such an iterative approach. The SQAE methodology as it exists today provides the basic framework and user interface necessary to make use of this fundamental approach as the underlying technologies and heterogeneous data base facilities improve over the next decade.

ACKNOWLEDGMENTS: This work was funded by the MITRE Corporation.

BIBLIOGRAPHY

(1) Boehm, B. W., et al, "Characteristics of Software Quality", Document #25201-6001-RU-00, NBS Contract #3-36012, 28 December 1973.
(2) McCall, J. A., P. K. Richards, and G. F. Walters, "Factors in Software Quality", Volumes I, II and III, US. Rome Air Development Center Reports NTIS AD/A-049 014, NTIS AD/A-049 015 and NTIS AD/A-049 016, U. S. Department of Commerce, 1977.
(3) Kaposi, A. and B. Kitchenham, "The Architecture of Software Quality", Software Eng J, Vol. 2, No. 1, Jan. 87, 2-8.
(4) Kitchenham, B., "Towards a Constructive Quality Model", Software Eng J, Vol. 2, No. 4, July 87, 105-123.
(5) Kitchenham, B., L. M. Wood, and S. P. Davies, "Quality Factors, Criteria and Metrics", ESPRIT Report R1. 6. 1, REQUEST/ICL-bak/028/S1/QL-RP/01, 1985.
(6) "A Standard for Software Quality Metrics Methodology", IEEE Computer Society Standard P1061, 1991.
(7) Schneidewind, M. F., "A Methodology for Validating Software Metrics", IEEE Trans on SE, Vol 18, No. 5, May 92, 410-421.
(8) Grady, R. B., "Practical Results from Measuring Software Quality", CACM Vol. 36, No. 11, Nov. 93, 62-68.
(9) Stark, G., R. C. Durst, and C. W. Vowell, "Using Metrics in Management Decision Making" IEEE Computer, Vol. 27, No. 9, Sept. 94, 42-48.
(10) "Software Quality Framework As Implemented in QUES Release 1. 5", Prepared for RADC Software Quality Technology Transfer Consortium, Software Productivity Solutions, Inc., Indialantic, FL., Feb. 95.
(11) "Software Product Evaluation - Quality Characteristics and Guidelines for their Use", ISO/IEC Standard ISO-9126, 1991.
(12) Dromey , R. G., "A Model for Software Product Quality", IEEE Trans on SE, Vol. 21, No. 2, Feb. 95.
(13) Dick, R., "Subjective Software Measurement", Dept. of Computer Science, University of Strathclyde, Glasgow, 1993.
(14) "Software Maintainability Evaluation Guide", AFOTEC Pamphlet 99-102, Vol. 3, Kirtland AFB, New Mexico, 1994.
(15) McCabe, T. J., "A Complexity Measure", IEEE Trans on SE, Vol SE2, No. 4, 1976, 308-320.
(16) Halstead, M. H., "Elements of Software Science", New York: Elsevier North-Holland, 1977.
(17) Tamine, J. J., "On the Use of Tree-like Structures to Simplify Measures of Complexity", SIGLAN Notices, Vol. 18, No. 9, Sept. 82, 62-69.
(18) McCabe, T. J. and Butler, "Structured Testing using Cyclomatic Complexity", CACM, Dec. 89.
(19) Oman, P, and J. Hagemeister, "Metrics for Assessing Software System Maintainability", Proc. Conference on Software Maintenance, IEEE CS Press, Los Alamitos CA, 1992, 337-344.
(20) Rosenblum, B. D., "Rapid Code Evaluation", Software Development, April 94, 41-45.
(21) Weyuker, E., "Evaluating Software Complexity Measures", IEEE Trans on SE, Vol. 14, No. 9, Sept. 88.
(22) Harris, D. R., H. B. Reubenstein, and A. S. Yeh, "Reverse Engineering to the Architectural Level", Proceedings of 17th Annual Conference on Software Engineering, 1995, 186-195.
(23) Boyer, R. S. and J. S. Moore, "A Fast String Searching Algorithm", CACM, Vol. 20, No. 10, 1977.
(24) Campbell, R. H. and A. N. Haberman, "The Specification of Process Synchronization by Path Expressions", in Lecture Notes in Computer Science, Vol. 16, New York : Springer Verlag, 1974.
(25) Bartussek, W. and D. L. Parnas, "Using Traces to Write Abstract Specifications for Software Modules", in Information Systems Methodology, Lecture Notes in CS 65, New York: Springer-Verlag, 1978, 211-236.
(26) Parnas, D. L., J. Madey, and M. Iglewski , "Precise Documentation of Well-structured Programs", IEEE Trans on SE, Vol. 20, No. 12, Dec. 94, 948-976.
(27) Paul, S. and A. Prakash, "A Framework for Source Code Search using Program Patterns", IEEE Trans on SE, Vol. 20, No. 6, June 94, 463-475.
(28) Howden, W. E. and B. Wieand, "QDA - A Method for Systematic Informal Program Analysis", IEEE Trans on SE, Vol. 20, No. 6, June 94, 445-462.
(29) Albrecht, A. J. and J. E. Gaffney, "Software Function, Source Lines of Code and Development Effort Prediction: A Software Science Validation", IEEE Trans on SE, Vol. 9, No. 6, Nov. 83, 639-647.

