Quantitative Evaluation of Software Quality

B. W. Boehm
J. R. Brown
M. Lipow

TRW Systems and Energy Group

Keywords

software engineering
quality assurance
software quality
software measurement and evaluation
quality metrics
quality characteristics
management by objectives
software standards
software reliability
testing

Abstract

The study reported in this article establishes a conceptual framework and some key initial results in the analysis of the characteristics of software quality. Its main results and conclusions are:

- Explicit attention to characteristics of software quality can lead to significant savings in software life-cycle costs.
- The current software state of the art imposes specific limitations on our ability to automatically and quantitatively evaluate the quality of software.
- A definitive hierarchy of well-defined, well-differentiated characteristics of software quality is developed. Its higher-level structure reflects the actual uses to which software quality evaluation would be put; its lower-level characteristics are closely correlated with actual software metric evaluations that can be performed.
- A large number of software-quality-evaluation metrics have been defined, classified, and evaluated with respect to their potential benefits, quantifiability, and ease of automation.
- Particular software life-cycle activities have been identified that have significant leverage on software quality.

Most importantly, we believe that the study reported in this paper provides for the first time a clear, well-defined framework for assessing the often slippery issues

associated with software quality, via the consistent and mutually supportive sets of definitions, distinctions, guidelines, and experiences cited. This framework is certainly not complete, but it has been brought to a point sufficient to serve as a viable basis for future refinements and extensions.

I. Introduction

Why Evaluate Software Quality? Suppose you receive a software product that is delivered on time, within budget, and correctly and efficiently performs all its specified functions. Does it follow that you will be happy with it? For several reasons, the answer may be "no." Here are some of the common problems you may find:

1. The software product may be hard to understand and difficult to modify. This leads to excessive costs in software maintenance, and these costs are not trivial. For example, a recent paper by Elshoff [I] indicates that *75%* of General Motors' software effort is spent in software maintenance, and that GM is fairly typical of large industry software activities.
2. The software product may be difficult to use or easy to misuse. A recent GAO report [2] identified over $10,000,000 in unnecessary Government costs due to ADP problems; many of them were because the software was so easy to misuse.
3. The software product may be unnecessarily machine dependent, or hard to integrate with other programs. This problem is difficult enough now, but as machine types continue to proliferate, it will get worse and worse.

Major Software Quality Decision Points. There are a number of familiar situations in which it is possible to exert a strong influence on software quality, and for which it is important to have a good understanding of the various characteristics of software quality. Here are a few:

1. Preparing the quality specifications for a software product. Formulating what functions you need and how much performance (speed, accuracy) you need are fairly straightforward. Indicating that you also need to maintain ability or understandability is important, but much more difficult to formulate in some testable fashion.
2. Checking for compliance with quality specifications. This is essential if the quality specifications are to be meaningful. It can clearly be done with a large investment of good people, but this sort of checking is both expensive and hard on people's morale.
3. Making proper design trade-offs between development costs and operational costs. This is especially important because tight development budgets or

schedules cause projects to skimp on maintainability, portability, and usability.

4. Software package selection. Here again, many users need a relative assessment of how easily each package can be adapted to their installation's changing needs and hardware base.

The primary payoff of an increased capability to deal with software quality considerations would be an improvement in software maintenance cost-effectiveness. Too little of a quality (e.g., maintainability) translates directly into too much cost (i.e., the cost of life-cycle maintenance such as correction of errors and response to new user requirements). In view of this simple truth, it is rather surprising that more serious and definitive work has not been done to date in the area of evaluating software quality.

Previous Studies. Development of methods for evaluating software quality appears to have first been attempted in an organized way by Rubey and Hartwick [3]. Their method of Ref. 3 was to define code "attributes" and their "metrics," the former being a prose expression of the particular quality desired of the software, and the latter a mathematical function of parameters thought to relate to or define the attribute. Attributes such as "A_1 – mathematical calculations are correctly performed," or "A_5 – The program is intelligible," or "A_6 – The program is easy to modify" were each further analyzed to define less abstract, i.e., more concrete, attributes capable of being directly measured as to whether the attribute is present in software to some degree (on a scale of 0 to 100). Although a detailed breakdown of each major attribute was given, only a few metrics were defined and no particular application was mentioned.

A later study [4] performed by the authors included the formulation of metrics and their application in a controlled experiment to two computer programs (approximately 400 FORTRAN statements each) independently prepared to the same specification. In this study, only a limited number of attributes were considered, primarily those corresponding to attributes A_5 and A_6 of Ref. 3, mentioned previously.

In addition, there was a deliberate difference in quality emphasis in the two programming efforts: one was done by a "hotshot" programmer who was encouraged to maximize code efficiency, and one by a careful programmer who was encouraged to emphasize simplicity. The main results of the study were:

- Ten times as many errors were detected in the "efficient" program (over an identical series of 1000 test runs).
- The measures of program quality were significantly higher on the "simple" program; thus, they were good indicators of relative operational reliability, at least in this context.

Concurrently, other authors were recognizing the significance of characterizing and dealing explicitly with software quality attributes. Wulf [5] identified and provided

concise definitions of seven important and reasonably non-overlapping attributes: maintainability/modifiability, robustness, clarity, performance, cost, portability, and human factors. Abernathy, et al. [6] defined a number of characteristics of operating systems and analyzed some of the trade-offs between them. Weinberg [7] performed experiments in which several groups of programmers were given the same assignment but different success criteria (development speed, number of statements, amount of core used, output clarity, and program clarity). For each characteristic, the highest performance was achieved by the group given that characteristic as its success criterion.

An increasing number of people were recognizing the importance of software quality, and dealing implicitly with software quality attributes by addressing the establishment of "good programming practices." The book on programming style by Kemighan and Plauger [8] is the best example of this work. Also, a series of reports by CIRAD [9,10] on software maintainability provide some source material and some items such as the following checklist of practices and features enhancing software maintainability: conceptual grouping, top-down programming, modularity, meaningfulness, uniformity, compactness, naturalness, transferability, comments, parentheses, and names. A report by Warren on software portability [11] provides an excellent summary of alternative approaches to portability—e.g., simulation, emulation, interpretation, bootstrapping, and higher-order language features—with primary emphasis on the portability of language processors.

Recently, a number of initiatives have recognized the importance of explicitly considering quality factors in software engineering. For example, four presentations at the recent AIAA/ACM/IEEE/DOD Software Management Conference address the subject: DeRoze's presentation [12] identifies "software quality specifications and trade-off' as a high-priority DOD initiative; Kossiakoff [13] identifies seven attributes of "good" software specifications: Whitaker [14] identifies twelve explicit quality goals for new DOD programming languages; and Light's presentation [15] on software quality assurance identifies five important measures of software quality.

