1978

Measuring Improvements in Program Clarity

Ronald D. Gordon

Report Number:
78-268

Measuring Improvements in Program Clarity

by Ronald D. Gordon

CSD-TR-268

Purdue University
Department of Computer Sciences
West Lafayette, Indiana 47907

April 1978

ABSTRACT

The sharply rising cost incurred during the production of quality software has brought with it the need for the development of new techniques of software measurement. In particular, the ability to objectively assess the clarity of a program is essential in order to rationally develop useful engineering guidelines for efficient software production and language development.

A functional relation between the clarity of a program and the number and frequency of operators and operands which occur in the program is presented. This measure of program clarity provides an estimate of the amount of mental effort required to understand the program, assuming that the reader is fluent in the programming language employed.

This measure is tested by applying it to several published examples which demonstrate improvements in program clarity. The objective assessment which is provided using this measure is found to agree with the experimental data gathered.

Keywords and Phrases: program clarity, software measurement, software complexity, cognitive psychology, software science

CR categories: 4.0, 4.6

Introduction

Several program transformations have been documented which may be applied to programs containing stylistic flaws. These transformations remove the flaw and enhance the program's clarity. A measure of program clarity, in order to be useful, should indicate that the amount of mental effort required for comprehension has been reduced as a result of such a modification. In this report, six transformations are analytically studied and their effect on the program and the measure E_c is presented. In most cases, the improvement in clarity which results is appropriately reflected by a decrease in the amount of effort estimated by this measure.

Early research in the area of software science led to the identification of six impurity classes[1]. These classes characterize specific flaws in programming style. Initially, the presence of an impurity was observed to perturb the invariance of V^* and $\hat{C}V$. In addition, the estimator of program length, $\hat{N}$, would provide results which were no longer an accurate approximation of the observed program length, N. These aberrations enabled researchers to identify and classify the impurities present in several program samples[6].

This initial research was undertaken because at that time the fundamental relations between $\hat{N}$ and N, and V^* and the product $\hat{C}V$ had yet to be substantiated. While many programs did obey such invariant relationships, some programs, most noticeably those written by novices, exhibited values of both V^* and $\hat{N}$ which were not in agreement with the hypotheses developed. A careful analysis of these anomalous cases clearly identified six classes of impurities which contributed to the observed discrepancies. Subsequently, other researchers also verified the relationships governing $\hat{N}$ and V^* for programs without such impurities[4].

At that time it was observed that the impurities which were identified were found only in poorly written programs. Well-written, highly polished programs contained none of them. Yet no consistent pattern was observed which would explain what aspect of the program was improved as the impurities were removed. The estimator $\hat{N}$ did not consistently increase, nor consistently decrease as it approached N. At times, the removal of an impurity would increase $\hat{C}$ while in other situations $\hat{C}$ would decrease. The program volume might also rise or fall as an impurity was removed. The product of $\hat{C}$ and V did not consistently vary, either.

Alone, these observations do not lead to the conclusion that impurities are manifistations of "bad" programming, nor do they explain why professionally prepared code contains none of them. On the other hand, the decrease in the measured value of E_c in response to the removal of impurities for a wide class of typical programs provides a simple explanation for the motivation behind

program purification.

COMPLEMENTARY OPERATORS

The first impurity class concerns the successive application of two complementary operators to the same operand. The use of complementary operators is easily seen to be less desirable in a programming language than the simple deletion of the pair of operators altogether. This is surely true in any language in which such a construct accomplishes nothing, as is the case in all present programming languages. Even in English, a double negative results in a positive, and because such a construction is less clear than a simple statement of the fact, its use is exceptional.

Consider a program as a finite string of operators and operands. At some point, a pair of operators, say α and β occur whose use is complementary. The removal of the two operators leaves the program functionally the same, and conceptually easier to understand. How is the measure E_c affected?

