PROCEEDINGS OF THE SYMPOSIUM ON
LANGUAGE DESIGN AND PROGRAMMING METHODOLOGY
SYDNEY, 10-11 SEPTEMBER, 1979

AN EXPERIMENT IN SOFTWARE SCIENCE

Dan B. Johnston and Andrew M. Lister
Department of Computer Science
University of Queensland

ABSTRACT

This paper describes an experiment which was undertaken for two purposes: firstly to test the applicability of software science in the realm of student programming, and secondly to obtain quantitative inferences about the programming language PASCAL. The results suggest that software science offers little in the area studied, and possible reasons for this are discussed.

1. INTRODUCTION

Software science (Halstead, 1977; Fitzsimmons and Love, 1978; Van der Knijff, 1978) is an embryonic experimental science which attempts to analyse programs in terms of certain basic measures on them. It is concerned with quantifying such program properties as comprehensibility, likelihood of error, and effort to write, and with establishing relationships, or "laws", which allow these quantities to be predicted from simple measures such as counts of operators and operands. The aim of the experiment described in this paper was to investigate the validity of the software science laws in the domain of student programming, and in this context to see what, if anything, software science can tell us about the merits of PASCAL as a programming language.

More precisely, the experiment was designed to investigate the following areas:

(1) The correlation between the measure of "goodness" of a program provided by software science and the subjective assessment of the same program by a practised programmer (in this case a tutor marking the program for assessment). In particular we were interested to learn whether the software science measures could form a reliable basis for automatic grading of programs.

(2) The accuracy of certain approximations in software science.

(3) The language level of PASCAL, in the sense of "high" or "low", as quantified by software science.

A notable feature of the experiment was the large number of sample programs measured: about 13,000 programs written by over 500 students. Indeed the availability of this large sample was a compelling reason for performing the experiment - the opportunity seemed too good to waste! The sample is described in detail in section 3 of the paper after a brief presentation of notation in section 2. Section 4 details the measurements made and the results obtained, and section 5 discusses what conclusions can be drawn.

2. SOFTWARE SCIENCE MEASURES

The fundamental measures of software science, from which all others are derived, are (for any program)

$n1$ - number of distinct operators used
$n2$ - number of distinct operands used
$N1$ - number of operator occurrences
$N2$ - number of operand occurrences

The *vocabulary* of the program is

$$n = n1 + n2$$

and the program *length* is

$$N = N1 + N2$$

The *volume* of a program, which is the minimum number of bits required to hold it, is

$$V = N\log_2 n$$

The *potential volume* V^* of an algorithm is the volume of the minimal program required to express it. Such a minimal program assumes the algorithm to be implemented as a procedure built into the language used, and therefore comprises only a single procedure call. V^* is a property of an algorithm, and is independent of the programming language used.

The ratio

$$L = V^*/V$$

is the *level* of a program, and measures the degree of compaction which would be achieved if the language used allowed the algorithm to be expressed in its minimal form.

The *effort* required to write a program is given by

$$E = V/L$$

This formulation of E is based on the number of mental discriminations required to write the program, and is therefore an indicator of the probable number of errors in the program, the time required to write it, and the effort required to understand it. The measure of "goodness" of a program which expresses a given algorithm in a given language is inversely related to E.

The final measure of software science is the *language level* of a programming language, given by the product

$$\lambda = LV^*$$

which is asserted to be constant over all well-written programs in the language. λ is the quantitative measure which corresponds to intuitive ideas of the level of a programming language.

Calculation of the quantities above for a particular program requires the measurement of *n1*, *n2*, *N1*, and *N2* for the program, together with a knowledge of V^* for the corresponding algorithm. If *N1* or *N2* are not available the estimator

$$\hat{N} = n1 \log_2 n1 + n2 \log_2 n2$$

is claimed to be a good approximation to the length N, and if V^* is unknown then

$$L = (2 \times n2)/(n1 \times N2)$$

can be used as an approximation to the program level L (which can in turn be used to compute approximations $\hat{E}$ and $\hat{\lambda}$ for E and λ). Justification of the estimators $\hat{N}$ and $\hat{L}$ is given by Halstead (1977) on both theoretical and empirical grounds.