Key Issues in Software Quality Evaluation. Some of the major issues in software quality evaluation are the following:

1. Is it possible to establish definitions of the characteristics of software quality that are measurable and sufficiently nonoverlapping to permit software quality evaluation? This will be discussed in Section II, "Characteristics of Quality Code."
2. How well can one measure overall software quality or its individual characteristics? This will be discussed in Section III, "Measuring the Quality of Code."
3. How can information on software quality characteristics be used to improve the software life-cycle process? This will be discussed in Section IV.

II. Characteristics of Quality Code

Code is the realization of the software requirements and the detailed software design. It is the production article that directly controls the operations of the user's system. This section addresses the problem of characterizing the quality of the code itself. This section, and the following section on measuring the quality of code, are based primarily on a study performed on the subject by TRW for the National Bureau of Standards, [16] and on subsequent work in the area at TRW, including a Software Reliability Study performed for Rome Air Development Center. [17]

Initial Study Objectives and Conclusions. The initial objectives of the "Characteristics of Software Quality" study [16] were to identify a set of characteristics of software quality and, for each characteristic, to define one or more metrics such that:

1. Given an arbitrary program, the metric provides a quantitative measure of the degree to which the program has the associated characteristic, and
2. Overall software quality can be defined as some function of the values of the metrics.

Although "software" can have many components such as functional specifications, test plans, and operational manuals, this study concentrated on metrics that could be applied to FORTRAN source programs.

The initial conclusions of the study are summarized below. First, in software product development and evaluation, one is generally far more interested in where and how rather than how often the product is deficient. Thus, the most valuable automated tools for software quality analysis would generally be that which flagged deficiencies or anomalies in the program rather than just producing numbers. This has, of course, been true in the past for such items as compiler diagnostics; one would be justifiably irritated with a mere statement that "1.17 percent of your statements have unbalanced parentheses."

Second, we found that for virtually all the simple quantitative formulas, it was easy to find counterexamples that challenged their credibility as indicators of software quality. Some examples are given below.

1. A metric was developed to calculate the average size of program modules as a measure of structuredness. However, suppose one has a software product with n 100-statement control routines and a library of m 5-statement computational routines, which would be considered well structured for any reasonable values of m and n. Then, if n = 2 and m = 98, the average module size is 6.9

statements, while if m = 10 and n = 10, the average module size is 52.2 statements.

2. A "robustness" metric was developed for the fraction of statements with potential singularities (divide, square root, logarithm, etc.), which were preceded by statements that tested and compensated for singularities. However, often the operation is in a context that makes the singularity impossible; a simple example is that of calculating the hypotenuse of a right triangle:

```
Z = SQRT(X**2 + Y**2)
```

3. Some "self-descriptiveness" metrics were developed for the number of comment cards, the average length of comments, etc. However, it was fairly easy to recall programs with fewer and shorter comments that were much easier to understand than some with many extensive but poorly written comments.

Third, we concluded that the software field is still evolving too rapidly to establish metrics in some areas. In fact, doing so would tend to reinforce current practice, which may not be good. For example, the use of data clusters [18] and automatic type-checking would invalidate some reliability metrics based on checking for mixed-mode expressions, parameter range violations, etc.

Finally, we concluded that calculating and understanding the value of a single, overall metric for software quality may be more trouble than it is worth. The major problem is that many of the individual characteristics of quality are in conflict: added efficiency is often purchased at the price of portability, accuracy, understandability, and maintainability; added accuracy often conflicts with portability via dependence on word size; conciseness can conflict with legibility. Users generally find it difficult to quantify their preferences in such conflict situations. Another problem is that the metrics are generally incomplete measures of their associated characteristics. To summarize these considerations:

1. The desirable qualities of a software product vary with the needs and priorities of the prospective user.
2. There is, therefore, no single metric that can give a universally useful rating of software quality.
3. At best, a prospective user could receive a useful rating by furnishing the quality rating system with a thorough set of checklists and priorities.
4. Even so, since the metrics are not exhaustive, the resulting overall rating would be more suggestive than conclusive or prescriptive.
5. Therefore, the best use for metrics at this point is as individual anomaly indicators, to be used as guides to software development, test planning, acquisition, and maintenance.

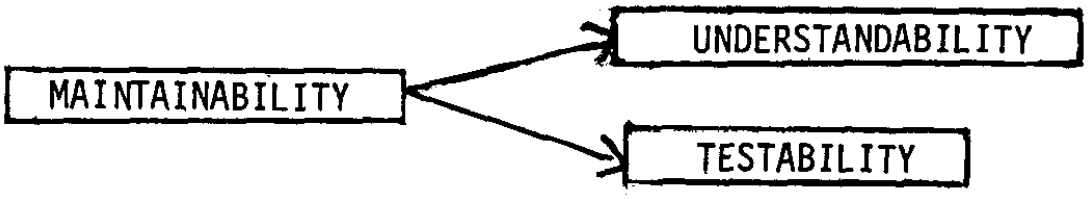
Identification and Classification of Quality Characteristics. Having reached the above conclusion, it was decided to develop a hierarchical set of characteristics and a set of anomaly-detecting metrics. Our plan and approach were as follows.

1. Define a set of characteristics that are important for software, and reasonably exhaustive and non-overlapping.
2. Develop candidate metrics for assessing the degree to which the software has the defined characteristic.
3. Investigate the characteristics and associated metrics to determine their correlation with software quality, magnitude of potential benefits of using, quantifiability, and ease of automation.
4. Evaluate each candidate metric with respect to the above criteria, and with respect to its interactions with other metrics: overlaps, dependencies, shortcomings, etc.
5. Based on these evaluations, refine the set of software characteristics into a set that is more mutually exclusive and exhaustive and supportive of software quality evaluation.
6. Refine the candidate metrics and realign them in the context of the revised set of characteristics.

The following initial set of software characteristics were developed and defined as a first step: (1) Understandability, (2) Completeness, (3) Conciseness, (4) Portability, *(5)* Consistency, (6) Maintainability, (7) Testability, (8) Usability, (9) Reliability, (10) Structuredness, (11) Efficiency. Definitions of these characteristics are given in the Appendix.

As a second step, we then defined candidate measurements of FORTRAN code (i.e., metrics, which would serve as useful indicators of the code's Understandability, Maintainability, etc. In doing so, we found that any measure of Understandability was also a measure of Maintainability—since any code maintenance requires that the maintainer understand the code. On the other hand, there were measures of Maintainability that had nothing to do with Understandability. For example, Testability features such as support of intermediate output and echo-checking of inputs are important to the retest function of software maintenance, but are unrelated to Understandability.

Thus, we began to find that the characteristics were related in a type of tree structure, e.g.,



in which the direction of the arrow represents a logical implication: if a program is Maintainable it must necessarily be Understandable and Testable; e.g., a high degree of Maintainability implies a high degree of Understandability and Testability.

We also began to find that there was another level of more primitive concepts below the level of Understandability and Testability. For example, if a program is Understandable, it is also necessarily Structured, Consistent, and Concise (three of the original characteristics), and, additionally, Legible and Self-Descriptive (two additional characteristics not implied by the three above). We were thus generating some additional characteristics and finding that the entire set of characteristics could be represented in a tree structure, in which each of the more primitive characteristics was a necessary condition for some of the more general characteristics. (This result anticipated step 5 of the plan outlined above.)