In this analysis, we assume that the operators α and β occur elsewhere in the program, used in a noncomplementary context. Then, simplification will not reduce the number of unique operators present. This is reasonable for suitably large programs, and most importantly, this condition represents the worst case, or minimal amount of improvement, which must be reflected by a measure of program clarity. The net effect of the purification is simply to reduce the total number of occurrences of operators by 2.

For example, the program segment:

$$\cdots X \leftarrow --Y; \cdots$$

would become, after purification:

$$\cdots X \leftarrow Y; \cdots$$

The effort required to understand the original program, considering it exhibited the measurable properties η_1, η_2, N_1, and N_2 is given by the equation $E_c = V/\hat{L}$, which may be expanded in terms of these basic variables so that their effect can be visualized:

$$E_c \text{ impure} = \frac{(N_1+N_2)\eta_1 N_2 \log_2(\eta_1+\eta_2)}{2\,\eta_2}$$

The purified program is easier to understand, and as a result, we would expect a measure of program clarity to yield a result,

E_c, which was less than that of the original. Since N_1 is reduced by 2, we obtain:

$$E_c \text{ pure} = \frac{(N_1+N_2-2)\eta_1 N_2 \log_2(\eta_1+\eta_2)}{2\ \eta_2}$$

Clearly, E_c impure > E_c pure as a result of removing the impurity presented by the presence of the complementary operators. Further, if such purification removes the last occurrences of either or both of the operators, the amount of effort which is measured will be further decreased. The effect here would be to decrease η_1. The measured effort would decrease, since η_1 appears in the numerator of the equation. This is in agreement with the decrease in effort actually observed, when the use of an operator may be eliminated from a program entirely.

In some situations, complementary operators might also involve the use of operands, so that as a result of purification, N_2 is decreased by 2. Such is the case if, for example, the program adds and subtracts a term in an expression:

```
. . . X ← Y + Z - Z; . . .
```

The effects of complementary operators on the clarity of a program are most simply demonstrated in the context of algebraic expressions. Of course, complementary operators may also appear as impurities affecting the control flow of a program. This is a very important area, as much of the recent work in the area of programming style has been concerned with the proper use of control structures within a language. Here too, the proposed measure is most useful in assessing program clarity in view of the presence of such control impurities. The following simple example will serve to illustrate how the presence of this impurity type results in programs which are ultimately more difficult to understand.

```
      IF (.NOT.condition) GO TO 100
      GO TO 200
100 CONTINUE
```

The structure above exhibits an instance of complementary operators which affect the control flow of the program. The operators .NOT. and GO TO 100, may both be removed from the code. This purification will actually decrease N_1 by 3 since an end-of-statement operator is also eliminated. The code which results is:

```
      IF (condition) GO TO 200
100 CONTINUE
```

Most of the time, however, the original code is "optimized" by subsuming the operator .NOT. into the condition tested. For

example, A.NE.B may appear instead of .NOT.A.EQ.B, and the presence of the complementary operators would not be explicit. Reversing the condition and removing the redundant GO TO-operator will reduce N_1 by 2 and perhaps decrease η_1 by 1. This purification enhances the program's clarity, and this is properly reflected by the proposed measure as the amount of mental effort estimated, E_c, decreases.

Several authors have observed that the presence of complementary operators, either in algebraic expressions, or involved with the control flow of a program, degrade program clarity. While they do not present the foregoing conceptualization, they do provide numerous examples which demonstrate the types of purifications outlined above. This includes the material presented by Chmura and Ledgard[2] in support of their clever proverb, "Don't GO TO," and most of the examples presented by Kernighan and Plauger[8] demonstrating the proper formulation of expressions and the correct use of conditional statements. Many of the techniques for avoiding GO TO-statements presented by Knuth and Floyd[11] are simply methods which remove complementary GO TO-operators of the type shown above.

AMBIGUOUS OPERANDS

In some situations it is possible to use the same variable to represent different types of values in different portions of a program. When the use of the variable is suitably disjoint, such a scheme may be carried out without affecting the operation of the program adversely, and with some rudimentary compilers, the technique might even save a few words of storage. Nonetheless, such ambiguous usage makes a program more difficult to comprehend because the meaning of the variable depends on the exact context in which it is used.

