Evaluating Measures of Program Quality

K. A. REDISH AND W. F. SMYTH
Department of Computer Science and Systems, McMaster University, Hamilton, Ontario, Canada L8S 4K1

A number of approaches to the measurement of program quality or 'style' have recently been described in the computing literature. This article discusses criteria which may be considered for the evaluation of these and other approaches, and for the development of new ones.

Received February 1986

1. INTRODUCTION

Over the past few years an increasing number of articles have discussed the measurement of the quality (perhaps especially the complexity) of computer programs or, more generally, computer software. Much of this work relates either to Knuth's 1971 paper[1] on the syntactical characteristics of FORTRAN programs or to Halstead's 1977 proposals for complexity measures.[2] Recently two reports have appeared that in quite different ways provide outlines of a framework within which quality or complexity measures could be evaluated: Weyuker[3] proposes seven formal 'properties' which complexity measures should possess, while in a wider context Höcker *et al.*[4] provide structured descriptions of 50 software quality measures whose domains of application range from 'correctness' to 'maintainability' and 'portability'. A related study by Berghel and Sellach[5] compares two different measures of program 'similarity'.

Program complexity can of course be conceived as a reflection of the 'complexity' of the underlying problem, and so as a topic quite separate from program quality. In this paper, however, we treat the complexity of a computer program as just one of the aspects of its 'quality': we propose criteria, for the most part less formal than those of Weyuker and somewhat narrower that those implied by Höcker, which could be used to evaluate computer systems designed to measure program quality. Since it seems likely that such systems will become more common in both academic and software engineering environments, our primary objective is to stimulate discussion of some important ideas and assumptions which arise naturally during their design and development.

As indicated above, one main line of development for program quality measures goes back at least to Knuth's early attempt to characterize FORTRAN programs in terms of the frequency with which various language constructs (statement types) are used.[1] More recently, the work of Rees[6] has stimulated a number of researchers to design measures and develop corresponding software related to programming 'style' or quality.[7–10] In order to lay the groundwork for later discussion, we describe here briefly the latest of these software systems, called AUTOMARK; further details can be found in Refs 10 and 11.

Like Rees' system, AUTOMARK is designed to mark student programs and based on a given 'model' program P. An AUTOMARK mark is in fact based on both a 'dynamic' analysis (correctness of program output) and a 'static' analysis (program quality or style), but except for criterion **C4** (below) we concern ourselves here primarily with the latter. As discussed in Ref. 10, the measures of program style to be used for static analysis are selected by the instructor from a set of about 250 'factors' generated as a byproduct of the syntax-checking phase of program compilation. Suppose that n factors have been chosen and numbered $1, \ldots, n$. Then AUTOMARK marking is based on the following six vectors of length n:

F: the non-negative values of the factors computed for model program P; for example, if the first factor is *number of* **goto**s, and P contains three **goto**s, then $F[1] = 3$.

T: non-negative relative tolerances for the factor values F; continuing the above example, if $T[1] = 0.34$, then another program Q whose count of **goto**s fell in the range

$$[F[1]-T[1]*F[1],\ F[1]+T[1]*F[1]] = [2, 4]$$

would receive full marks for factor 1.

X: the maximum marks (> 1) available for each of the factors (P receives a mark of 1 for each factor; the *minimum* possible mark for each factor is zero).

W: the non-negative weights assigned to the marks for each factor; the mark achieved by P is then

$$\sum_{i=1}^{n} W[i].$$

L, G: indicators taking one of the three values $-1, 0, +1$ according as a factor value less (greater) than the tolerance range for the model program is regarded as

- *worse* (-1);
- *the same* (0);
- *better* $(+1)$.

The student's mark for a particular factor, say the ith one, is determined using these vectors as indicated in Fig. 1*a*; the mark is just the m-intercept on one of the dotted lines determined by the vertical line $f = F[i]$, where $F[i]$ is the value of the ith factor for program Q.

In the example $i = 1$ described above, values of the vector elements might reasonably be

$F[1] = 3$ (P contains 3 **goto**s)
$T[1] = 0.34$
$X[1] = 2$ (a program containing no **goto**s would receive maximum mark 2)
$W[1] = w$ (an appropriate weight reflecting the importance of this measure)

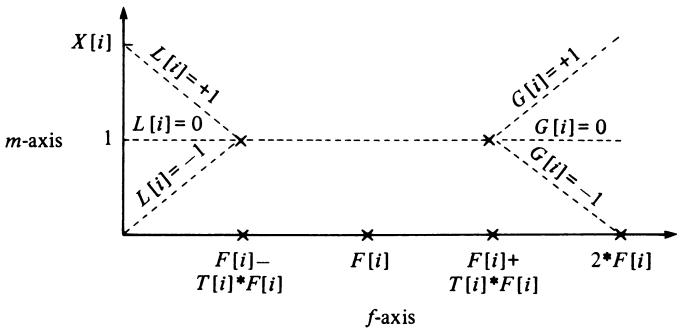

Fig. 1*a*. Calculation of student mark for factor *i* (AUTOMARK).