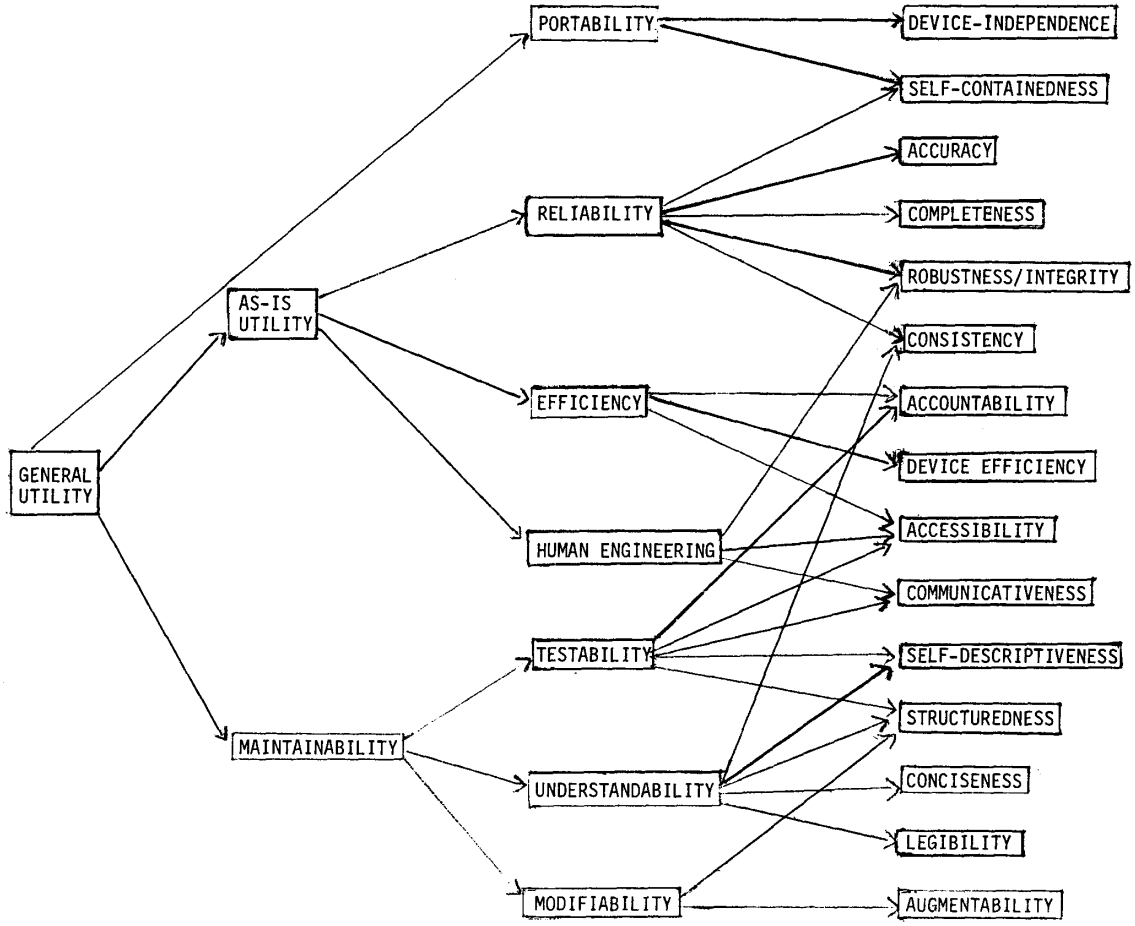
The resulting software quality characteristics tree is shown in Fig. 1. Its higher-level structure reflects the actual uses to which evaluation of software quality would be put. In general, when one is acquiring a software package, one is mainly concerned with three questions:

- How well (easily, reliably, efficiently) can I use it as is?
- How easy is it to maintain (understand, modify, and retest)?
- Can I still use it if I change my environment?

Thus, As-is Utility, Maintainability, and Portability are necessary (but not sufficient*) conditions for General Utility. As-is Utility requires a program to be Reliable and adequately Efficient and Human-Engineered, but does not require the user to test the program, understand its internal workings, modify it, or try to use it elsewhere. Maintainability requires that the user be able to understand, modify, and test the program, and is aided by good Human Engineering, but does not depend on the program's current Reliability, Efficiency, or Portability (except to the extent that the user's computer system is undergoing evolution).

* There are subsets of applications in which additional characteristics are necessary, applications that require computer security, for example. A more detailed discussion of characteristics of secure systems can be found in Ref. 19.

Fig. 1 – Software Quality Characteristics Tree



The lower-level structure of the characteristics tree provides a set of primitive characteristics that are also strongly differentiated with respect to each other, and that combine into sets of necessary conditions for the intermediate-level characteristics. For example:

- A program that does not initialize its own storage is not completely Self-Contained and therefore is not completely Portable even though it may be completely Device-Independent.
- A program using formats such as 12A6 is not completely Device-Independent, and is therefore not completely Portable, even though it may be completely Self-Contained.
- A program that is Device-Independent and Self-Contained but is not Accurate, Complete, Robust, Consistent, Accountable, Device-Efficient, Accessible, Communicative, Self-Descriptive, Structured, Concise, Legible, and Augmentable still satisfies the definition of Portability.

The primitive characteristics thus defined provide a much better foundation for defining quantitative metrics that can then be used to measure the relative possession of both the primitive and the higher-level characteristics. This can be done in terms that aid both in evaluating the utility of software products (at the high level) and in prescribing directions of needed improvements (at the primitive level). The definition and evaluation of such metrics is the subject of the next section.

III. Measuring the Quality of Code

Metrics. The term "metric" is defined as a measure of extent or degree to which a product (here we are concentrating on code) possesses and exhibits a certain (quality) characteristic. As described in the previous section, we found that, in fact, the development and refinement of both metrics and characteristics proceeded concurrently. Many metrics applicable to Fortran code were formulated and analyzed, leading in several iterations both to a refined set of metrics and to the generalized formulation of the hierarchical set of characteristics presented previously. We then had a basis for evaluating the usefulness of these entities, using the following criteria:

1. Correlation with Software Quality. For each metric purportedly measuring a given "primitive" characteristic, did it in fact correlate with our notion of software quality? Here, we mean roughly by positive correlation that most computer programs with high scores for a given metric would also possess the associated primitive characteristic. Clearly, a more precise statistical definition could be stated, but the evaluation was quite subjective at this point, and the more precise measures of correlation would need to await extensive data collection and judgments on many diverse computer programs. The following scale was used to rate each metric:

A – Very high positive correlation: nearly all programs with a high metric score will possess the associated characteristic
AA – High positive correlation; a good majority (say 75—90%) of all programs with a high metric score will possess the associated characteristic
U – Usually (say 50—75%) all programs with a high metric score will possess the associated characteristic
S – Some programs with high metric scores will possess the associated characteristic

2. Potential Benefit of Metrics. Some metrics provide very important insights and decision inputs for both the developer and potential users of a software product; others provide information that is interesting, perhaps indicative of potential problems, but of no great loss if the metric does not have a high score, even though highly correlated with its associated quality characteristic. The judgment as to its potential benefit is, of course, dependent on the uses for which the evaluator is assessing the product. The following scale of potential benefits was

defined:

5 – Extremely important for metric to have a high score; major potential troubles otherwise
4 – lmportant for metric to have a high score
3 – Fairly important for metric to have a high score
2 – Some incremental value for metric to have high score
1 – Slight incremental value for metric to have high score; no real loss otherwise.