To purify the program which contains an ambiguous operand, a unique operand must be introduced. This will increase η_2 by 1, while leaving the other parameters unchanged. For example, the following program segment contains an occurrence of just such an impurity:

```
. . . P + Q → R; R * R → R; . . .
```

Here, the variable R is used to store the sum of two quantities, while later in the code, R represents the square of this sum. In a low level language such a construction may be required, and the explicit use of a temporary will improve the program's clarity:

$$\cdots P + Q \rightarrow SUM;\ SUM * SUM \rightarrow R;\ \cdots$$

The effort required to understand the program after purification is easily calculated. Since only η_2 has been affected, we obtain:

$$E_c \text{ pure} = \frac{(N_1+N_2)\eta_1 N_2 \log_2(\eta_1+\eta_2+1)}{2\ (\eta_2+1)}$$

The value E_c pure which results is strictly less than the value obtained for the original, unpurified version. This must be so since for $\eta_1 \geq 2$ the first derivative of the function $\log_2(\eta_1+\eta_2)/\eta_2$ with respect to η_2 is negative. The conclusion which is reached on the basis of the proposed measure, is that the amount of effort required for comprehension has been decreased by avoiding the use of ambiguous operands. This conclusion is in agreement with the observed improvement in clarity which occurs in actual practice. Interestingly, measures of clarity which use the number of unique variable names as a factor simply contributing to the difficulty of comprehension, do not properly assess this situation.

Hill, Scowen, and Wichmann[7] have recognized the degradation in clarity which results as programmers attempt to reuse program variables. They point out that "programs are easier to understand if each variable has a constant meaning." The proposed measure mirrors this observation.

SYNONYMOUS OPERANDS

Rather than use one name for two different quantities, we may use two names for the same quantity. This represents an instance of synonymous operand usage, the third impurity class.

In order to remove this impurity, the assignment of a value to the second operand is eliminated. This will reduce N_1 by 2 since an occurrence of the assignment and end of statement operators have been eliminated. In addition, N_2 is reduced by 2. Because both occurences of the operand are then eliminated, η_2 is decreased by 1. As an example of this impurity, consider the following simple program segment:

$$\cdots P \rightarrow T1;\ T1 + Q \rightarrow T2;\ T2 * T2 \rightarrow R;\ \cdots$$

After purification, the following code results:

$$\cdots P + Q \rightarrow T2;\ T2 * T2 \rightarrow R;\ \cdots$$

Since both P and T1 contain the same value at all times, the presence of both variables in the program text serves no useful

role, and only confuses the presentation of the algorithm. The programmer was forced to study one more variable and enter its semantic meaning into the vocabulary being developed. The removal of the impurity improves the program's clarity, and this is reflected by a decrease in the estimate provided by the proposed measure.

In general, the presence of this type of impurity takes the form of an extraneous assignment statement:

$$\cdots \text{ expression} \rightarrow \text{variable}; \cdots$$

when the assigned variable and another have the same value at each point in the computation where the synonymous variable is referenced. Such a statement involves N_1' operators and N_2' operands. In the simplest case as in the previous example, $N_1' = N_2' = 2$. After purification, N_1 and N_2 are reduced by N_1' and N_2' respectively and η_2 is reduced by 1 as the synonymous operand is replaced in each occurance with the alternate variable. The resulting estimate of comprehensibility is:

$$E_c \text{ pure} = \frac{(N_1+N_2-N_1'-N_2')\eta_1(N_2-N_2')\log_2(\eta_1+\eta_2-1)}{2\ (\eta_2-1)}$$