APPENDIX - QUALITY FACTORS

Index No.	Attribute
1.1	Is there a representation of the design in the documentation? (D)
1.2	Is the software implemented in accordance with the representation in 1.1? (E)
1.3	Are there consistent global, unit, and data type definitions? (E)
1.4	Is there a definition of standard I/O handling in the documentation? (D)
1.5	Is there a consistent implementation of external I/O protocol and format for all units? (E)
1.6	Are data naming standards specified in the documentation? (D)
1.7	Are naming standards consistent across languages (e.g., Structured Query Language [SQL],Graphical User Interface [GUI], Ada, C, FORTRAN)? (C)
1.8	Are naming standards consistent across inter-process communications?
1.9	Is there a standard for function naming in the documentation? (D)
1.10	Are the naming conventions consistent for functional groupings? (E)
1.11	Are the naming conventions consistent for usage (e.g., I/O)? (C)
1.12	Are the naming conventions consistent for data type (e.g., constant boolean), etc.? (C)
1.13	Does the documentation establish accuracy requirements for all operations? (D)
1.14	Are there quantitative accuracy requirements stated in the documentation for all I/O? (D)
1.15	Are there quantitative accuracy requirements stated in the documentation for all constants? (D)

Table 1. Attributes of Consistency

Index No.	Attribute
	Software System Independence
2.1	Does the software avoid all usage of specific pathnames/filenames? (B)
2.2	Is the software free of machine, OS and vendor specific extensions? (C)
2.3	Are system dependent functions, etc., in stand-alone modules (not embedded in the code)? (A)
2.4	Are the languages and interface libraries selected standardized and portable? (i.e., ANSI…) (A)
2.5	Does the software avoid the need for any unique compilation in order to run (e.g., a custom post processor to "tweak" the code to run on machine X)? (C)
2.6	Is the generated code (i.e., GUI Builders) able to run without a specific support runtime component? (E)
	Machine Independence
2.7	Is the data representation machine independent? (C)
2.8	Are the commercial software components available on other platforms in the same level of functionality? (E)

Table 2. Attributes of Independence

APPENDIX - QUALITY FACTORS (continued)

Index No.	Attribute
3.1	Is the structure of the design hierarchical in a top-down design within tasking threads? (D)
3.2	Do the functional groupings of units avoid calling units outside their functional area? (D)
3.3	Are machine dependent and I/O functions isolated and encapsulated? (B)
3.4	Are interpreted code bodies (shell scripts and Fourth Generation Language [4-GL] scripts) protected from accidental or deliberate modification? (E)
3.5	Do all functional procedures represent one function (one-to-one function mapping)? (C)
3.6	Are all commercial software interfaces and APIs, other than GUIBuilders, isolated and encapsulated? (D)
3.7	Have symbolic constants been used in place of explicit ones? (B)
3.8	Are symbolic constants defined in an isolated and centralized area? (B)
3.9	Are all variables used exclusively for their declared purposes? (D)
3.10	Has the code been structured to minimize coupling to globally available data? (B)

