Quantitative measurement of program quality*

by RAYMOND J. RUBEY and
R. DEAN HARTWICK
Logicon, Incorporated
San Pedro, California

INTRODUCTION

A black-box approach typifies current software quality assurance procedures: a program is good it it satisfies certain operating specifications. While it is common to manage the development of software under quality assurance systems previously devised for hardware, the tools of measurement are not transferable owing to the very basic differences in the nature of hardware and software. In the absence of specific, applicable quantitative measurement tools there exists no means of defining the desired level of quality in a computer program, where quality is considered as something beyond correct program functioning, nor of ascertaining whether the desired level has been achieved. A user should be able to specify precisely how good a product he wishes to buy, such things as how easy the program should be to run production with and how easily it can be modified. Rarely can the user even discuss these factors, much less specify the extent of their importance to him.

The problems in achieving and measuring quality in spaceborne software—that is, software which operates on a vehicle-borne aerospace computer—are particularly acute because this type of software has stringent requirements to be error-free, functionally precise, and responsive to modifications. A study of quality in spaceborne software performed for the Air Force Space and Missile Systems Organization forms the basis of this discussion. Although the study has its primary emphasis in the field of spaceborne software, the approach taken and the techniques developed are applicable to other software fields. This study considered the programming and check-out phases of the software development cycle but did not enter into the earlier phases of problem definition and development of the programming specification. The present discussion follows along the same lines.

* Portions of the material presented here were developed under Contract F04695-67-C-0165 with the Air Force Space and Missile Systems Organization.

Quality was considered in terms of the components which go into its makeup: the quality attributes. Definitive statements of the quality attributes were formulated in the first phase of the study. Each attribute is a precise statement of a specific software characteristic. The attribute statements in themselves constitute a definition of quality for software. For a program to be of high quality, it must possess substantially all of the applicable quality attributes. During the second phase, a metric was developed for quantitative measurement of each quality attribute. These metrics, which are stated as mathematical formulas relating measurable characteristics to the determination of program quality, can be used to produce a numerical value that makes it possible to compare a given program with other programs or a desired standard.

This discussion concentrates on presentation and discussion of the attributes. All of the attributes developed in the study are listed; although some reflect the study's orientation to spaceborne software, the majority are general in nature. Only a few of the metrics are given to exemplify their relationship to the attributes. Many of them require a detailed analysis of the program being evaluated, which would be very difficult for any but the simplest programs. One way to circumvent this difficulty would be to develop computer programs to mechanize the analysis. Preference for this approach during the study made it necessary to define the metrics rigorously; hence the complete set is far too bulky to present here. Another way to circumvent detailed analysis of an entire program would be to extrapolate the results obtained from analyzing only sample sections.

Being able to define the quality attributes and express them in metrics does not complete the story, since many external factors influence a program's performance with respect both to individual attributes and to the overall quality determination. These external factors and their influence were considered in the third phase of the study, and an overall quality model was then constructed using the metrics together with weighting

coefficients and normalization factors derived from the external factors evaluation. This quality model, besides being used as a tool for the measurement or comparison of program quality, can be useful in evaluating the importance of various aspects of the programming environment and directing the development of the program. Following brief discussion of external factors and the quality model, some of the potential applications of the quality model are suggested in the final section of this presentation.

QUALITY ATTRIBUTES AND METRICS

The attributes of software are visualized as forming a pyramid whose apex is the attribute that the program be of high merit and whose base is a myriad of minor computer-, system-, and application-dependent attributes. The attributes, A_i, defined here lie in the middle range, being general enough to apply to a wide range of programs yet specific enough to permit practical program evaluation. They are classed in the following seven groups, each group corresponding to what might be considered a major requirement imposed upon a program; those in any group are conditions that generally should be met to assure that the corresponding major group attribute is satisfied.

A_1 - Mathematical calculations are corrected performed.
A_2 - The program is logically correct.
A_3 - There is no interference between program entities.
A_4 - Computation time and memory usage are optimized.
A_5 - The program is intelligible.
A_6 - The program is easy to modify.
A_7 - The program is easy to learn and use.