In order to study the ratio E_c pure / E_c impure as program vocabulary, η, increases, the following approximations are introduced in order to transform the above equation into a function of η and N_1': $N_1' \approx N_2'$, $N_1 \approx N_2$, and $\eta_1 \approx \eta_2$. The length equation[5] $N \approx \eta_1\log_2\eta_1 + \eta_2\log_2\eta_2$ reduces in this environment to $N \approx \eta\log_2(\eta/2)$. The following ratio is then obtained:

$$\frac{E_c \text{ pure}}{E_c \text{ impure}} = \frac{\eta\log_2(\eta-1)}{(\eta-2)\log_2\eta}\left[1-\frac{N_1'}{(\eta/2)\log_2(\eta/2)}\right]^2$$

It is possible to solve for the critical value of N_1', the point at which E_c pure equals E_c impure as η for the impure version varies. The results are summarized in Table 1.

Table 1: Critical values of N_1' during the removal of synonymous operands

η	N_1' critical	
5	0.55	Because $N_1' \geq 2$ we have
8	0.84	E_c pure < E_c impure for
16	1.28	small modules.
32	1.75	
64	2.22	When $N_1' = 2$ we may find
128	2.70	E_c pure > E_c impure for
256	3.19	large modules.
512	3.68	

The crossover occurs when $\eta = 46$ at which point N_1' critical = 2. For small modules, whose length N is below roughly 208 operators and operands, the removal of the most trivial instance of the synonymous operand impurity is reflected in the decrease of E_c. It is somewhat unfortunate that for larger modules, such a purification may result in an increase in the measured value of E_c when the superflous assignment statement is simply of the form A → B;. Nonetheless, more typical occurances of this impurity involve more operators and operands and as can be seen in Table 1, their removal would be properly assessed by the measure.

COMMON SUBEXPRESSIONS

It is a common practice, whenever a specific combination of terms must be used more than once, to assign a new name to that combination and to utilize that new name in the subsequent occurrence(s) of that term. The primary justification for this procedure in the past had been to save space and speed execution, but with the advent of modern optimizing compilers, the programmer is no longer forced to do this optimization. Nonetheless, the practice is still useful because the resulting code may be more easily understood by the reader. In effect, the programmer modularizes the unwieldly expression much as one might modularize a large program.

Consider a program containing n instances of a subexpression. This program will have the measured properties η_1, η_2, N_1, and N_2. The subexpression itself will contain N_1' operators and N_2' operands. Purification will introduce a unique operand. This new operand will be used in n-1 of the n occurrences of the common subexpression. The purified program will contain the same number of unique operators, and the new operand will increase η_2 by 1. In assigning a value to this new operand, an additional assignment statement will have been introduced. This will increase N_1 by 2: one for the assignment operator, and one for the end-of-statement operator. Since the expression need occur only once, the number of operators is decreased by $(n-1)N_1'$ and the number of operands by $(n-1)N_2'$. Finally, note that the n+1 occurrences of the new operand increases the total number of operands in the purified version of the program. These results are summarized in Table 2.

Table 2: The Effect of Removing a Common Subexpression

Parameter	Original	Purified
Unique Operators	η_1	η_1
Unique Operands	η_2	η_2+1
Total Operators	N_1	$N_1+2-(n-1)N_1'$
Total Operands	N_2	$N_2+n+1-(n-1)N_2'$

The values listed in Table 2 for the purified program version may be used in order to obtain an estimate of the effort which is required for the comprehension of the revised program. For the case when the common subexpression occurs twice in the program (n=2) the expression for the effort expended understanding the purified program is:

$$E_c \text{ pure} = \frac{(N_1+N_2+5-N_1'-N_2')\eta_1(N_2+3-N_2')\log_2(\eta_1+\eta_2+1)}{2\ (\eta_2+1)}$$