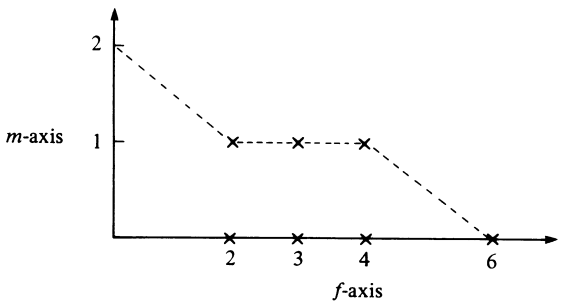

Fig. 1*b*. Calculation of student mark for number of GOTOs (AUTOMARK).

$L[1] = +1$ (fewer **gotos** is 'better')
$G[1] = -1$ (more **gotos** is 'worse')

Then the mark for factor 1 would be determined as shown in Fig. 1*b*.

The overall mark for the student's program Q is then given by the scalar product $m = W.M'$, where M' is the vector of marks achieved by Q for individual factors.

A second main line of development of software evaluation measures goes back at least to McCabe[12] and Halstead,[2] and concerns itself primarily with measures of complexity. Two interesting measures have recently been proposed by Oviedo:[13] a control flow (*CF*) measure and a data flow (*DF*) measure. Both of these measures depend upon breaking down the program into maximal atoms called *blocks*, and then constructing the flow graph of the program treating the blocks as vertices. The *CF* measure is then just the number of vertices in the flow graph (usually slightly adjusted to take account of function/procedure calls). The *DF* measure is however rather more complicated: it depends also upon an analysis of references to, and assignments of, program variables. Roughly speaking, *DF* computes for each block the number of relevant *previous* assignments of those variables which are assigned within the block, and then sums over all blocks; it may be thought of as representing the overall uncertainty about the values of the program variables.

Ignore
Variable naming, logical errors, missing data checks, pretty-printing, indentation, lack of comments.
Consider
(1) the number and type of variables used;
(2) the use of variables;
(3) the number and kind of control structures used;
(4) the use of control structures;
(5) the degree of difficulty in determining the conditions which cause a section of code to be executed;
(6) the degree of difficulty in following sequences of statements in program paths;
(7) program modularity.
Finally
Can the program be written in a simpler way?

Fig. 2. Criteria for grading.[14]

Adopting the view that, in relation to a fixed problem, program quality is the 'inverse' of program complexity, Van Verth[14] uses extensions of Oviedo's measures, together with certain modularity measures, to mark student programs, and reports reasonably good statistical agreement with expert human marking. The main concerns of her marking software are perhaps best conveyed by the criteria for grading provided to the human markers (Fig. 2).

Van Verth's final mark is just the sum

$$m = CF + DF + LP + SP$$

where *LP* is a penalty assessed for having modules which are too large, and *SP* is a penalty for too many small modules. Note that a *smaller* value of *m* indicates a *better* program.

In the remainder of this article, making reference to AUTOMARK and the Van Verth system, we discuss general criteria for the evaluation of measures of program quality. In Section 2 we discuss criteria related to the basis for the evaluation, and in Section 3 we discuss other, more specific criteria, and their relation to criteria suggested by Weyuker in a somewhat different context.[3]

2. THE BASIS FOR EVALUATING PROGRAM QUALITY

We propose a first criterion.

C1 The basis upon which programs are to be evaluated should be as much as possible subject to specification by one of the users (usually a course instructor or project manager).

Of course there will be limits on the generality of C1: no software package will encompass all possible ways of evaluating program quality. On the other hand, it should be recognised by the designers of such software that users are likely to disagree on the relative importance of various measures, and that, furthermore, there may be good reason for such disagreement. In the academic environment instructors may, depending on the particular course being given, or the nature of the audience for the course, or the relationship of the course to the overall programme of instruction, place more or less emphasis on one measure or another, or adopt various approaches to the assignment of values to these measures. In a software engineering environment there may be good reasons for preferring measures of clarity to measures of simplicity, or for encouraging small program modules, or for insisting on careful program commenting, and these reasons can arise naturally out of a diversity of situation.