Not all of the attributes within a group may be applicable to a particular program and not all have equal importance; some reflect minor program details and others relate to whether a program is useful at all. In some cases an attribute could be properly considered as belonging in several groups; and indeed the interrelationships among attributes has been found to be extremely complex.

Each metric, M_i, is designed so that a program of highest merit will achieve a score of 100 for the respective attribute, and a program of lowest merit a score of 0. The aim has been to make the metrics as objective as possible. In the case of those attributes for which subjective evaluation is necessary, particularly those in the last three groups, a degree of objectivity is preserved by so formulating the metrics that the attributes are separated into components that are evaluated subjectively. These subjective decisions are made by having the evaluator assign a rating from a range of permissible values. The ratings thus obtained for separate aspects of the same attribute are then summed and combined according to a formula to obtain a single numerical value.

The list of quality attributes follows, segregated according to the seven major groups. Each list is accompanied by brief discussions relating the particular attributes to the software characteristics they define and outlining some of the concepts used in formulating their respective metrics.

A_1 - **Mathematical calculations are correctly performed.**
$A_{1.1}$ - Fixed-point variables and constants are scaled to allow storage at the required accuracy.
$A_{1.2}$ - Fixed-point variables and constants are scaled to allow storage for the allowable range of values.
$A_{1.3}$ - Intermediate scalings and scaling readjustments are minimized for all calculations.
$A_{1.4}$ - Arithmetic calculations, including data-base conversions, maintain the maximum accuracy possible within the given number of bits of storage allocated for each variable and overall admissible combinations of possible input values.
$A_{1.5}$ - Arithmetic calculations maintain the requisite accuracy overall admissible combinations of possible input values.
$A_{1.6}$ - Calculations are capable of processing data without overlow over all admissible combinations of input.
$A_{1.7}$ - Constants are biased upon conversion to compensate for subsequent numerical truncation or hardware arithmetic peculiarities.

One of the prime functions of computer programs is to perform numerical calculations. The results of these calculations may be the output of the program; they may be used as input quantities by still other calculations performed by the program; or they may be used by the program to make decisions regarding other functions to be performed. In all three cases, the calculations must be correctly performed and their results must have the requisite accuracy.

Since the attributes in this group all relate to the maintenance of accuracy in calculations and the prevention of overflow, their metrics tend to be measures of the accuracy lost in calculations. To evaluate the significance of accuracy degradation due to word size and instruction functioning, it is necessary to know the overall software system accuracy requirements. If the accuracy requirement is not known for each constant and for each variable, it is usually possible to establish an overall accuracy requirement, for example requiring that each item be calculated to five significant figures or that calculations maintain the accuracy achievable with six-place tables.

Multiple-precision arithmetic can be used where sufficient precision cannot be maintained within the normal word size of the computer. Although this will result

in higher scores for several attributes in this group, lower scores will then result for several attributes in other groups, particularly group A_4, which is concerned with optimizing computation time and memory utilization.

As an example of the metrics in this group, that for $A_{1.1}$ is given. The evaluation of this attribute involves comparing the actual storage allocation with that needed to satisfy precision requirements. The metric is such that a program's measured quality diminishes each time not enough bits have been allocated, but no credit is given for allocation of more than the required number of bits. The metric is as follows:

$$M_{1.1} = 100(m+n)/\left[\sum_{i=1}^{m} \varphi(SC_i,SC_i') + \sum_{i=1}^{n} \varphi(SV_i,SV_i')\right]$$

where

$$\varphi(a,b) = \begin{cases} 1 & \text{if } a \le b \\ a-b+1 & \text{otherwise} \end{cases}$$

C_i = each of the fixed-point constants in the program, $i = 1,2, \ldots, m$

V_i = each of the fixed-point variables in the program, $i = 1,2, \ldots, n$

r = radix of the data representation, e.g., 2 for binary machines, 16 for hexadecimal machines

SC_i = scaling index of C_i, i.e., that power of r such that $C_i \times r^{SC_i}$ gives the true value of the constant represented by C_i

SC_i' = scaling index of C_i that would give the required precision

SV_i = scaling index of V_i

SV_i' = scaling index of V_i that would give the required precision

The scaling notation has been chosen such that numbers with any integer part have a positive scaling index and those with only a fractional part have a zero or negative scaling index.