This expression indicates that the purified program will be significantly easier to understand after removing a subexpression which occurs only twice, if it consists of two or more operators and three or more operands. This conclusion is easy to justify. Clearly, the factor $\log_2(\eta_1+\eta_2)/\eta_2$ is slightly greater than $\log_2(\eta_1+\eta_2+1)/(\eta_2+1)$ since the denominator is increasing faster than the numerator. The smallest possible values of N_1' and N_2' which would then guarantee that the value E_c pure is less than the measured value E_c impure for the original version must satisfy $N_1'+N_2'\geq 5$ and $N_2'\geq 3$. Selecting $N_2'=3$ yields $N_1'=2$. For example, the program segment

$$\cdots P*P + P*Q + P*Q + Q*Q \rightarrow R; \cdots$$

contains the simple common subexpression P*Q. Here, $N_1'=1$, $N_2'=2$, and n=2. The program segment may be purified by introducing a new operand, T, and the required assignment statement:

$$\cdots P*Q \rightarrow T;\ P*P + T + T + Q*Q \rightarrow R; \cdots$$

The resulting code is, at best, only slightly easier to understand than the original! The overhead involved in introducing the new variable, T, and the added assignment statement, very nearly negates the increase in simplicity which results when the redundant expression is removed. If we were to consider only the fragments of code shown above, the software measure yields values of E_c of 303 and 300 elementary mental discriminations respectively. As can be seen, the value of E_c changes little. This result agrees with the subjective assessment of clarity for just such a modification.

The proposed measure has simply identified that situation in which little improvement can be made, and has resulted in a formulation of an easily applied engineering guideline: The removal of a common subexpression will noticeably improve the clarity of the resulting program, only if the expression contains two or more operators and three or more operands, or occurs more than twice in the original code. This engineering principle has been developed from the proposed measure of clarity and is consistent with the observed improvements in program clarity in actual practice. The hypothesis is robust in this respect, as the implication drawn from the formulation of the equation is valid.

Yohe[12] recommends this type of modification because it improves program clarity. He states: "Assuming that descriptive variable names have been chosen for the subexpressions, and that choices of subexpressions have been made carefully, readability will be enhanced." The measure of program clarity which is proposed, reflects this observation and provides a useful engineering principle which may be used as a guide as one "carefully" chooses suitable subexpressions.

The appearance of a common subexpression, as with other impurities, is not confined to algebraic expressions. Quite frequently, this particular impurity is observed to affect the structure of the program. Many times in this context, however, the programming language is unable to provide the needed facilities required for the removal of the redundant code. A simple example is that of a repeated parameter list. In most all programming languages, two modules, invoked with the same argument lists, must explicitly present that list twice:

```
CALL PERIOD (RADIUS, MASS, BODY[I]);
CALL VELOCITY (RADIUS, MASS, BODY[I]);
```

Here, the complex parameter list must appear with both procedure calls because the language contains no mechanism which may be used to effectively factor out this redundant term. A simpler representation would accrue if a new item could be defined as a parameter list, and used for these two calls. The code might appear as:

```
ORBIT ← (RADIUS, MASS, BODY[I]);
CALL PERIOD ORBIT;
CALL VELOCITY ORBIT;
```

A common subexpression may also appear as executable code, repeated in two or more locations in the program, when that code may not be broken down into separate modules. For example, conditional statements often lead to this situation as the code below demonstrates:

```
IF (condition)
   THEN WORKER[EMPLOYEE NO].SALARY←RATE[I]
   ELSE WORKER[EMPLOYEE NO].SALARY←0;
```

The variable WORKER[EMPLOYEE NO].SALARY must be specified twice, and the operators for subscripting, record item specification, and assignment, along with the operands involved, lead to the formation of quite a complex structure. As suggested by Hill, Scowen, and Wichmann[7], much of this can be removed from the program, if we write:

```
WORKER[EMPLOYEE NO].SALARY ←
   IF(condition) THEN RATE[I] ELSE 0;
```

This type of simplification is not new. Many other authors have advocated its use in improving program clarity, and several languages allow such structures to be freely used. What is important here, is that the proposed measure of clarity handles just such purifications, as simply as it does the simplification of algebraic expressions. The measure of clarity indicates that such a transformation does improve the clarity of the program, and it provides a result which specifies how much easier the code will be to understand.