3. Metric Quantifiability and Feasibility of Automated Evaluation. While a metric may rate at the top for both correlation with quality and potential benefit, it may be time-consuming or expensive to determine its numerical value. In fact, if to evaluate a metric requires an expert to read a program and make a judgment, the numerical value will generally provide much less insight than the understanding that the expert will pick up in the evaluation process. Furthermore, metrics requiring expert inspectors are extravagantly expensive to use. Therefore, one would prefer for large programs an automated algorithm that examines the program and produces a metric value (and preferably also a list of local exceptions to possession of the relevant characteristics). An intermediate capability that is often more feasible is an automated compliance checker, for which the user must provide a checklist of desired quality characteristics.

   In the evaluation, a judgment was made as to which combination of methods of quantification would provide the most cost-effective rating for the metric, using the following set of options:

AL – can be done cost-effectively via an automated algorithm
CC – can be done cost-effectively via an automated compliance checker if given a checklist (Code Auditor is such a tool, described in Section IV)
UI – requires an untrained inspector
TI – requires a trained inspector
EI – requires an expert inspector
EX – requires program to be executed

Automating some evaluations, such as counting average module length or checking for the presence of certain kinds of self-descriptive material, can be done in a fairly easy and straightforward fashion. Automating other evaluations, such as scanning the program globally for repeated subexpressions and guaranteeing that the components in the subexpressions are not modified between repetitions, are possible but more difficult. Others, such as judging the descriptiveness of the self-descriptive material as well as its presence, are virtually impossible to automate. Thus, some automated tools could provide useful but only partial support to quality evaluation, leaving the remainder to be

supplied by a human reader. The following scales were used to rate each metric with respect to ease and completeness of automated evaluation:

Ease of Developing Automated Evaluation

E – Easy to develop automated algorithm or compliance checker
M – Moderately difficult to develop automated algorithm or compliance checker
D – Difficult to develop automated algorithm or compliance checker

Completeness of Automated Evaluation

C – Algorithm or checker provides total evaluation of metric
P – Algorithm or checker provides partial evaluation of metric
I – Algorithm or checker provides inconclusive results

Evaluation of Metrics. The above criteria were applied to the candidate metrics developed in the study. Some examples are given in Table 1. Only a small fraction of the 151 candidate metrics [16] could be included here, but the ideas are amply illustrated. In Table 1 are shown, for each of six primitive characteristics, the evaluation of two of its associated metrics. An explanation of the ratings established for the first metric in the able is given below for clarity.

The metric (DI-1) was found to be highly correlated with Device Independence (rating "A"), and to be extremely important with respect to Device Independence (rating "5"). It was found that a combination of automated algorithm, execution, and a trained inspector would generally be most cost-effective for determining to what degree a software product possessed the characteristic (rating "AL + EX + TI"). An automated algorithm could check format statements for device dependence, e.g., 12A6 or F15.11 (more precision than most machines possess), and similarly flag extra-precise constants (e.g., PI = 3.14159265359). These checks would be easy to automate (rating "E"), but would only provide partial results (rating "P").

Table 1
EVALUATION OF QUALITY METRICS

Primitive Characteristics	Definition of Metrics	Correlation with Quality	Potential Benefit	Quantifi-ability	Ease of Developing Automated Evaluation	Completeness of Automated Evaluation
Device-Independence DI- 1	Are computations independent of computer word size for achievement of required precision or storage scheme?	A	5	AL + EX + TI	E	P
DI-2	Have machine-dependent statements been flagged and commented upon (e.g., those computations that depend upon computer hardware capability for addressing half words, bytes, selected bit patterns, or those that employ extended source language features)?	A	5	AL	M	P
Self-Containedness SC-1	Does the program contain a facility for initializing core storage prior to use?	A	5	AL	E	P
SC-2	Does the program contain a facility for proper positioning of input/output devices prior to use?	A	5	CC	E	P
Accuracy AR-1	Are the numerical methods used by the program consistent with application requirements?	A	5	TI		
AR-2	Are the accuracies of program constants and tabular values consistent with application requirements?	A	5	AL + TI	E	P
Completeness CP-l	Are all program inputs used within the program or their presence explained by a comment?	U	3	AL	E	C
CP-2	Are there no "dummy" subprograms referenced?	S	2	AL	E	C
Robustness R-1	Does the program have the capability to assign default values to non-specified parameters?	A	5	AL + TI	E	P
R-2	Is input data checked for range errors?	AA	5	AL + TI	E	P
Consistency CS-1	Are all specifications of sets of global variables (i.e., those appearing in two or more subprograms) identical (e.g., labeled COMMON)?	AA	4	AL	E	C
CS-2	Is the type (e.g., real, integer, etc.) of a variable consistent for all uses?	A	5	AL	E	P

Evaluation of Metrics Versus Project Error Experience. The best opportunity to evaluate the metrics and ratings exemplified above in Table 1 presented itself in the availability of an extensive database of software error types and experience in detecting

and correcting them, compiled by TRW for the Air Force CCIP-85 study. The initial segment of this database is presented in Table 2. It includes the classification of 224 software errors typed into 13 major categories:

1. Errors in preparation or processing of card input data
2. Tape handling errors
3. Disk handling errors
4. Output processing errors
5. Error message processing errors
6. Software interface errors
7. Hardware interface errors
8. Data base interface errors
9. User interface errors
10. Computation errors
11. Indexing and subscripting errors
12. Iterative procedures errors
13. Bit manipulation errors

Table 2. Evaluation of error-detecting capabilities (metrics) versus error type (first 12 of 224 error types)

	1	2	3	4	5	6	7	8
Software Phase / Error Type	Requirements	Design	Code	Development Test	Validation	Acceptance	Integration	Delivery
<u>Errors in Preparation or Processing of Card Input Data</u>								
1. Program expects parameter in different format than is given in Program Requirement Specification.		0 CS-13				F		
2. Program does not expect or accept a required parameter.		0 CS-13					F	
3. Program expects parameters in a different order than that which is specified.		0 CS-13					F	
4. Program does not accept data through the entire range that is specified.		0 CS-13				F		
5. Program expects parameter in units different from that which is specified.		0 CS-13				F		
6. Nominal or default value utilized by program in the absence of specific input data is different from that which is specified.		0					F	
7. Program accepts data outside of allowable range limits.		0						F
8. Program will not accept all data within allowable range limits.		0						F
9. Program overflows core tables with data that is within the allowed range.		0		CP-9				F
10. Program overflows allotted space in mass storage with data that is within the allowed range.		0		CP-9				F
11. Program executes first test case properly but succeeding test cases fail.		0				F		
12. Program expects parameter in a different location than specified.		0 CS-13				F		

0 = Error origin CS-N = Consistency-checking aid N applied at this phase would generally have detected error

F = Error found CP-N = Completeness-checking aid N applied at this phase would generally have detected error.