A_2 - **The program is logically correct.**

$A_{2.1}$ - There are no open branches.

$A_{2.2}$ - Branches point to the correct place in the program.

$A_{2.3}$ - Branches do not initiate an unending loop.

$A_{2.4}$ - Equality comparisons between floating-point operands are avoided.

$A_{2.5}$ - Limit checks are provided on index tables.

$A_{2.6}$ - Program entities are capable of performing their required functions in less than the maximum and more than the minimum time allowed.

$A_{2.7}$ - Input and computed variables are time-coherent.

$A_{2.8}$ - Validity checks are made for input data.

$A_{2.9}$ - Diagnostic outputs are implemented for both recoverable and catastrophic errors.

$A_{2.10}$ - Recovery procedures are implemented for momentary, correctable errors.

$A_{2.11}$ - The program initializes all functional elements such that no assumptions are made about their existing states.

$A_{2.12}$ - Reference to illegal or unfilled locations will cause a branch to an alarm or error recovery routine.

In addition to performing all mathematical calculations correctly, a computer program must be constructed so as to insure that it performs all of its functions in the proper sequence, at the proper time, with appropriate constants and variables, etc. This general characteristic is classified as logical correctness. As an example, the metric for attribute $A_{2.6}$ is:

$$M_{2.6} = 100\left[\sum_{i=1}^{n} (A_i + B_i)\right]/2n$$

where

$A_i = \begin{cases} 1 \text{ if the maximum execution time for the } i^{th} \text{ program entity is less than its allowable maximum} \\ 0 \text{ otherwise} \end{cases}$

$B_i = \begin{cases} 1 \text{ if the minimum execution time for the program entity is greater than its allowable minimum} \\ 0 \text{ otherwise} \end{cases}$

n = number of program entities

A_3 - **There is no intereference between program entities.**

$A_{3.1}$ - Entities that may be referenced at the same time do not physically overlap.

$A_{3.2}$ - Program entities change only those other entities which have been designed to be changed or which act as communication media.

$A_{3.3}$ - All program entities accept input and transmit output at proper rates.

$A_{3.4}$ - Subroutines are capable of being reentrant where usage requires.

$A_{3.5}$ - Subroutines preserve and restore all common locations and registers they use.

The attributes in this group are meaningful for programs having a structure which is correlated to such things as function, memory overlay, or input/output requirements. To measure quality with respect to these attributes, this structure and the individual program entities of which it is composed must be discernible by those performing the analysis. To some extent these entities can be identified by the coding that exists; for example, a subroutine can be identified by the fact that it is executed as the result of some return jump instruction and exits to the locality of that return jump. However, for higher levels in the structure, information additional to the computer code is required.

A_4 - **Computation time and memory usage are optimized.**

$A_{4.1}$ - Several data items are packed into a single word.

$A_{4.2}$ - Constants and variables are so located in memory as to allow indexing operations for acquiring, using, and storing them.

$A_{4.3}$ - Redundant subroutines are used to optimize time utilization.

$A_{4.4}$ - The conditional branch coding selected from the available set uses the smallest possible amount of memory.

$A_{4.5}$ - The conditional branch coding selected from the available set executes in the shortest possible time for the longest possible path.

$A_{4.6}$ - The conditional branch coding selected from the available set requires the shortest possible execution time for the most likely path.

$A_{4.7}$ - Routine usage of coding techniques that carry burdensome overheads is avoided.

$A_{4.8}$ - Items stored in logical arrays have uniform scaling.

$A_{4.9}$ - The redundant portions of constants and variables stored in arrays and of stored character strings are eliminated.

$A_{4.10}$ - Unnecessary storing of intermediate results is avoided.

$A_{4.11}$ - Frequently exercised sequences of code are programmed using subroutines.

$A_{4.12}$ - The manipulation of those registers required for accessing different memory segments is minimized.