The proposed hypothesis may simply identify those features of a language which assist in preparing programs which are easy to understand. In COBOL, for example, one may increment the contents of a field quite simply by coding:

```
ADD 1 TO WORKER[EMPLOYEE NO].DAYS WORKED.
```

Many programming languages do not support a facility which allows for such a simple statement for this operation. Inevitably, in such languages, a certain number of common subexpressions must remain, even after the code has been polished. Removing these impurities would improve the understandability of the program. In ALGOL 68, for example, the statement I:=I+1 may be more simply expressed as I:+=1. Dijkstra[3] used the construction I plus 1 to simply express the reflexive use of the variable I. Hoare's notes indicate a similar scheme for the selective updating of sets. For example, to exclude from the set x all members which are also members of y, the notation x:-y is used[12]. Knuth[10] recognizes the need for such a notation and suggests that a reflexive operator be employed. Denoting the required operator by *=, the statement X:=X+Y is coded X*=+Y. When the specification of X is complex, the simplification which results when one uses such schemes noticeably improves the clarity of the program for those fluent with the adopted conventions.

UNWARRANTED ASSIGNMENTS

When an expression is evaluated, given a unique name, and then used only once, we have an occurrence of the fifth type of impurity, unwarranted assignment. This impurity is removed by first deleting the assignment statement from the program. This decreases N_1 by 2 since the assignment operator and the end-of-statement operator are eliminated. The expression itself is subsequently used in place of the operand where it occurs. This will decrease N_2 by 2, and η_2 will be reduced by 1 since we have removed the only two occurrences of the operand. After purification, the value which is obtained as a measure of the program's clarity is given by:

$$E_c \text{ pure} = \frac{(N_1+N_2-4)\eta_1(N_2-2)\log_2(\eta_1+\eta_2-1)}{2\ (\eta_2-1)}$$

This is precisely the same result which is obtained for the removal of synonymous operands when $N_1' = N_2' = 2$. Clearly this is to be expected since the simplest examples of these two impurity classes are indistinguishable. Consider, for example, the program fragment

$$\cdots P \rightarrow T1;\ T1 + Q \rightarrow R;\ \cdots$$

which may be considered to contain either an unwarranted assignment to the variable T1, or the synonymous variables P and T1. More complex examples, however, make the distinction between unwarranted assignment and synonymous operands evident. The following code segment, suggested by Kernighan and Plauger[40] contains unwarranted assignments to the variables F1 and F2:

```
F1=X1-X2*X2
F2=1.0-X2
FX=F1+F2
```

As an example of their proverb "Avoid temporary variables," they state: "The fewer temporary variables there are in a program, the less chance there is that one will not be properly initialized, or that one will be altered unexpectedly before it is used, and the easier the program will be to understand." Their suggestion leads to the following version:

```
FX = X1-X2**2 + 1.0-X2
```

The proposed measure of program clarity reflects the improved comprehensibility of the later version. A value of 120 elementary mental discriminations is obtained for the improved version, down from a value of 288 for the original.

We may investigate the effect of removing an unwarranted assignment from a program by examining the ratio E_c pure / E_c impure as program vocabulary increases. In order to express the

ratio as a function of η, the approximations utilized previously are again employed. Table 3 summarizes the effect on this ratio as η for the impure version varies.

Table 3: The effect of removing an unwarrented assignment as program vocabulary increases

η	E_c pure / E_c impure	
5	0.22	
8	0.70	removal of the impurity
16	0.94	decreases the measure E_c.
32	0.99	
64	1.003	For large modules the
128	1.004	Removal of the impurity
256	1.003	does not effect E_c greatly,
512	1.002	but E_c pure > E_c impure.

Above a vocabulary of 46, purification increases E_c slightly. Although the removal of the impurity does not in this case result in the consistent reduction in E_c, it is perhaps not unreasonable. For very large modules, the use of a meaningful variable to represent the value of a complex expression may assist in program comprehension even though that variable is used only once. In this light, the behavior of the measure appears most satisfactory.