Table 3. Attributes of Modularity

Index No.	Attribute
4.1	Is the documentation structured per the development plan? (D)
4.2	Does the design documentation depict control flow to the CSU/CSC level? (D)
4.3	Does the design documentation depict data flow? (D)
4.4	Do the design documents depict the task and system initialization hierarchy/relationships? (D)
4.5	Is the documentation adequately indexed (functionality can be easily located in the code)? (D)
4.6	Are all inputs, process and outputs adequately defined in the documentation? (D)
4.7	Does the documentation contain comprehensive descriptions of system/software interfaces? (D)
4.8	Does the documentation contain comprehensive descriptions of all internal operations? (D)
4.9	Does the documentation contain comprehensive descriptions and justification of all esoteric processing methods? (D)
4.10	Does the documentation establish a requirement for commenting global data within a software unit to show where the data is derived, the data composition and how data is used? (D)
4.11	Are all environmental variables and the default values clearly defined? (B)
4.12	Does the documentation explain the high-level functionality of the system? (D)
4.13	Does the documentation layout the functional allocation of the system to CPCIs? (D)
4.14	Are external interfaces and systems depicted in the documentation? (D)
4.15	Are the high-level flows of data into, out of, and through the system detailed? (D)
4.16	Does the documentation discuss/rationalize the usage of COTS, GOTS, and OS services? (D)

Table 4. Attributes of Documentation

APPENDIX - QUALITY FACTORS (concluded)

Index No.	Attribute
5.1	Does the documentation specify a standard prologue? (E)
5.2	Is a standard prologue consistently implemented? (C)
5.3	Does the standard prologue provide the following information: module name; version number; author; date; purpose; function; assumptions; limitations and restrictions; accuracy requirements; error handling; and COTS dependencies? (C)
5.4	Is a standard format for organizations of modules implemented consistently? (C)
5.5	Are comments set off from code and of consistent style throughout? (C)
5.6	Are comments accurate, and do they describe the "whats and whys?" (C)
5.7	Do code generation tools (screen builders, DBMS query tools, etc.) produce reusable "source code" that is documented? (E)
5.8	Are inputs, outputs, and side effects (if any) clearly detailed for each procedure? (C)
5.9	Is any and all dead code clearly offset and the reason for its existence documented? (B)
5.10	Has whitespace been managed for legibility and to allow identification of nesting constructs? (C)
5.11	Are function and variable names helpful in understanding the functionality of the code? (C)

Table 5. Attributes of Self-Descriptiveness

Index No.	Attribute
6.1	Is there a defined statement of techniques for error handling in the documentation? (D)
6.2	Is the vendor's standard implementation of error handling consistently applied? (D)
6.3	Is there a defined statement of techniques for tolerance of input data in the documentation? (D)
6.4	Is the vendor's standard implementation of input data handling consistently applied? (D)
6.5	Are tasking and rendezvous exceptions handled in an orderly fashion? (C)

Table 6. Attributes of Anomaly Control

Index No.	Attribute
7.1	Do all modules have singular entry and exit and avoid unconditional branching internally? (A)
7.2	Is the source code of low complexity (e.g., McCabe Cyclomatic…)? (A)
7.3	Is the DBMS interaction properly isolated? (D)
7.4	Does the code avoid making non-linear jumps into or out of loops? (B)
7.5.	Does the code avoid modifying loop indices? (B)
7.6	Do all IPCs communicate over unique channels? (B)
7.7	Is the code segmented into procedure bodies that can be understood easily? (A)
7.8	Is all use of self modifying code fully documented and justified? (B)
7.9	Have all procedures been structured to avoid excessive nesting? (A)
7.10	Have all boolean expressions been parenthesized to clarify mixed operator evaluations? (B)
7.11	Do all boolean expressions avoid referring to both a predicate and its complement? (B)

Table 7. Attributes of Design Simplicity

NOTE: The definition of the notation (A), (B), (C), (D), and (E) is provided in section 5.4 of the main paper.