The metrics for the attributes in this group are directly related to the size and execution time of program elements. They reflect the fact that programming techniques for reducing the amount of memory required tend to increase execution time and vice versa. Thus it is clear that it will be difficult or impossible for a program to achieve high scores both on metrics concerned with program size and on those concerned with execution time. The complexity of these interrelationships can be described by the following example: Subroutines are used to save space at a slight increase in time, and to a large extent in situations in which program size is critical. When time is critical, however, a single general subroutine's timing penalty may be unacceptable. Several specialized subroutines could be used to reduce overall execution time, but this would result in an increase in the memory space used. In analyzing whether a program maintains an efficient balance between time and space constraints, the amount of and reason for subroutines must be considered.

It is important to establish early in the software development cycle the allowable memory space and execution time for various program elements. These budgets then serve as guidelines in evaluation of program performance. Even if budgets are not specifically established, an absolute limitation on program size is imposed by the memory capacity of the computer, and desired program response times often impose a timing requirement. A program of high quality not only meets these budgets, whether established or not, but also uses a minimum of space and time, thereby permitting modifications and expansions to be made to the original program.

A_5 - **The program is intelligible.**

$A_{5.1}$ - Consistent coding techniques are set up and followed.

$A_{5.2}$ - Frequent comments are inserted to clarify the code.

$A_{5.3}$ - Instructions are not modified during program execution. If they are modified, however, such modifications are clearly identified.

$A_{5.4}$ - Indirect methods of referencing quantities are clearly identified.

$A_{5.5}$ - The real-time constraints of a program are clearly identified.

$A_{5.6}$ - The program flow is easy to follow.

$A_{5.7}$ - Symbolic names and labels are clear and meaningful.

These attributes relate to the ease in which a program can be analyzed and do not, like the first four groups, affect program correctness. A program may function perfectly while operating and still be deficient with regard to these attributes. Such deficiencies increase the cost of program development and debugging and make acceptance testing difficult or impossible. These attributes also influence the ease in which a program can be modified (which is covered more fully in group A_6), and the ease in which a program can be learned and used (which is covered in group A_7).

The metric for attribute $A_{5.3}$ exemplifies this group.

$$M_{5.3} = max\ [0,100\ (I - 2N + C)/I]$$

where

I = number of instructions in the program

N = number of instructions modified during program execution

C = number of modified instructions that have comments which indicate the source of modification, the type, and the result that can occur during execution

A_6 - **The program is easy to modify.**

$A_{6.1}$ - The program structure is correlated to functional demands.

$A_{6.2}$ - The program logic is as simple as possible.

$A_{6.3}$ - Multiple storage assignments for constants or variables are minimized.

$A_{6.4}$ - Areas that require frequent changes are capable of being changed by input option.

$A_{6.5}$ - Data words are organized so as to be easily modified.

$A_{6.6}$ - Program units are standardized so as to be interchangeable.

Changes and refinements may require modification of a program to accomplish the original objectives. Further, many computations are common to various types of applications; if a program is easily modified it may be adaptable to many tasks at considerable savings in cost and time.

The metrics in this group indicate whether a program is amenable to change; they do not give a breakdown of the expense expected when modifications are made. For example, consider the following metric for $A_{6.2}$:

$$M_{6.2} = \frac{100}{n} \sum_{i=1}^{n} \frac{F_i}{R}$$

where

n = number of instructions in the program

F_i = number of programmer-accessible registers free after the i^{th} instruction, i = 1, 2, . . ., n

R = total number of programmer-accessible registers

It will be nearly impossible to score 100 on this metric; thus the resulting measurement is only a relative indication of how tight the coding is. For example, at least one of the 20 registers on the IBM 360 contains a base address at all times, so that the maximum any 360 program could score on this metric would be 95.

A_7 - **The program is easy to learn and use.**

$A_{7.1}$ - External communication is readily analyzed and acknowledged.

$A_{7.2}$ - The program documentation is meaningful, clear, concise, and readable, and provides a ready reference for learning, operating, and debugging the program.

$A_{7.3}$ - Intermediate instruction listings corresponding to higher level statements are adequate.

$A_{7.4}$ - Program images and supporting data or materials are explicitly and consistently identified.

$A_{7.5}$ - All errors are clearly communicated to the user in a meaningful manner and are indicative of the proper user response.