UNFACTORED EXPRESSIONS

It is clear that an expression which could be factored would be easier to understand after factoring. Yet, no general method of removing this impurity is available. This makes the general analysis which we have been following impossible for this particular impurity class. We can, however, examine the behavior of the proposed measure of clarity for an expression which represents a minimal amount of improvement as it is removed, and insure that the proposed measure of effort decreases for this case. Much more profitable situations would lead to more significant reductions under the measure.

The simplest unfactored expression is P*P + 2*P*Q + Q*Q, which becomes (P+Q)**2, once factored. As can be seen, factoring may introduce new operators. In this situation, we have used the grouping operator and exponentiation. In addition, new operands may appear as coefficients or exponents in the factored expression. These new operators and operands may or may not be unique, and will contribute to η_1 and η_2 only occasionally. In

the example given above, we assume that the new operators are indeed unique, and obtain a measure of effort, given by the equation below. No new operands were introduced in this example.

$$E_c \text{ pure} = \frac{(N_1+N_2-7)\eta_1(N_2-4)\log_2(\eta_1+\eta_2+2)}{2\ \eta_2}$$

This expression yields a value for the amount of effort required for comprehension after the impurity above has been removed. It indicates that the resulting code will be easier to understand since E_c pure is less than E_c impure, the value obtained for the original program. This must be so since the factor $(N-7)\log_2(\eta+2)$ is less than $N \log_2\eta$ whenever $N < 2\eta\log_2(\eta+2)$. Because $N \approx \eta\log_2(\eta/2)$ the previous inequality is expected to be valid. Similar results are obtained when more complex expressions are factored, and corresponding decreases in E_c are observed. This is in agreement with the improvement in clarity which is observed whenever this type of impurity is removed from a program.

PROGRAMMING GUIDELINES FOR PROGRAM CLARITY

Several authors have presented guidelines which, when followed, assist in the preparation of programs that are more easily understood. Not all of these guidelines may be applied in a general fashion, and the modifications which are necessary in order to bring the code into compliance with the rules, may not be simply specified. Instead, ad hoc alterations are required, and the programmer's ingenuity is put to the test in applying many of the rules. In some instances, exceptions to such guidelines may be demonstrated, and the resulting code proves to be more difficult to comprehend.

In spite of these serious drawbacks, such guidelines have been drawn up as the best means available for guiding the production of software. Consider, for example, the typical admonishment against the use of the GO TO-statement: Avoid the use of GO TO when a conditional statement, conditional expression, or FOR-statement can fulfill the same purpose[7]. The authors present several examples to demonstrate the application of this rule. They point out that the code

```
      IF K=1
         THEN GO TO L1;
      X := X+C;
      GO TO L2;
L1:   X := X+B;
L2:   continue
```

may be noticeably improved by eliminating the GO TOs and writing instead

```
IF K=1
   THEN X := X+B;
   ELSE X := X+C;
```

Such a guideline is indeed useful in improving the original version of the program. Further enhancements are achieved by writing

```
X := IF K=1
        THEN X+B;
        ELSE X+C;
```

A still better version is presented as

```
X := X + IF K = 1 THEN B ELSE C;
```

The code above may be improved by eliminating the common subexpressions entirely. Following a syntax similar to that developed by the designers of COBOL, we may more clearly state the desired operation by coding

```
ADD (IF K=1 THEN B ELSE C) TO X;
```

The examples presented demonstrate the intended application of such a rule, which bans the use of the GO TO-statement. A careful programmer may learn a great deal, generalizing from such examples. Analyzing the code and obtaining the estimate for the amount of effort required for comprehension, using the measure E_c, verifies that the proposed measure properly reflects the improvements in clarity which have been observed. More properly, we note that the guideline presented, demonstrates a special case of the general theory of program clarity being developed. Other guidelines reported in the literature[7,8,9] are also implied by the proposed measure, although in some instances they are not as precise as the results obtained here:

- Do not calculate the same value more than once.
- Do not insert statements which can never be executed.
- Maintain a constant meaning for each program variable.
- Avoid forming conditional statements which result in a null-THEN clause.
- Eliminate unnecessary intermediate variables.
- Use GO TOs to implement only the basic forms of break and iterate structures.