3. THE SAMPLE PROGRAMS

The data for the experiment consisted of 12,886 syntactically correct PASCAL programs submitted by over 500 first year students to the University of Queensland's central computer during the first semester of 1979. Syntactically incorrect programs were excluded from the sample since the measurements on these would have been somewhat arbitrary. The sample programs can be divided into two classes; *assignments*, which form part of each student's assessment, and *general programs*, which students run for interest but which are not assessed. The relevant characteristics of each class of programs are described below.

3.1 Assignments

Each student was expected to submit three assignments for assessment. The assignments may be briefly described as follows:

Assignment 1. A program to read the subject codes and examination grades of a hypothetical student, to validate the data, and print out either the credit points obtained for each subject or an indication of a data error.

Assignment 2. A program to simulate the action of a faulty clock over a 12 hour period, printing the actual time at hourly intervals as measured by the clock.

Assignment 3. A program to read a piece of English text, print all its distinct words, and to indicate with an asterisk all the distinct words which have the same initial letter as the final word.

"Ideal" programs for each assignment, written by the lecturer in charge of the course (Lister), are given in the Appendix.

Of course not all students submitted all three assignments, and the sample size was further reduced by eliminating

(1) programs which still contained syntax errors,

(2) programs which failed to meet the stated specifications (for example by simply not working properly),

(3) programs which did more than was asked for (for example by using elaborate output layout).

The programs in categories (2) and (3), which under- or over-achieved the specifications, can be regarded as not expressing the same algorithm as the rest, and were therefore excluded on grounds of comparability. The importance of excluding these programs was not fully realised at the start of the experiment, and they were inadvertently included in the sample for the first assignment. The measurements for this assignment may therefore be slightly less reliable than those for the other two, though since the algorithm was comparatively simple we believe that the number of programs erroneously included was quite small. The final sample sizes, after all eliminations, were 423, 376, and 343 for the three assignments respectively.

A significant characteristic of the assignments was that each program expressed a known algorithm, and thus (in theory at any rate) could be associated with a known value of V^*. This implied that no approximations were necessary in calculating the various measures on the programs, particularly E and λ. It also implied that this group of programs could be used to test the validity of the estimators $\hat{E}$ and $\hat{\lambda}$. Unfortunately the determination of V^* proved to be more difficult than the available literature suggested. The problems which arose, and our solutions to them, are described in more detail in section 4.2.

Each assignment submitted to a tutor for assessment was marked on a scale 0 (very poor) to 4 (very good). This gave us the opportunity to investigate any correlation between the effort E required to write (and understand) a program and the tutor's subjective assessment of the worth of the program as indicated by the mark awarded. Of course any such correlation could be clouded by the role in the marking scheme of factors such as adequacy of comments, which are quite unrelated to program construction. For the third assignment we therefore asked the tutors to give their subjective opinion of each program's clarity on a scale 0 (lowest) to 10 (highest), and investigated whether any correlation existed between this measure and E. Eleven tutors were involved, giving a wide cross-section of experienced opinion, and each tutor assessed between 30 and 80 programs for each assignment.

3.2 General programs

During the semester the students ran a large variety of programs which were not formally assessed. Some of these programs were copied from lecture notes or text books, and some were written as programming exercises. The number of programs which were syntactically correct, and could therefore be included in the sample was 11,744. This number includes preliminary attempts at assignments, since we had no way of distinguishing assignments from other programs except by what was submitted for assessment.