$A_{7.6}$ - Sufficient time is allowed for the user to comprehend and respond to messages.

$A_{7.7}$ - Minimal effort is needed for the user to effect a response.

$A_{7.8}$ - The program input is simple, intelligible, and easy to modify.

It is often difficult to separate the learning and using functions. In some on-line applications the user might not require an intimate knowledge of the program but would be concerned about the ease of using it. In other applications the user must have an accurate and detailed knowledge about how the program works. Ease of learning and use also minimizes the effects of personnel turnover and maximizes the benefits obtainable by using the program over a long time period in several applications.

The metric for $A_{7.2}$ illustrates how a quantitative value is assigned to qualitative judgments. The evaluator rates the program with respect to the applicable questions from the following list on a scale from 0 (low) to 10 (high).

1) Are the flow charts adequately descriptive without being slavish copies of the coding?
2) Are several levels of flow charts provided if necessary for clarity?
3) Can the flow charts be quickly related to the corresponding coding and vice versa?
4) Are potentionally marginal situations (such as overflow, critical timing, etc.) identified by the documentation?
5) Are user instructions sufficient without reference to detailed program documentation?
6) Can the documentation be easily used in training classes and design reviews?
7) Are adequate cross-references provided between program error outputs, pertinent documentation, and corresponding coding?
8) Does the documentation reflect the actual program?
9) Is the documentation organized so that it can be easily updated?
10) Are appropriate references made to ancillary documentation such as programming specifications and computer manuals?

The metric is:

$$M_{7.2} = \frac{10}{n} \sum_{i=1}^{n} Q_i$$

where

n = number of applicable questions

Q_i = rating for the i^{th} question

EXTERNAL FACTORS

The attainment of perfect scores on the metrics will often be impossible because of influences and constraints imposed by the environment within which a program is developed. For a certain environment there will be a maximum and a minimum score possible for each of the attributes; these maxima and minima are determined by subjectively evaluating the influence of all applicable external factors on the attributes. The external factors relate to the computer hardware, the programming specification, the schedules, the state of the art, and so forth.

The external factor concept permits the software developer to be judged in the context of how well he did under existing conditions. Only those external factors which cannot be controlled by the developer are considered; it is up to him to attain the highest level of quality possible under existing conditions. For example, consider the case in which low scores are made for $A_{5.7}$ (symbolic names and labels are clear and meaningful) because the imposed programming language did not permit meaningful statement identification. The external factor, the required programming language, therefore has caused a degradation in measured quality. Given this external factor, no organization or technique could overcome this handicap. However, if the choice of programming language had been left to the program developer, this same external factor would not apply; he could have scored higher on this attribute by using a more suitable language.

Consider as another example $A_{4.7}$ (routine usage of coding techniques that carry burdensome overheads is avoided). If the program has to be developed in a very short period of time, it would probably be unavoidable that off-the-shelf subroutines be used instead of subroutines tailored to the particular application. The external factor, imposed schedules, would therefore limit the achievable quality with regard to this attribute and should be accounted for in the quality evaluation. However, if there were adequate time for program development but performance with respect to $A_{4.7}$ was poor because inexperienced programmers were employed, no external factor would apply and no adjustment should be made in the final quality evaluation.

It is desirable that the significance and applicability of the external factors be established as early as possible. Both of the foregoing have been examples of such factors. In some cases, however, the influence of external factors will become apparent only as the program development progresses. If the programming specification is modified late in the development cycle, low scores are certain to result for many attributes.

THE QUALITY MODEL

The quality model provides a means of relating all factors necessary to judge a program's quality: the **absolute** measure of the degree to which it possesses the applicable attributes; the **normalized** measure indicating how good the program is in view of the external factors prevailing during its development; and the **weighted** measure accounting for the relative importance of the attributes. A vector notation is used to denote these three types of measures for the quality of the i^{th} attribute, as follows:

$$Q_i = \begin{pmatrix} M_i \\ M_i' \\ M_i'' \end{pmatrix}$$

where

M_i = absolute measure-of-quality for A_i
M_i' = normalized measure-of-quality for A_i
M_i'' = weighted measure-of-quality for A_i