In the earlier study, all the errors of each type were analyzed to determine during which phase of the software development process they were typically (but not always) committed and where they were typically (but not always) found and corrected. The phases used in this analysis were:

1. Requirements Definition
2. Design
3. Code and Debug
4. Development Test
5. Validation
6. Acceptance
7. Integration
8. Delivery

In Table 2, for each error type, a "0" is placed in the column corresponding to the phase in which that type of error typically originated, and an "F" is placed in the column corresponding to the phase in which that type of error was typically found and corrected. To evaluate the applicability of each metric to error detection and correction, an additional item was estimated for each error type: the phase in which that type of error would most likely be detected and corrected using that metric.

Example. Line 1 in the table indicates that the typical error ot the type in which the "program expects a parameter in a different format than was given in the Program Requirement Specification," originated in the design phase (at least for the software projects analyzed in the CCIP-85 study). That type of error was typically corrected during the acceptance testing phase. However, if metric CS-l3 (an extension of metric-SD-I covering header commentary) had been computed, this type of error would typically have been caught and corrected in the design phase.

As is evident from this example, one result of the analysis was to identify several extensions of the previous metrics which would be effective in error detection and correction. For example, the CS-13 capability cited in the table implies the need for an automated tool that would scan standard software module header blocks. It would check the consistency of their information with respect to assertions in the header blocks about the nature of inputs and outputs, including:

- Data type and format
- Number of inputs
- Order of inputs
- Units
- Acceptable ranges
- Associated storage locations
- Source (device or logical file or record)

- Access (read-only, restricted access)

The CS-13 entries in the table indicate that if coding of the module had been preceded by such a module description with the assertions about its inputs and outputs, then an automated consistency checker would generally have caught the error before coding began.

Of the 12 types of card processing errors shown in Table 2, the CS-13 consistency checker would have caught 6. Overall, out of the 224 types of errors, this capability would generally have caught 18. The next most effective capability would perform checks on the consistency of the actual code with the module description produced during the design phase for capability CS-13 above. (For example, for each output assertion, it would check if the variable appeared on the left of an equal sign in the code, and perform a units check on the computation.) This capability would have caught 10 types of errors, but not until the initial code-scanning phase.

In general, as is seen in Table 3, the Consistency metrics were the most ettective aids to detecting software errors. Overall, they would have caught 34 of the 224 error types; their total phase gain (the sum of the number of phases that error detection was advanced by the metrics) amounted to 89, or an average gain of 2.5 phases per error type. The next most effective metrics were those for Robustness, followed by Self-Containedness and Communicativeness.

Table 3
ERROR CORRECTION EFFECTIVENESS OF METRICS

Metric/Primitive Characteristic	Error Types Corrected (No.)	Phase Gain (Total)
Consistency	34	89
Robustness	29	47
SeIf-containedness	15	28
Communicativeness	9	18
Structuredness	2	2
Self-descriptiveness	1	4
Conciseness	1	4
Accuracy	1	2
Accessibility	1	2

The main message of Table 3 is that the early application of automated and semiautomated Consistency, Robustness, and Self-Containedness checkers leads to significant improvements in software error detection and correction. This is an important conclusion, but it should not be too surprising, since Consistency, Robustness, and Self-Contained-ness are three of the primitive characteristics associated with Reliability.

Another useful consistency check was to compare a metric's error-correction potential with the estimate of a metric's potential benefit in Table 1. Satisfactorily, virtually all of the significant error-detecting and correcting metrics had maximum potential benefit ratings of 5, and none that contributed to error correction had ratings less than 3.

Of course, this was just a partial evaluation, but since testing occupies such a great proportion of a total software effort, the above evaluation has been a most useful one for helping us to decide which metrics should have high priorities for development and use. We have subsequently developed some of these metric checkers and used them with some success, and, in particular, the CS- 13 metric was used as a basis for the recently developed Design Assertion Consistency Checker described in Ref. 20.

IV. Using Quality Characteristics to Improve the Software Life-Cycle Process

The software life-cycle process begins with a system and software requirements determination phase, followed by successive phases for system design, detailed design, coding, and testing, and culminating in an operations and maintenance phase. There are four major means we have found for using the quality characteristics discussed above to improve the life-cycle process:

- Setting explicit software quality objectives and priorities
- Using software quality checklists
- Establishing an explicit quality assurance activity
- Using quality-enhancing tools and techniques

Explicit Software Quality Objectives and Priorities. The experiments reported in References 4 and 7 showed that the degree of quality a person puts into a program correlates strongly with the software quality objectives and priorities he has been given. Thus, if a user wants portability and maintainability more than code efficiency, it is important to tell the developer this, preferably in a way that allows the user to determine to what extent these qualities are present in the final product.

Probably the best way to accomplish this to date is through the practice of software quality benchmarking. Benchmarking is generally just used to determine device efficiency on a typical operational profile of user jobs, but it can be used similarly as an acceptance test or software-package selection criterion for other qualities also. Thus, for maintainability, one constructs a representative operational profile of likely modifications to the software, and measures how efficiently and effectively these modifications are made by the actual software maintenance personnel.

Used as an acceptance test, software quality benchmarking provides an explicit set of quality objectives for all participants throughout the various phases of the software development cycle. It has been used primarily to date in specifying hardware and

software reliability and availability objectives (with particular success in the Bell Labs' Electronic Switching System, for example), but has also been used successfully for other quality objectives.

Software quality benchmarking can and should be used also to evaluate alternative software products for procurement. In doing so, the level of effort expended in quality benchmarking should be proportional to the amount of use the product is expected to have, rather than to its price. More than once, we lost over $10,000 in software development and operational costs because of incomplete quality benchmarking in procurement of $1 ,000—$2,000 software products.

Software Quality Checklists. The quality metrics summarized above, and presented in detail in Reference 16, can be used as the basis for a set of software quality checklists. These can be used to support reviews, walk-throughs, inspections, and other constructive independent assessments of a software development product. Again, to date, these have been used primarily to support software reliability objectives, but can be used effectively for other software quality objectives. For example, Table 4 gives a portion of a checklist for judging the self-descriptiveness of a computer program, an important characteristic in evaluating the long-term costs of understanding, testing, and maintaining the program.

