Providing a Framework for Effective Software Quality Assessment

MAKING A SCIENCE OF RISK ASSESSMENT

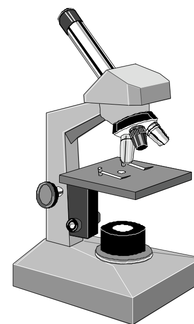
Robert A. Martin and Lawrence H. Shafer

To Be Presented At

The 6th Annual International Symposium of INCOSE

"Systems Engineering: Practices and Tools"

9-11 July 1996
Hosted by the New England Chapter of the
INternational Council on Systems Engineering



MITRE

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

PROVIDING A FRAMEWORK FOR EFFECTIVE SOFTWARE QUALITY MEASUREMENT: MAKING A SCIENCE OF RISK ASSESSMENT

Robert A. Martin and Lawrence H. Shafer
The MITRE Corporation
202 Burlington Road
Bedford, Massachusetts 01730

Abstract. 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 downsizing, 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 approach has been used successfully for more than 75 system evaluations as part of source selection or project management studies.

Introduction. 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.

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.

STRUCTURES FOR QUALITY ASSESSMENT

The original analysis of a structure for software quality assessment was done by B. Boehm and associates at TRW (Boehm, 1973) and incorporated by McCall and others in the Rome Air Development Center (RADC) report (McCall, 1977). 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 (Kaposi, 1987, Kitchenham, 1987, and Kitchenham, ESPRIT Report R1. 6. 1) in connection with the ESPRIT work that was input to the standards work producing the IEEE Standard P1061 (IEEE Computer Society, 1992). Work establishing the relationship 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, 1992), (Grady, 1993), and (Stark, 1994).

The model originally proposed in 1973 by Boehm and his coworkers used a basic outline for relating measurable properties to the perceived aspects of quality in a system is shown in Figure 1.

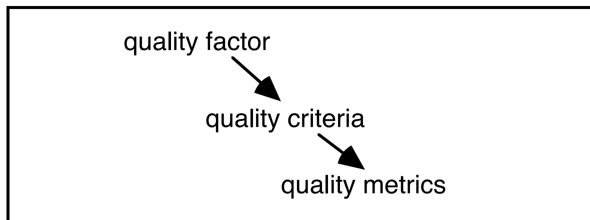


Figure 1. Boehm Quality Model

Direct and indirect measures were used to determine the level of agreement with a particular criterion that affects the fundamental quality factors. Integral to this approach was a list of primitive characteristics that influence a set of intermediate characteristics intended to be measured as quality factors. Factors included portability, reliability, efficiency, human engineering, testability, understandability, and modifiability.

The later RADC work included such criteria as generality, modularity, data commonality, communications commonality, access control and simplicity. It also decomposed independence into system and machine independence. 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) (RADC Technology Transfer Consortium, 1995). This framework includes 13 factors and expands 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.

The RADC model was expanded and modified by Kaposi and Kitchenham as part of a coordinated effort (called the European Strategic Programme for Research in Information Technology or ESPRIT consortium) of European researcher's 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. Their overall model can be described by the simple diagram in Figure 2.

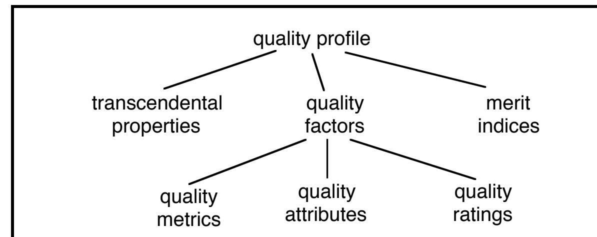


Figure 2. Quality Profile Model

Within this model 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 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.

The quality factors and merit indexes described the engineering concerns with defining the quality of a given software system. 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, thus we must look for both objective and subjective measures to capture all of the important issues surrounding the "quality" of a system. For this discussion we are defining a subjective quality factor as one that has little or no 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 Software Product Evaluation Standard, ISO-9126 (ISO/IEC, 1991). 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 effort to build profiles directly from metrics and ratings. (Dromey, 1995) reemphasized this distinction and reported significant repeatable results with his PASS (Program Analysis and Style System) methodology, based on a model with 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.

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 (Dick, 1993). He discusses use of objective techniques for combining subjective and objective measures in a controlled way. An important practical issue is to compose these metrics meaningfully into useful evaluation techniques to apply to real maintenance and other software life-cycle problems. Similarly, a way of distinguishing good and bad features for software by combining expert intuitions is the way quality is actually evaluated. 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.

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.