The absolute measure-of-quality for A_i is the computed value obtained by the use of the metric. The normalized measure is given by

$$M_i = 100\left(\frac{M_i - M_{i_{min}}}{M_{i_{max}} - M_{i_{min}}}\right)$$

where $M_{i_{min}}$ and $M_{i_{max}}$ define the practical range which could be obtained for a given set of external factors. The normalized measure allows the program developer to be rated from 0 to 100 depending on his relative position within this range. The $M_{i_{min}}$ and $M_{i_{max}}$ terms are determined by considering for each A_i all applicable external factors and making a judgment as to their influence on the minimum and maximum scores possible. These minimum and maximum values must satisfy the relationships

$$0 \leq M_{i_{min}} \leq M_{i_{max}} \leq 100$$

$$M_{i_{min}} \leq M_i \leq M_{i_{max}}$$

The weighted measure is obtained from

$$M_i'' = \frac{k_i M_i'}{100}$$

where k_i are attribute weights assigned by the user, ranging in value from 0 (no importance) to 100 (maximum importance), for the specific program.

The concept of attribute weights provides the user with an opportunity to specify the relative importance of the attributes to him; in other words, the desired character of the software to be produced. These weights permit the user to convey attribute importance from the

very first, when the program development task is defined or solicited. He can assign weights to the individual attributes on an independent basis, especially for those attributes he considers particularly vital, or he can specify the same weights for all attributes within a group. As an example, the user would be likely to attach highest importance to the attributes within groups A_1, A_2, and A_3 for software to be developed for a one-time operational application; medium importance for group A_4; and low importance for the more subjectively-oriented attributes in groups A_5, A_6, and A_7. On the other hand, if a prototype program was to be developed for initial laboratory usage and there was every intention to use it as a stepping-stone for future operational program development, then the relative importance of the attribute groups would very likely be reversed to improve its flexibility, intelligibility, and longevity.

To gain an overall figure of quality for an entire program, the values achieved for each of the n attributes are combined as follows:

$$Q = \begin{pmatrix} Q' \\ Q'' \\ Q''' \end{pmatrix}$$

where

$$Q' = \sum_{i=1}^{n} \frac{M_i}{n}$$

$$Q'' = \sum_{i=1}^{n} \frac{100}{n} \left(\frac{M_i - M_{i_{min}}}{M_{i_{max}} - M_{i_{min}}} \right)$$

$$Q''' = 100 \sum_{i=1}^{n} M_i'' / \sum_{i=1}^{n} k_i$$

The quality model does not include any coupling effects between attributes, such as would be significant for those in group A_4. However, such effects can be accounted for in the selection of $M_{i_{max}}$ and $M_{i_{min}}$.

The quality model is clearly oriented toward application from the initial stages of program procurement, and it should provide a very positive influence on the quality of the end-item software. The primary dividends are expected to accrue from its use to express the objectives to be achieved by the developer, and their relative importance; the user would be able to specify the desired quality and the developer would have a goal to work toward. The statement of these objectives would almost certainly cause the developer to slant his programming effort in such a way as to achieve higher scores—certainly a positive result assuming the objectives had been correctly determined in the first place. The effect of the environment expected to exist during the development would also be predicted and the terms and conditions relative to an adverse environment clearly indicated. The user would be likely to make an effort to improve an adverse environment because many of the resultant penalties to him would be known beforehand. Thus the model can be used as a management tool by both the user and the developer to direct the effort toward the defined objectives.

The quality model is also expected to be of value in pinpointing troublesome areas after program completion. Although a program may have passed all acceptance tests, the receipt of consistently low scores in an area such as ease of modification might portend future troubles that could be obviated by timely remedial action. More immediately, low scores would indicate that the developer had not completely performed his tasks and lead to his being required to make in-scope modifications to improve the program.

Use of quality model might increase software development cost, since the tasks of performing the necessary tests and obtaining the data would probably be added to the cost of the programming effort. This is typical of quality assurance procedures, however; a trade-off exists between making expenditures during development to assure a high-quality product and preventing unnecessary future expenditures. Any excess cost seems a small price to pay for establishing a means of procuring a product in which both developer and user have confidence.