Table 4
PARTIAL CHECKLIST FOR JUDGING THE SELF-DESCRIPTIVENESS OF A SOFTWARE PRODUCT

A software product possesses self-descriptiveness to the extent that it contains enough information for a reader to determine or verify its objectives, assumptions, constraints, inputs, outputs, components, and revision status. Checklist:

a. Does each program module contain a header block of commentary that describes (1) program name, (2) effective date, (3) accuracy requirement, (4) purpose, (5) limitations and restrictions, (6) modification history, (7) inputs and outputs, (8) method, (9) assumptions, (10) error recovery procedures for all foreseeable error exits that exist?
b. Are decision points and subsequent branching alternatives adequately described?
c. Are the functions of the modules as well as inputs/outputs adequately defined to allow module testing?
d. Are comments provided to support selection of specific input values to permit performance of specialized program testing?
e. Is information provided to support assessment of the impact of a change in other portions of the program?
f. Is information provided to support identification of program code that must be modified to effect a required change?

g. Where there is module dependence, is it clearly specified by commentary, program documentation, or inherent program structure?
h. Are variable names descriptive of the physical or functional property represented?
i. Do uniquely recognizable functions contain adequate descriptive information (e.g., comments) so that the purpose of each is clear?
j. Are adequate descriptions provided to allow correlation of variable names with the physical property or entity that they represent?

Quality Assurance Activity. We are finding it increasingly advantageous, from both product quality and cost-effectiveness standpoints, to have an explicit quality assurance activity on our software projects. The manager of this activity generally reports to the project manager and is purposely held from producing any of the deliverable product in order to provide an independent view of the project. Tasks included in this quality activity are tailored to the project and depend upon the size and scope of the project. This approach has proven effective in ensuring that the project is responsive to the quality requirements of the customer and the particular system application. The responsibilities of the quality assurance activity generally include:

- Planning – Preparation of a software quality assurance plan that interprets quality program requirements and assigns tasks, schedules, and organizational responsibilities.
- Policy, Practice and Procedure Development – Preparation of standards manuals for all phases of software production, including requirements, design, coding, and test, tailored to specific project requirements. A key point here is attention to quality provisions early in the software life cycle.
- Software Quality Assurance Aids Development – Adaptation and development of manual and automated procedures for verifying compliance to software functional and performance requirements and project quality standards.
- Audits – Review of project procedures and documentation for compliance with software development plan standards, with follow-up and documentation of corrective actions.
- Test Surveillance – Reporting of software problems, analysis of error causes, and assurance of corrective action.
- Records Retention – Retention of design and software problem reports, test cases, test data, logs verifying quality assurance reviews, and other actions.
- Physical Media Control – Inspection of disks, tapes, cards, and other program-retaining media for verification at all times of physical transmittal or retention, and assurance that contents are not destroyed or altered by environment or mishandling.

To date, most of these responsibilities involve considerations of reliability and software product compliance to standards and the software requirements specification.

However, the quality assurance activity can also be a useful focal point for assuring that the product possesses the other characteristics of software quality, such as Maintainability and Portability.

Quality-Enhancing Tools and Techniques. Practically every software tool and technique—cross-reference generators, flow charters, configuration management procedures, software monitors—supports some kind of quality enhancement with respect to at least one quality characteristic. Here, we concentrate on several tools and techniques having particularly high leverage for software quality enhancement, first those that apply to software requirements and design specifications and then those that apply to code.

Quality-Enhancing Tools and Techniques: Requirements and Design. During the requirements and design phases, some assurance that desired quality characteristics are present can be obtained by using guidelines and detailed checklists. Here, of course, the primary objective is not to obtain a numerical measure of the extent to which a quality characteristic is present, but to identify problems of varying levels of criticality with which we need to deal. A great deal of software quality leverage is gained by using machine-analyzable software specifications and automated aids to analyze them for Consistency, Completeness, etc, such as ISDOS [21] and SREP [22,23] provide for software requirements. These systems provide a significantly increased assurance of specification quality over manual methods, giving considerable leverage for limiting the cost of software errors during both testing and maintenance. This is because a majority of errors arise owing to faulty expression of requirements and incomplete design [20], and are much less expensive to correct during the early phases of production than during subsequent phases.

One of the biggest sources of software problems stems from ambiguity in the software requirements specifications. A number of different groups—designers, testers, trainers, and users—must interpret and operate with the requirements independently. If their interpretations of the requirements are different, many development and operational problems will result.

One of the best counters to this problem is a review to make sure that the requirements are Testable. For example, consider the pairs of specifications below.

Nontestable	Testable
1. Accuracy shall be sufficient to support mission planning	1. Position error shall be: $\leq$ 50' in the horizontal $\leq$ 20' in the vertical
2. System shall provide real-time response to status queries	2. System shall respond to: Type A queries in $\leq$ 2 sec Type B queries in $\leq$ 10 sec Type C queries in $\leq$ 2 min

3. Terminate the simulation at an appropriate shift break	3. Terminate the simulation after 8 hours of simulated time

It is clear that the specifications on the right are not only more testable but also less ambiguous and better suited as a baseline for designing, costing, documenting, operating, and maintaining the system.

There is one technique for explicitly analyzing such quality considerations during the requirements phase that has been reasonably successful on small-to-medium projects. This is the Requirements-Properties Matrix: a matrix whose columns consist of the individual functional requirements and whose rows consist of the major qualities (or properties) desired in the software product (or vice versa). The elements of the matrix consist of additional specifications that arise when one considers the quality implications of each requirement. For example, consider the third pair of requirements above when treated in a requirements-properties matrix as in Fig. 2. It is clear that the resulting specifications will lead to a higher quality software product. Some additional effort would be necessary to achieve the enhanced product, but if the program is to have a good deal of use and maintenance, the effort will pay off in reduced life-cycle costs.

Fig. 2 – Portion of a Requirements-Properties Matrix

Requirement / Property	Terminate the simulation at an appropriate shift break	…
Testability	Terminate the simulation after 8 hours of simulated time	…
Modifiability	Allow user to specify termination time as an input parameter, with a default value of 8 hours	…
Robustness	Provide an alternate termination condition in case the time criterion cannot be reached.	...
⋮	⋮	⋮

Quality-Enhancing Tools and Techniques: Code. As shown in the detailed characteristics tree of Fig. 1, code Structuredness is one of the necessary primitive characteristics for Testability, Understandability, or Modifiability; all of the latter are necessary for Maintainability. A set of allowable FORTRAN constructs for the basic control structures SEQUENCE, IFTHENELSE, CASE, DOWHILE, and DOUNTIL [24] were developed as a standard for Structuredness on a large real-time software project. An automated FORTRAN source code scanning program called STRUCT was developed and is regularly used to determine for each routine whether it is a properly nested

combination of the allowable constructs, and when violations are recognized, the code causing the violation is identified and a diagnostic issued.