In AUTOMARK, a program's mark is determined by its vector, say *F*, of factor values, together with the six vectors *F*, *T*, *X*, *W*, *L* and *G*, which are all essentially specified by the user. Although apparently at the time of writing Ref. 14 Van Verth's software simply computed

the sum m, there would be no difficulty in introducing the same six vectors into her system by identifying $n = 4$ factors (*CF*, *DF*, *LP*, *SP*), numbering them 1 to 4, and then assigning appropriate values to four elements of each vector.

In this context, it should be noted that there is an important distinction between F and the other vectors: F represents an analysis of a 'model' program, whereas the other vectors are simple assignments of values, based on judgement and experience, by a user. The question arises whether this dependence on a model program is desirable. Van Verth, discussing this point, states (Ref. 14, p. 1.4): 'We acknowledge that the need for relative comparisons is a limitation.' We argue however that it is *not* a limitation, at least if there is a requirement for the apparent precision of a single mark.

C2 Whenever possible, the user should specify a model program which correctly solves the problem.

The model program is the primary mechanism by which the user (instructor or supervisor) expresses his sense of how the problem should be solved. It seems to us that there is no way this information can be made available other than by actually writing the program: in doing so, the user implicitly evaluates a large number of trade-offs and, based presumably on experienced judgement, makes implementation decisions which strike a balance, from his point of view a presumably optimal balance, among various alternative approaches. It is this *balance* which is critical to the evaluation of the quality of the other programs written to solve the same problem: without it, without the model program, there is no objective basis for evaluating trade-offs such as may (or, depending on the problem, may not) exist among certain factors; for example,

number of **goto**s
depth of nesting
number of executable statements
etc.

It follows then almost as a corollary of C2 that the individual factors on which evaluation is based may, and indeed in many cases should, conflict with each other, in the sense that an improvement in one may be associated with deterioration in one or more others. This feature of a software evaluation package will exist, at least in some measure, if the following criterion is satisfied.

C3 The factors (measures) on which evaluation is based should be selected to represent a number of diverse program 'qualities'.

The qualities proposed or considered by three authors are shown in Fig. 3; depending on circumstances, the relative importance of various qualities would vary, and this variation would be reflected in the basis of evaluation selected by the user.

In a software engineering environment, it will unfortunately *not* as a rule be possible to specify a model program. In such cases, measures of program quality will

Author	Qualities
Höcker et al.[4]	analysability, complexity, reliability, modifiability, modularity, system independence, testability, comprehensibility
AUTOMARK[10]	economy, modularity, simplicity, structure, documentation, layout
Van Verth[14]	complexity, modularity

Fig. 3. Program 'qualities' used for evaluation.

therefore be less exact, less specifically related to the computing problem being solved. One way to deal with this difficulty is to prepare model solutions for relatively small sample problems, and use these to evaluate programs written as tests by candidate programmers. This approach effectively simulates the student environment and indirectly uses a *program* evaluation system to evaluate *programmers*.

Another approach has been used with some success as a by-product of AUTOMARK an ASSESS package analyses individual modules of programs (functions or procedures) and displays various measures ranked on a low-to-high (bad-to-good) scale; for example,

comments in the initial block
ratio: comments to statements
simplicity: operand measure

While such measures can provide some indication of overall program quality, especially in directing attention to particular program features, they are not sufficiently problem-sensitive to be meaningful without further detailed investigation and interpretation.

Even when a model program is used, difficult questions of interpretation arise, primarily related to the fact that the model program cannot be expected to dominate every other program with respect to every factor. Suppose factor vectors F_i have been determined for programs P_i, $i = 1, 2, \ldots$. Then denote by

$$M_{ij} = M_i(F_i; F_j, T, X, L, G)$$

the mark vector obtained by P_i when marked using F_j, $j = 1, 2, \ldots$, and T, X, L, G. Corresponding marks would then be represented by the scalar products

$$m_{ij} = W . M_{ij}.$$

At first it might appear to be desirable that the marks should be 'consistent'; for example, that

$$m_{1j} > m_{2j} \leftrightarrow m_{1k} > m_{2k};$$

in other words, that if P_1 achieves a higher mark than P_2 using one model program, it should also achieve a higher mark using another. That this consistency condition is not necessarily satisfied, however, is easily seen from simple examples such as the one shown in Fig. 4.