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 reworkings of it, suggests that 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 Complexity (McCabe, 1976), Halstead Software Science (Halstead, 1977) volumetric techniques, and a wide range of tailored structural and graph-based approaches developed over the last two decades. 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 (Tamine, 1982) and McCabe (McCabe, 1989) involved applying these notions successively to issues of complexity, maintenance and testing. Work on this area continues to develop models of local graphical behavior, parameterized to combine structural factors similar to Halstead volumes on an extended linear basis.

While quite interesting research, these suggestions still have not been accepted by enough of a

consensus of experts to be useful in improving or supplanting the earlier metrics for purposes of quality assessment. 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 titles are often quite similar (as noted above), the intent is to develop metrics related to specially parameterized models that are addressed to particular software system application areas and even life-cycle stages of those applications.

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 (Rosenblum, 1994). 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

produces results that can be used to determine major points of interest in programs, and then utilized 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 straight-forwardly, 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 (Weyuker, 1988), involving essentially their inability to distinguish significant differences in programs with the same complexity values. Concepts such as cohesion and coupling with inherently complicated 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. Encouraging results appear occasionally which suggest 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.

A SYSTEM FOR ASSESSING SOFTWARE QUALITY

The result of our integration of the above, with our own ideas, is discussed in the following paragraphs. Here we introduce seven "quality factors" that serve as the 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 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 3 and echo the types of relationships between quality areas and quality factors that appear in the IEEE Software Quality Metrics Methodology standard.

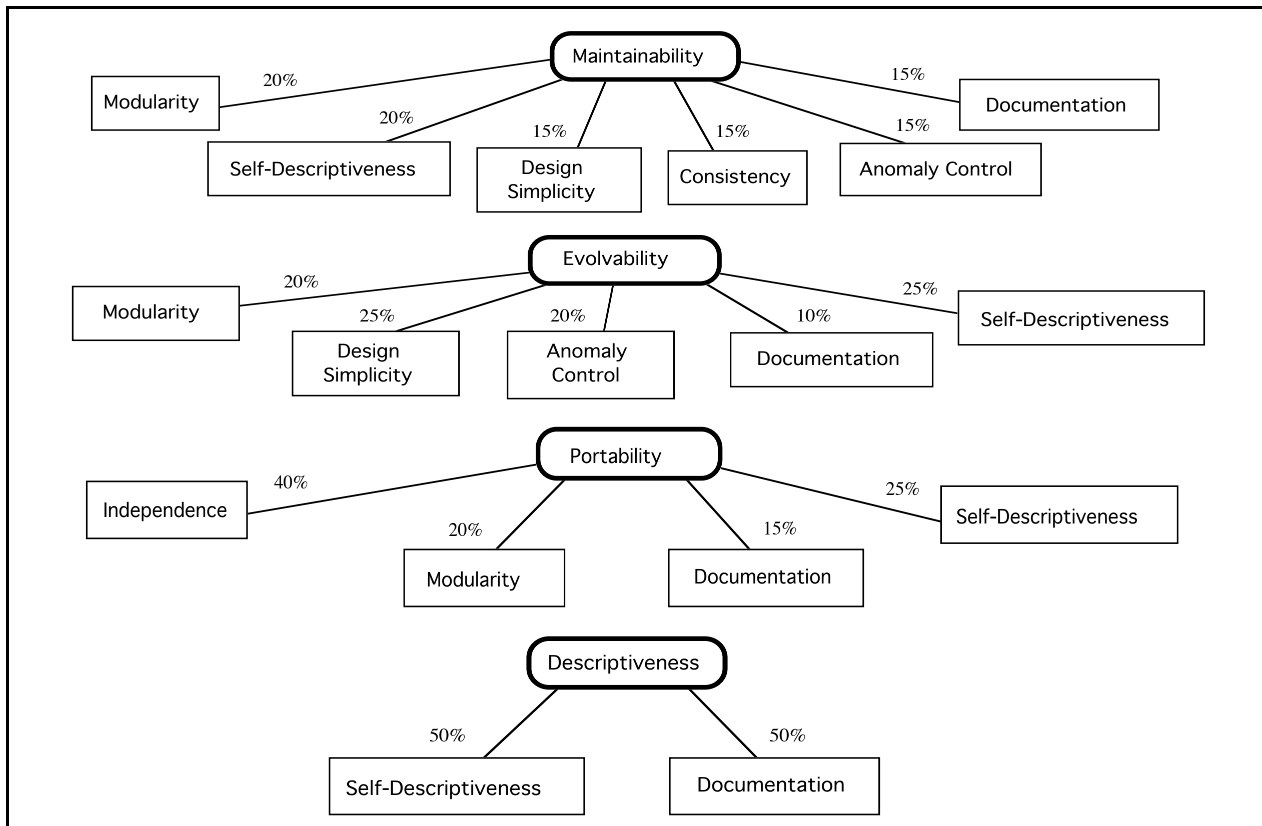


Figure 3: Quality Areas to Quality Factors Map

For each quality factor shown in Figure 3 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.