The discipline invoked by this quality requirement on the particular project met with a certain amount of resistance and disgruntlement by programmers. Functional team leaders were somewhat dismayed at first since routines previously coded before the standard had been required needed to be redone to a great extent, which, consequently, strained labor cost budgets and made the original schedules difficult to meet. Subsequently, however, in a survey of programmers and their supervisors, most were of the opinion that maintenance costs would be reduced, in addition to expressing positive opinions on "quality of code," including consistency and understandability. The opinion was also expressed that had the standard been invoked in the first place, most of the development problems would have been avoided.

Subsequent evaluations of software errors observed in testing on the referred-to project have shown an extremely low rate, believed to be partly attributable to the application of the Structuredness standard, although the requirement to exercise all branches of a routine prior to turnover for integration testing was found to be much more influential in reducing the number of errors.

The structuring standard is simply one of over 30 other coding standards formulated for this software project, and automatically checked by a FORTRAN source code analysis tool called CODE AUDITOR. This tool determines whether these standards are violated, and shows the source code location of the violation in a diagnostic printout. Many of the primitive characteristics of Fig. 1 are measurable by CODE AUDITOR. For example, Self-Descriptiveness is measured in part by checking for the presence of standard module header commentary cards to explain transfers of control. Consistency is measured in part by checks for mixed-mode expressions and compliance with standards for parameter passing.

In the future, some of these will be done more efficiently through standard language features for structured programming, data typing, etc. Yet, there will remain a need for automatic post-scanning of code to assure compliance to local standards (e.g., naming conventions) and to check for partial indicators of potential quality problems.

Of course, as is evident from the ratings in Table 1, there are some evaluations of code (and design) quality that require trained humans to perform. Reference 25 reports some experiences on large software projects with emphasis on the relative merit of a variety of techniques involving both human inspectors and automated tools for controlling and measuring software quality. One particularly valuable technique in this regard is that of the design or code inspection [26] and its "cousin" technique, the structured walk-through. In our analysis [17] of software errors found in the validation phase of one large project, we determined that 58% of them could have been eliminated early by appropriate design inspection techniques. Fagan's results [26] on one fairly well-

controlled project indicate that the extra effort involved in performing inspections paid off in a 23 percent reduction in operational errors.

V. Conclusions

Explicit attention to characteristics of software quality can lead to significant savings in software life-cycle costs (Section I).

The current software state of the art imposes specific limitations on our ability to automatically and quantitatively evaluate the quality of software (Section II).

A definitive hierarchy of well-defined, well-differentiated characteristics of software quality has been developed. Its higher-level structure reflects the actual uses to which software quality evaluation would be put; its lower-level characteristics are closely correlated with actual software metric evaluations that can be performed (Section II).

A large number of software quality-evaluation metrics have been defined, classified, and evaluated with respect to their potential benefits, quantifiability, and ease of automation (Section III).

Particular software life-cycle activities have been identified that have significant leverage on software quality (Section IV). These include:

- Setting explicit software quality objectives and priorities
- Performing software quality benchmarking
- Using software quality checklists
- Establishing an explicit quality assurance activity
- Using machine-analyzable software specifications
- Ensuring testable software requirements
- Using a requirements-properties matrix
- Establishing standards, particularly for structured code
- Using an automated code auditor for standards-compliance checking
- Performing design and code inspections

Most importantly, we believe that the study reported in this paper provides for the first time a clear, well-defined framework for assessing the often slippery issues associated with software quality, via the consistent and mutually supportive sets of definitions, distinctions, guidelines, and experiences cited [18]. This framework is certainly not complete, but it has been brought to a point sufficient to support the evaluation of the relative cost-effectiveness of prospective code-analysis tools presented here, and to serve as a viable basis for future refinements and extensions.

References

I. Elshoff, J. L., An Analysis of Some Commercial PL/I Programs. IEEE Transactions on Software Engineering, pp. 113–120, June, 1976.

2. Improvements Needed in Managing Automated Decision making &v computers Throughout the Federal Government, U.S. General Accounting Office, April 23, 1976.

3. Rubey, R. J., and R. D. Hartwick, Quantitative Measurement of Program Quality, in Proceedings of ACM National Conference, pp. 671–677, 1968.

4. Brown, J. R., and M. Lipow, The Quantitative Measurement of Software Safety and Reliability, revised from TRW Report No. SDP-I776, August 1973, TRW Software Series (in press August 1976).

5. Wulf, W. A., Programming Methodology, in Proceedings of a Symposium on the High Cost of Software, J. Goldberg (Ed.), Stanford Research Institute, September 1973.

6. Abernathy, D. H., et al., "Survey of Design Goals/or Operating Systems," Georgia Institute of Technology Report GTIS-72-04.

7. Weinberg, G. M., The Psychology of Improved Programmer Performance, Datamation, 82–85, November, 1972.

8. Kernighan, B. W., and P. J. Plauger, The Elements of Programming Style, McGraw-Hill, 1974.

9. A Study of Fundamental Factors Underlying Software Maintenance Problems, CIRAD, Inc., December 1971.

10. Research Toward Ways of Improving Software Maintenance, CIRAD, Inc., January 1973.

11. Warren. J., Software Portability, Stanford University Digital Systems Laboratory, Technical Note No. 48, September 1974.

12. DeRoze, B. C., "DOD Defense System Software Management Program," Abridged Proceedings from the Software Management conference, 1976 (obtainable through Los Angeles Section AIAA).

13. Kossiakoff, A., and T. P. Sleight, "Software Requirements Analysis and Validation," ibid.

14. Whitaker, W. A., "DOD Common High Order Language (HOL) Program," ibid.

15. Light, W., "Software Reliability/Quality Assurance Practices," ibid.

16. Boehm, B. W., J. R. Brown, H. Kaspar, M. Lipow, G. J. MacLeod, and M. J. Merritt, Characteristics of Software Quality. TRW Software Series TRW-SS-73-09, December 1973.

17. Thayer, T. A., et al., Software Reliability Study (Final Technical Report), TRW Report No. 76-2260.1.9-5, March 1976.

18. Liskov, B. H., and S. N. Zilles, Programming with Abstract Data Types, ACM SIGPLAN Notices, 50–59, April, 1974.

19. Stepczyk, F. M., Requirements for Secure Operating Systems, TRW Software Series, TRW-SS-74-05, June 1974.

20. Boehm, B. W., R. K. McClean, and D. B. Urfrig, "Some Experiences with Automated Aids to the Design of Large-Scale Software," IEEE Transactions on Sof ware Engineering, 125–133, March, 1975.
21. Teichroew, D. and H. Sayari, "Automation of System Building," Datamation, August, 25—30, 1971.
22. Alford, M. W., Jr., "A Requirements Engineering Methodology for Real-Time Processing Requirements," Proceedings of IEEE-ACM Second International conference on Software Engineering, October 1976.
23. Bell, T. E., and D. C. Bixler, "An Extendable Approach to Computer-Aided Software Requirements Engineering," ibid.
24. Mills. H. D., Mathematical Foundations of Structured Programming, IBM-FSD Report-72-6012, 1972.
25. Brown, J. R., Proceedings of the AJIE Conference on Software, Washington, DC, July 19-2 1,1976.
26. Fagan, M. E., Design and Code Inspections and Process Control in the Development of Programs, IBM- TR-21-572, December 1974.