The algorithms expressed by the programs in this sample were unknown and hence the corresponding values of V^* were also unknown. This meant the measures L, E, and λ for this sample could not be computed and had to be replaced by the estimates $\hat{L}$, $\hat{E}$, and $\hat{\lambda}$. However, provided that the accuracy of the estimators was confirmed by measurements on the assignments, the general programs were intended to provide a useful extension of the total sample. The extent to which this intention was fulfilled is discussed in later sections.

4. MEASUREMENTS AND RESULTS

4.1 Counting scheme

The PASCAL compiler used by students was modified to provide the operator and operand counts $n1$, $n2$, $N1$, and $N2$. The precise counting scheme used required some consideration as the available literature gave few examples, none of them for PASCAL. However, by trying to follow the philosophy which appears to have guided earlier workers we were able to adopt a scheme which seemed reasonable. Our resolution of some possibly contentious problems is outlined below.

(1) Only executable text was counted, all declarative text being ignored.

(2) Composite symbols such as *repeat*...*until* and *for*...*to*...*do* were considered as single operators.

(3) Each distinct procedure call was regarded as a separate operator, and commas between parameters were counted as operators only for procedures (such as readln) which are variadic.

(4) *if*...*then* and *if*...*then*...*else* were regarded as separate operators, and *case*...*of* was regarded as a single operator irrespective of the number of case labels used.

(5) The colon in output field width specifications was regarded as an operator, and the field width itself as an operand.

The question of whether a different choice of counting scheme would greatly affect our results is an open one. Work on PL/1 programs (Elshoff, 1978) suggests that some measures, such as V, are insensitive to changes in the counting scheme, while others, such as $\hat{E}$ and $\hat{\lambda}$, are more sensitive. Unfortunately

we did not have the resources to test Elshoff's conclusion by trying various counting schemes in our own experiment.

4.2 Determination of V^* for assignments

The conventional derivation of V^* is to regard the minimal form of an algorithm as a procedure call with two operators (the procedure name and a grouping symbol) and as many operands as there are conceptually distinct parameters. Since each symbol is used only once,

$$V^* = (2 + n2^*) \log_2(2 + n2^*)$$

where $n2^*$ is the number of parameters. Thus the calculation of V^* is straightforward provided the parameters can be readily enumerated.

Unfortunately this was not the case with the algorithms for our three assignments. How many outputs, for example, are there from a simulation, and how many inputs does a piece of English text represent? The accessible literature provides little guidance: all the examples we can find are of programs which transform readily identifiable inputs into readily identifiable results. Furthermore, the inputs and outputs of these programs are unstructured atomic data items, whereas those of our own algorithms seem to need a specification of their structure as part of their definition. It seems important that this structure be taken into account when determining the minimum number of symbols in which each algorithm can be expressed. One way of doing this is to describe the input and output in terms of abstract structuring operations, such as *sequence* and *pair*, as well as the atomic data items themselves. The results of this approach are given below.

Assignment 1

input structure : sequence of pairs (subject, grade)
input symbols : sequence operator, pair operators, subject, grade
output structure: sequence of triplets (credit, error flag 1, error flag 2)
output symbols : sequence operator, triplet operator, credit, flag 1, flag 2

Since the input sequence maps one-one to the output sequence, the sequencing operator need appear only once in a description of the algorithm. Hence $V^* = 10 \log_2 10 = 33.22$

Assignment 2

input structure : single integer (period of simulation)
input symbols : period
output structure: sequence of pairs (hours, minutes)
output symbols : sequence operator, pair operator, hours, minutes

Since no symbol is used more than once, $V^* = 7 \log_2 7 = 19.65$

Assignment 3

input structure : sequence of characters
input symbols : sequence operator, character
output structure: sequence of groups of characters (words)
output symbols : sequence operator, grouping operator, character