Many other guidelines may also be included in such a list, some very specific and with limited applicability. Yet, these guidelines do not collectively form a simple, coherent, and robust theory of program clarity. The proposed hypothesis, however, does define just those conditions under which an

improvement in clarity will be ovserved. The interaction of the parameters η_1, η_2, N_1, and N_2, and the role each plays in determining a program's clarity is reflected in the formulation of the equation $E_c = V/\hat{L}$.

CONCLUSION

An estimator of program clarity should reflect an improvement in clarity by indicating a decrease in the amount of mental effort required for comprehension. Six impurity classes had been presented which characterize program flaws shown to impede program understanding. The removal of such impurities can be handled deterministically, and the resulting program structure easily ascertained. Typically, such modifications reduce the estimated amount of mental work provided by the measure, E_c. Alternate hypotheses which have been considered do not exhibit this desired behavior. These failing measures include the number of program steps, the program volume, and the implementation level.

The removal of impurities from a computer program was shown to reduce the amount of mental effort required for comprehension estimated using the measure E_c, for a wide class of typical programs. These impurities had been observed to disturb the invariance between the minimum program volume, V^*, and $\hat{L}V$, the product of the program volume and implementation level. When such impurities are removed, equality is achieved, but neither the program volume nor implementation level are affected in a consistent manner. The impurities cited are not found in well-written programs produced by professionals. This observation alone, however, does not explain what aspect of the program or the programming process is optimized by purification. On the other hand, the measure of program clarity, E_c, generally decreases in response to the removal of such program deficiencies.

BIBLIOGRAPHY

[1] Necdet Bulut and Maurice H. Halstead, "Impurities Found in Algorithm Implementations," ACM SIGPLAN Notices, Volume 9, Number 3, March 1974, pages 9-12.

[2] Louis J. Chmura and Henry F. Ledgard, Cobol With Style: Programming Proverbs, Rochelle Park, New Jersey: Hayden Books, 1976.

[3] Ole-J. Dahl, Edsger W. Dijkstra, and C. A. R. Hoare, Structured Programming, New York, New York: Academic Press, 1972.

[4] James L. Elshoff, "Measuring Commercial PL/I Programs Using Halstead's Criteria," ACM SIGPLAN Notices, Volume 11, Number 5, May 1976, pages 38-46.

[5] Maurice H. Halstead, "Natural Laws Controlling Algorithm Structure?", ACM SIGPLAN Notices, Volume 7, Number 2, February 1972, pages 19-26.

[6] Maurice H. Halstead, "Software Physics: Basic Principles," Research Report RJ 1582, IBM Thomas J. Watson Research Center, Yorktown Heights, New York, May 1975.

[7] I. D. Hill, R. S. Scowen, and B. A. Wichmann, "Writing Algorithms in ALGOL 60," Software-Practice and Experience, Volume 5, Number 3, July-September 1975, pages 223-244.

[8] Brian W. Kernighan and Phillip J. Plauger, The Elements of Programming Style, New York, New York: McGraw-Hill, 1974.

[9] Brian W. Kernighan and Phillip J. Plauger, "Programming Style: Examples and Counterexamples," ACM Computing Surveys, Volume 6, Number 4, December 1974, pages 303-319.

[10] Donald E. Knuth, "A Review of 'Structured Programming'," Technical Report 371, Department of Computer Sciences, Stanford University, Stanford, California, June 1973.

[11] Donald E. Knuth and R. W. Floyd, "Notes on Avoiding GO TO-Statements," Information Processing Letters, Volume 1, Number 1, February 1971, pages 23-31.

[12] James M. Yohe, "An Overview of Programming Practices," ACM Computing Surveys, Volume 6, Number 4, December 1974, pages 221-245.