Suppose
$W = (1, 1)$, $T = (0, 0)$, $X = (2, 2)$, $L = (-1, -1)$, $G = (+1, +1)$; $F_1 = (1, 1)$, $F_2 = (2/3, 3/2)$.
Then
$M_{11} = (1, 1)$, $M_{21} = (2/3, 3/2)$; $M_{12} = (3/2, 2/3)$, $M_{22} = (1, 1)$.
Hence
$m_{11} = 2 < 2.16 = m_{21}.1$; $m_{12} = 2.16 > 2 = m_{22}$.

Fig. 4. 'Inconsistency' of marks with respect to model program.

Indeed, further reflection convinces us that there is no reason why the marks should be consistent in this sense: the basis F_j of marks m_{ij} represents an opinion as to an optimum balance among partially conflicting measures; it is to be expected that the marks m_{ik} with respect to a different opinion (represented by $F_k \neq F_j$), will be ranked quite differently. Only in the case (unrealistic, as we have seen) that (for given T, X, L and G) F_j 'dominates' (is 'better' than) another mark vector F_k in every position can we assert that $m_{jj} > m_{kj}$ and $m_{jk} > m_{kk}$; and even in this case it is *not* necessarily true that for every pair i, i'

$$m_{ij} > m_{i'j} \leftrightarrow m_{ik} > m_{i'k}.$$

In cases where the user is uncertain that the model solution does in fact represent an optimum balance, the possibility of course exists of using multiple model programs and averaging the resulting marks in some way. As a rule, even when the user is dogmatic about his model program, considerable experimentation will be necessary to arrive at 'reasonable' assignments of the vectors T, X, L, G and W.

To conclude this section, we remark that the qualities displayed in Fig. 4 do *not* include correctness and run-time efficiency, both or which are certainly highly relevant to the evaluation of software. In particular, when a model program is used, the correctness criterion can be used to determine whether or not two programs are comparable; that is, whether or not they do in fact solve the same problem. To lend emphasis to this point, we state

C4 The program qualities mentioned in C3 should include correctness and (time and space) efficiency.

Descriptions of two very different approaches to the evaluation of correctness may be found in Refs 11 and 15.

3. FORMAL PROPERTIES OF QUALITY MEASURES

Given any syntactically correct programs A, B and C, we may define a distance function d as follows:

(1) $d(A, A) = 0$;
(2) $d(A, B) \geqslant 0$;
(3) $d(A, B) \leqslant k$, a fixed constant;
(4) $d(A, B) = d(B, A)$;
(5) $d(A, C) \leqslant d(A, B) + d(B, C)$.

If A and B are the same program, we write $A = B$. We introduce also a *null program* Z which has the property that

(6) $d(A, Z) > 0$ and $d(B, Z) > 0 \rightarrow d(A, B) < d(A, Z) + d(B, Z)$.

We call $d(A, Z)$ the *mark* of A and state

C5 A measure d of program quality should satisfy properties (1)–(6).

In terms of the AUTOMARK marking scheme described in Section 1, we may think of the marks vector M_A for a given program A as representing the *coordinates* of A in an n-dimensional Euclidean space whose axes are scaled according to the weight vector W. Then, given a fixed model program P (so that the factor vector F is well defined), and given fixed vectors T, X, L and G, the AUTOMARK distance function is given simply by

$$d(A, B) = W.|(M_A - M_B)|.$$

It is not difficult to verify that, choosing $M_Z = (0, \ldots, 0)$, this function does in fact satisfy C5. Another possible choice, which also satisfies C5, is

$$d(A, B) = \left\{ \sum_{i=1}^{n} W^2[i] (M_A[i] - M_B[i])^2 \right\}^{\frac{1}{2}},$$

the normal Euclidean distance. Observe that it may well not be possible, under the given marking scheme, to specify a program Z whose mark vector is zero; this need not however prevent us from making use of the point $(0, \ldots, 0)$ in our n-dimensional space! See C8.

Weyuker proposes other criteria,[3] some of which can be modified and translated into the present context with interesting results. The first of these essentially requires that the measure used have a certain minimum sensitivity:

C6 For every non-trivial program A, there exists a program A' formed by permutation of the lines (statements) of A, such that $d(A, A') > 0$.

Here we have supposed that a suitable language-dependent definition of 'non-trivial' can be formulated.

We suppose now that a program A is separated into blocks, where a *block* may be thought of roughly as a maximal sequence of program lines whose permutation in any way does not affect *syntactic* correctness; (for more precise and detailed definitions, see Refs 3 and 14). We say A' is *similar* to A (written $A' \sim A$) if A' can be constructed from A by the following operations:

(*a*) permute lines within blocks of A;
(*b*) replace relational operators by other relational operators;
(*c*) replace logical operators by other logical operators;
(*d*) replace constants by other constants of the same type;
(*e*) replace identifiers by other identifiers of the same length.