Since the two sequence operators are different,

$$V^* = 7 \log_2 7 = 19.65$$

A different approach to deriving V^* is to use the relation $V^*=LV$, and to substitute the estimator $\hat{L}$ for L. This produces an approximation $\hat{V}^*$ whose proximity to V^* is governed by the proximity of $\hat{L}$ to L. Of course it would be foolish to use a value of $\hat{L}$ derived from the sample of student programs, since one of the aims of the experiment was to test the validity of such a value. However, one program outside the sample which could be used is the "ideal" program for the assignment in question. Some justification for this is that the program contains no "impurities", and therefore should produce an estimator $\hat{L}$ which is close to the true value L (Halstead, 1977). Although it is clearly dangerous to argue from a sample of one, we feel that this derivation of an approximation to V^* serves as a useful supplement to the value of V^* obtained earlier. To put it bluntly, two derivations are better than one, particularly when neither is confidently arrived at. The values obtained by both means are shown in Table 1. V^* is the value obtained by analytic derivation, while $\hat{V}^*$ is that obtained from the ideal program. Since the discrepancies are small but significant, both values were used in subsequent calculations.

	V^*	$\hat{V}^*$	$(V^* - \hat{V}^*) / V^*$
Assignment 1	33.22	26.72	0.195
Assignment 2	19.65	22.57	-0.149
Assignment 3	19.65	17.34	0.118

Table 1

4.3 The length estimator $\hat{N}$

The operator and operand counts for each program in the sample were recorded, and from these the values of N and $\hat{N}$ were computed. The validity of $\hat{N}$ as an approximation to N was assessed by computing the mean and variance of the ratio $N/\hat{N}$ over all programs. The results were

$N/\hat{N}$: Mean = 1.075 Variance = 0.072 Sample size = 12,886

The results suggest that $\hat{N}$ is a good estimator for N, even in the domain of student programs. This extends the area of application of the estimator beyond those already established by other workers. However, we do not regard $\hat{N}$ as a particularly important measure, since if it is possible to gather the data to compute $\hat{N}$ then it should also be possible to gather the data to compute N itself.

4.4 The level estimator $\hat{L}$

The level estimator $\hat{L}$ for each program was computed directly from the operator and operand counts. The true level L was computed for those programs (viz. the assignments) for which V^* was known - or at least for which we had a reasonable value. The validity of $\hat{L}$ as an estimator for L was then assessed by computing the mean and variance of the ratio $\hat{L}/L(=E/\hat{E})$ over all assignments. The results, using both values of V^* as derived in section 4.2, are shown in Table 2.

Assignment number	Sample size	V Mean	V Std. Dev.	V^*	$\hat{L}/L$ ($= E/\hat{E}$) Mean	Variance	Std. Dev.
1	423	491	158	33.22	1.04	0.084	0.29
				26.72	1.30	0.130	0.36
2	376	596	423	19.65	1.39	0.093	0.30
				22.57	1.21	0.071	0.27
3	343	963	575	19.65	0.93	0.045	0.21
				17.34	1.05	0.057	0.24

Table 2

Of the three means derived from the analytic computation of V^* two are reasonably close to unity, while the other (Assignment 2) is not. The average of the three means is 1.12. Of the means derived from the estimate of V^* only one (Assignment 3) is close to unity. The average of these means is 1.19. In all cases the standard deviation is about one quarter of the mean. In our view these results indicate that for the programs studied $\hat{L}$ does give a rough estimate of L, but that the estimate is too unreliable to be useful.

4.5 The language level λ

The language level λ ($= LV^*$) was computed for all assignments, using both values of V^* as derived in section 4.2. The results are shown in Table 3.

Assignment number	Sample size	V Mean	V Std. Dev.	V^*	λ Mean,	Variance,	Std. Dev.
1	423	491	158	33.22	2.42	0.386	0.62
				26.72	1.57	0.161	0.40
2	376	596	423	19.65	0.68	0.028	0.17
				22.57	0.90	0.049	0.22
3	343	963	575	19.65	0.41	0.005	0.07
				17.34	0.32	0.004	0.06

Table 3

It is apparent that whichever value of V^* is taken the value of λ declines sharply over the three assignments, and certainly does not display the constant behaviour claimed by Halstead. On reflection we do not find this surprising: indeed what is surprising is the supposition that λ ever could be constant over a range of programs written in the same language.