Appendix

Definitions of Quality Characteristics

ACCESSIBILITY: Code possesses the characteristic of *accessibility* to the extent that it facilitates selective use of its parts. (Examples: variable dimensioned arrays, or not using absolute constants.) *Accessibility* is necessary for *efficiency*, *testability*, and *human engineering*.

ACCOUNTABILITY: Code possesses the characteristic of *accountability* to the extent that its usage can be measured.

This means that critical segments of code can be instrumented with probes to measure timing, whether specified branches are exercised, etc. Code used for probes is preferably invoked by conditional assembly techniques to eliminate the additional instruction words or added execution times when the measurements are not needed.

ACCURACY: Code possesses the characteristic of *accuracy* to the extent that its outputs are sufficiently precise to satisfy their intended use. Necessary for *reliability*.

AUGMENTABILITY: Code possesses the characteristic of *augmentability* to the extent that it can easily accommodate expansion in component computational functions or data storage requirements. This is a necessary characteristic for *modifiability*.

COMMUNICATIVENESS: Code possesses the characteristic of *communicativeness* to the extent that it facilitates the specification of inputs and provides

outputs whose form and content are easy to assimilate and useful. *Communicativeness* is necessary for *testability* and *human engineering*.

COMPLETENESS: Code possesses the characteristic of *completeness* to the extent that all its parts are present and each part is fully developed.

This implies that external references are available, required functions are coded and present as designed, etc.

CONCISENESS: Code possesses the characteristic of *conciseness* to the extent that excessive information is not present.

This implies that programs are not excessively fragmented into modules, overlays, functions and subroutines, nor that the same sequence of code is repeated *n* numerous places, rather than defining a subroutine or macro; etc.

CONSISTENCY: Code possesses the characteristic of *internal consistency* to the extent that it contains uniform notation, terminology, and symbology within itself, and *external consistency* to the extent that the content is traceable to the requirements.

Internal consistency implies that coding standards are homogeneously adhered to; e.g.. comments should not be unnecessarily extensive or wordy at one place, and insufficiently informative at another; the number of arguments in subroutine calls should match the subroutine header, etc. External consistency implies hat variable names and definitions, including physical units, are consistent with a glossary; or, there is a one-one relationship between functional flowchart entities and coded routines or modules; etc.

DEVICE-INDEPENDENCE: Code possesses the characteristic of *device-independence* to the extent it can be executed on computer hardware configurations other than its current one. Clearly, this characteristic is a necessary condition for *portability*.

EFFICIENCY: Code possesses the characteristic of *efficiency* to the extent that it fulfills its purpose without waste of resources.

This implies that choices of source code constructions are made in order to produce the minimum number of words of object code; or that, where alternate algorithms are available, those taking the least time are chosen; or that information-packing density in core is high, etc. Of course, many of the ways of coding efficiently are not necessarily efficient in the sense of being cost-effective, since portability, maintainability, etc., may be degraded as a result.

HUMAN ENGINEERING: Code possesses the characteristic of *human engineering* to the extent that it fulfills its purpose without wasting the users' time and

energy, or degrading their morale. This characteristic implies *accessibility*, *robustness*, and *communicativeness*.

LEGIBILITY: Code possesses the characteristic of *legibility* to the extent that its function is easily discerned by reading the code. (Example: complex expressions have mnemonic variable names and parentheses even if unnecessary.) *Legibility* is necessary for *understandability*.

MAINTAINABILITY: Code possesses the characteristic of *maintainability* to the extent that it facilitates updating to satisfy new requirements or to correct deficiencies.

This implies that the code is understandable, testable, and modifiable; e.g., comments are used to locate subroutine calls and entry points, visual search or locations of branching statements and their targets is facilitated by special formats, or the program is designed to fit into available resources with plenty of margins to avoid major redesign, etc.

MODIFIABILITY: Code possesses the characteristic of *modifiability* to the extent that it facilitates the incorporation of changes, once the nature of the desired change has been determined. Note the higher level of abstractness of this characteristic as compared with *augmentability*.

PORTABILITY: Code possesses the characteristic of *portability* to the extent that it can be operated easily and well on computer configurations other than its current one.

This implies that special language features, not easily available at other facilities, are not used, that standard library functions and subroutines are selected for universal applicability, etc.

RELIABILITY: Code possesses the characteristic *reliability* to the extent that it can be expected to perform its intended functions satisfactorily

It implies that the program will compile, load, and execute, producing answers of the requisite accuracy; and that the program will continue to operate correctly, except for a tolerably small number of instances, while in operational use. It also implies that it is complete and externally consistent, etc.

ROBUSTNESS: Code possesses the characteristic of *robustness* to the extent that it can continue to perform despite some violation of the assumptions in its specification.

This implies, for example, that the program will properly handle inputs out of range, or in different format or type than defined, without degrading its performance of functions not dependent on the nonstandard inputs.

SELF-CONTAINEDNESS: Code possesses the characteristic of *self-containedness* to the extent that it performs all its explicit and implicit functions within itself. Examples of implicit functions are initialization, input checking, diagnostics, etc.

SELF-DESCRIPTIVENESS: Code possesses the characteristic of *self-descriptiveness* to the extent that it contains enough information for a reader to determine or verify its objectives, assumptions, constraints, inputs, outputs, components, and revision status. Commentary and traceability of previous changes by transforming previous versions of code into non-executable but present (or available by macro calls) code are some of the ways of providing this characteristic. *Self-descriptiveness* is necessary for both *testability* and *understandability*.

STRUCTUREDNESS: Code possesses the characteristic of *structuredness* to the extent that it possesses a definite pattern of organization of its interdependent parts.

This implies that evolution of the program design has proceeded in an orderly and systematic manner, and that standard control structures have been followed in coding the program, etc.

TESTABILITY: Code possesses the characteristic of *testability* to the extent that it facilitates the establishment of verification criteria and supports evaluation of its performance.

This implies that requirements are matched to specific modules, or diagnostic capabilities are provided, etc.

UNDERSTANDABILITY: Code possesses the characteristic of *understandability* to the extent that its purpose is clear to the inspector.

This implies that variable names or symbols are used consistently, modules of code are self-descriptive, and the control structure is simple or in accordance with a prescribed standard, etc.

USABILITY (AS-IS UTILITY): Code possesses the characteristic of *usability* to the extent that it is reliable, efficient, and human engineered.

This implies that the function performed by the program is useful elsewhere, is robust against human errors (e.g., accepts either integer or real representations for type real variables), or does not require excessive core memory, etc.