For example, Figure 3 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.

For each attribute of a quality factor we define the scope of the evaluation, the attribute weight, a rigid scoring criteria with evaluation guidelines, and a supplemental evaluation context. This means specifying a rigid scoring criteria with evaluation guidelines.

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.” 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.

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 4. 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.

Exercise A The first exercise area concentrates on those activities that can be accomplished by examining the two largest functional areas of the code. The activities in this exercise are listed below.

1.10 Are the naming conventions consistent for functional groupings?

Examine the scheduling modules and one other large functional grouping and cross reference between them.

Rating will be either Ideal, Good, Marginal, or Failing. If at least one of the programmers is either consistent or uses distinguishable naming conventions (marginal), if he/she uses both (good), if all programmers do both (ideal).

2.2 Is the software free of mach OS and vendor specific extensions?

2.3 Are system dependent funct etc., in stand-alone modules embedded in the code)?

CAT

File Edit Worksheets Utilities Reports

Exercise Worksheet_popup

Exercise B

Does the standard prolog provide the following information: module name; version number; author; date; purpose; function; assumptions; limitations and restrictions; accuracy requirements; error handling; and COTS dependencies?

Read paper documentation and verify the presence of the information. If the standard includes: module name; version number; purpose; and function, score MARGINAL. If it also includes: assumptions; limitations and restrictions, score GOOD. If all items are addressed, score IDEAL, otherwise issue a score of FAILING.

Vendor Corporation | 5.3 | Anna List

Please enter score & notes: Ideal Good Marginal Failing

The prologue covers the majority of our critera, however there is no discussion of the assumptions made by the software or of any limitations that there may be.

Save Clear Done Mail Help

Figure 4: Example Attribute Questions with Sample Screen from the Automated Questionnaire

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.

The actual "evaluation" of an attribute of a quality factor consists of two parts: the assigning of a numeric score in accordance with the attribute's scoring criteria; and, the creation of an annotation detailing the specific motivation behind the score awarded. When all attributes have been evaluated, the data is reduced, analyzed, and reviewed for use as the basis for the creation of a system wide quality risk profile that clearly defines the risk drivers and mitigators in the system. This allows the analysts to relate details of the implementation to high-level life-cycle engineering concepts quickly and easily and allows developers and program managers to see the long term ramifications of decisions and oversights that occurred during a system's initial development. The performance of a given system against the grading criteria can also be charted graphically at both the Quality Factor and Quality Area level as an aid in rapid identification of trends.

Summary. 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. 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. It is based is 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.

Acknowledgments. This work was funded by the MITRE Corporation.

REFERENCES

Boehm, B. W., et al, "Characteristics of Software Quality", Document #25201-6001-RU-00, NBS Contract #3-36012, 28 Dec. 1973.

Dick, R., "Subjective Software Measurement", Dept. of Computer Science, University of Strathclyde, Glasgow, 1993.

Dromey, R. G., "A model for software product quality", "*IEEE Transactions on Software Engineering*, Vol. 21, No. 2, Feb. 1995, 146-162.

Grady, R. B., "Practical results from measuring software quality", *CACM* Vol. 36, No. 11, Nov. 93.

Halstead, M. H., "Elements of Software Science", New York: Elsevier North-Holland, 1977.

IEEE Computer Society, "A Standard for Software Quality Metrics Methodology", IEEE Computer Society Standard P1061, 1992.

ISO/IEC, "Software Product Evaluation - Quality Characteristics and Guidelines for their Use", ISO/IEC Standard ISO-9126, 1991.

Kaposi, A. and B. Kitchenham, "The architecture of software quality", *Software Engineering Journal*, Vol. 2, No. 1, Jan. 1987, 2-8.

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.

Kitchenham, B., "Towards a Constructive Quality Model", *Software Engineering Journal*, Vol. 2, No. 4, July 1987, 105-123.

McCabe, T. J., "A Complexity Measure", *IEEE Transactions on Software Engineering*, Vol SE2, No. 4, 1976, 308-320.

McCabe, T. J. and Butler, "Structured testing using cyclomatic complexity", *CACM*, Dec. 1989.

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.

RADC Technology Transfer Consortium, "Software Quality Framework As Implemented in QUES Release 1. 5", Prepared by Software Productivity Solutions, Inc., Indialantic, FL., Feb. 1995.

Rosenblum, B. D., "Rapid Code Evaluation", Software Development, April 1994, 41-45