Since $\lambda = LV^*$,
and $L = V^*/V$,
we have $\lambda = (V^*)^2/V$

Thus for λ to be constant it is necessary for V^* to vary with the square root of V. Now V^* depends on the number of parameters of the algorithm, while V depends on the internal complexity of the algorithm. It seems most unlikely that the complexity of an algorithm is in any mathematical sense related to the number of parameters. Indeed, to take a single example, there is an infinite number of algorithms of widely varying complexity (and hence widely varying V when implemented in a particular language) which can all be expressed in the form $y := f(x)$ and which all therefore have $V^* = 4\log_2 4$.

In our view the only value of λ is as a basis for comparison of *different* programming languages. If the same algorithm is expressed in languages A and B then the ratio λ_A/λ_B can be regarded as a measure of the relative expressive power of the languages. However, this ratio is equal to V_B/V_A, so the value of λ as a measure distinct from V is negligible.

4.6 Correlation between effort measures and marks awarded

Two methods were employed for computing an effort measure for the assignments. The first was to compute the measure E from the relation $E = V^2/V^*$, using both values of V^* obtained in section 4.2. The second was to compute the estimator $\hat{E}$ from

the relation $\hat{E} = V/\hat{L}$. The closeness of these measures for each of the assignments is shown in the last column of Table 2, and has been discussed in section 4.4.

Since the effort measure is claimed to be an inverse measure of the "goodness" of a program, we plotted histograms showing the distribution of both E and $\hat{E}$ against the marks awarded by tutors. Programs with a mark of 0 were omitted since their number was too small to be a valid sample. These histograms are shown in Figures 1-3. (Only the histograms for values of E computed from the analytically derived value of V^* are shown; those for E computed from the estimated value of V^* display a similar shape with a lateral transposition.)

It is clear that the histograms indicate no startling correlation between either E or $\hat{E}$ and the marks awarded. In particular, given a program with a certain value of E (or $\hat{E}$) it would be quite impossible to infer what mark the program had been given. However, there are some general overall patterns: the mean and the variance of both E and $\hat{E}$ tend to decrease as the number of marks awarded increases. The extent to which this is true is illustrated in Figures 4-6, which plot the mean and standard deviation of E and $\hat{E}$ against marks for each of the three assignments. (The kink in the graphs for Assignment 2 (Figure 5) is perhaps explained by the small number (4) of programs awarded a mark of 1.)

One further point is worth noting. The measure $\hat{E}$ we have used here is identical to the measure E_C which Gordon (1979) has suggested is a better measure of program clarity than E. If Gordon's suggestion is valid, and if the marks awarded bear any relation to program clarity (as they should), then one would expect a higher correlation between $\hat{E}$ and the mark than between E and the mark. This expectation is not borne out by the evidence of Figures 1-6. We shall return to this point in the next section, which discusses direct assessment of program clarity.

4.7 Correlation between effort measures and clarity

As mentioned in section 3.1 the tutors marking Assignment 3 were asked to give a subjective assessment of the clarity of each program. Clarity was assessed independently of the mark awarded, the aim being to isolate that quality of a program which might most closely correlate with the effort measures E and $\hat{E}$. The result is indicated in Figures 7 and 8, which show the distribution of E and $\hat{E}$ over clarity. Programs with clarity 0 and 1 were omitted, since their number (3) was too small to be a valid sample.

The observations to be made about these histograms are similar to those made in section 4.6 about the histograms over marks awarded. There is no useful correlation between either E or $\hat{E}$ and the clarity of the programs, and certainly no basis for inferring the clarity of a program from either E or $\hat{E}$.