Observing that the similarity relation is both reflexive and transitive, we state

C7 $A \sim B \rightarrow d(A, B) = 0$.

Measures satisfying C7 will presumably be such as to facilitate detection of 'clone' programs A and B. The operations specified in the definition of similarity are certainly included in the operations normally performed by students (or software pirates) who want to clandestinely transform A into a hopefully unrecognisable clone program B. In order to detect such activity, some searching of program clusters in n-dimensional space will nevertheless still be required, but C7 ensures at least that the measure used is not too sensitive to trivial program changes. In the context of the AUTOMARK scheme, we observe that clone detection may be facilitated by re-marking the programs in each cluster on the basis of a model program selected as 'typical' of the cluster together with a small tolerance vector T. Some problems of clone detection are discussed in Ref. 5.

Weyuker includes a requirement that the number of programs with a given mark should be bounded. For our purposes, however, since by property (3) the mark itself is bounded, this requirement does not seem to be necessary. On the other hand, it is of interest to be able somehow to characterize the *distribution* of program marks in the range $[0, k]$. Presumably collecting statistics about programs actually marked will provide some information about the distribution of *actual* programs over this range; but it is far from clear what this distribution *ought* to be, and how the distribution of actual programs is related to the distribution of all possible programs which solve a particular problem. As a start, we propose

C8 Corresponding to a given problem, there exist fixed values a and b such that, for any correct program A,

$$0 < a \leqslant d(A, Z) \leqslant b < k.$$

In conclusion, we observe that our experience to date with the development and use of program quality measures has been generally positive. One often-

mentioned difficulty with the use of such measures is the assignment of appropriately low marks to programs which are merely 'cosmetic' and do not try to solve the given problem at all. We believe that this difficulty can be dealt with quite effectively by systems which include program correctness in the evaluation of program quality (see C4). To our way of thinking, a much more serious drawback of such quality-measurement systems is that they have no reliable capability to reward and encourage those who in their solutions go beyond the original statement of the problem, or who find a unique and original way to approach it. There seems to be no way round this latter drawback until the day of real (not artificial) machine intelligence finally dawns.

REFERENCES

1. D. E. Knuth, An empirical study of FORTRAN programs. *Softw. Pract. Exper.* **1** (2), 105–133 (1971).
2. M. H. Halstead. *Elements of Software Science*, Elsevier North-Holland (1977).
3. Elaine J. Weyuker, *Evaluating Software Complexity Measures*, Courant Institute of Mathematical Sciences Department of Computer Science Technical Report No. 149 (1985).
4. Hanns Höcker, Wolf D. Itzfeldt, Monika Schmidt and Michael Timm, *Comparative Descriptions of Software Quality Measures*. Gesellschaft für Mathematik und Datenverarbeitung MBH, Bonn (1984).
5. H. L. Berghel and D. L. Sallach, Measurements of program similarity in identical task environments. *SIGPLAN Notices* **19** (8), 65–72 (1984).
6. Michael J. Rees, Automatic assessment aids for Pascal programs. *SIGPLAN Notices* **17** (10), 33–42 (1982).
7. D. P. Hodgson, *Automatic Assessment of Pascal Program Style*. Western Australia Institute of Technology School of Mathematics and Computing Report No. 1 (1983).
8. B. A. E. Meekings, Style analysis of Pascal programs. *SIGPLAN Notices* **18** (9), 45–54 (1983).
9. R. E. Berry and B. A. E. Meekings, A style analysis of C programs. *Comm. ACM* **28** (1), 80–88 (1985).
10. K. A. Redish and W. F. Smyth, Program style analysis: a natural by-product of program compilation. *Comm. ACM* **29** (2), 126–133 (1986).
11. K. A. Redish, W. F. Smyth and P. G. Sutherland, AUTOMARK: an experimental system for marking student programs. *Proc. Ann. Conf. Can. Inf. Proc. Soc.*, 43–46 (1984).
12. T. J. McCabe, A complexity measure. *IEEE Trans. Softw. Eng.* **2** (4), 308–320 (1976).
13. E. I. Oviedo, Control flow, data flow, and program complexity. *Proc. IEEE COMPSAC*, 146–152 (1980).
14. Patricia B. Van Verth, *A System for Automatically Grading Program Quality*, SUNY (Buffalo) Technical Report No. 85–05 (1985).
15. W. Lewis Johnson and Eliot Soloway, PROUST. *Byte* **10** (4), 179–190 (1985).