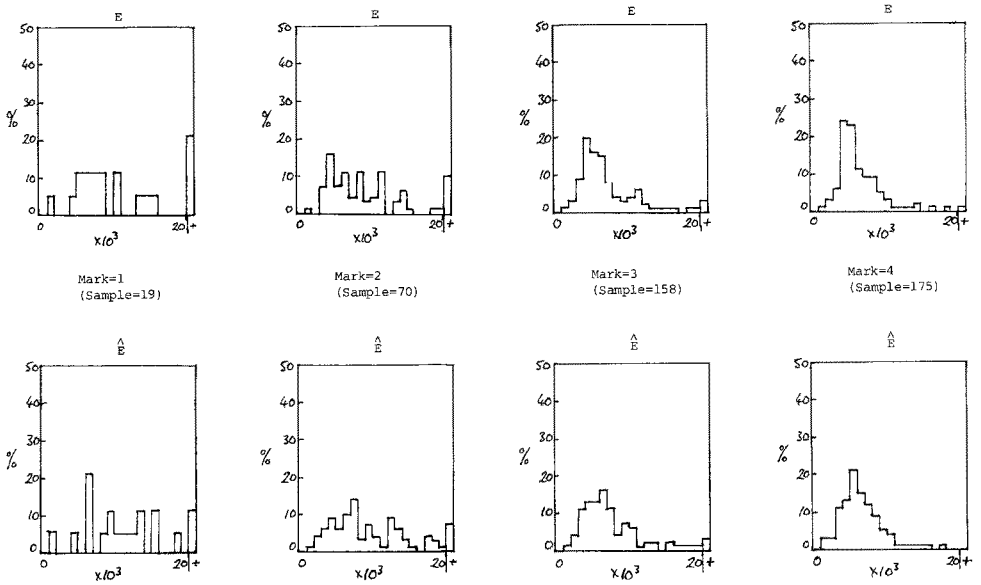

FIGURE 1 DISTRIBUTION OF EFFORT MEASURES OVER MARKS AWARDED (ASSIGNMENT 1)

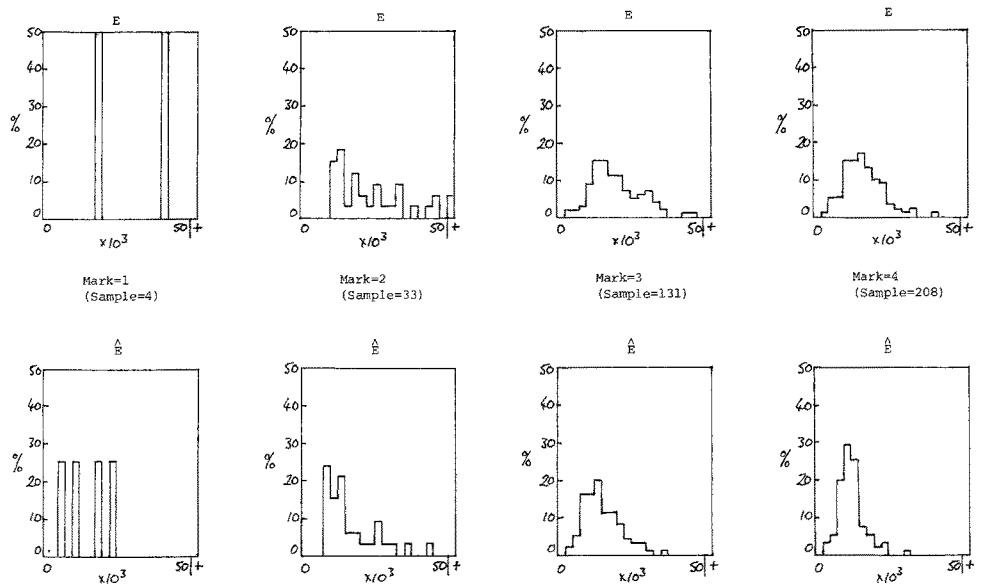

FIGURE 2 DISTRIBUTION OF EFFORT MEASURES OVER MARKS AWARDED (ASSIGNMENT 2)

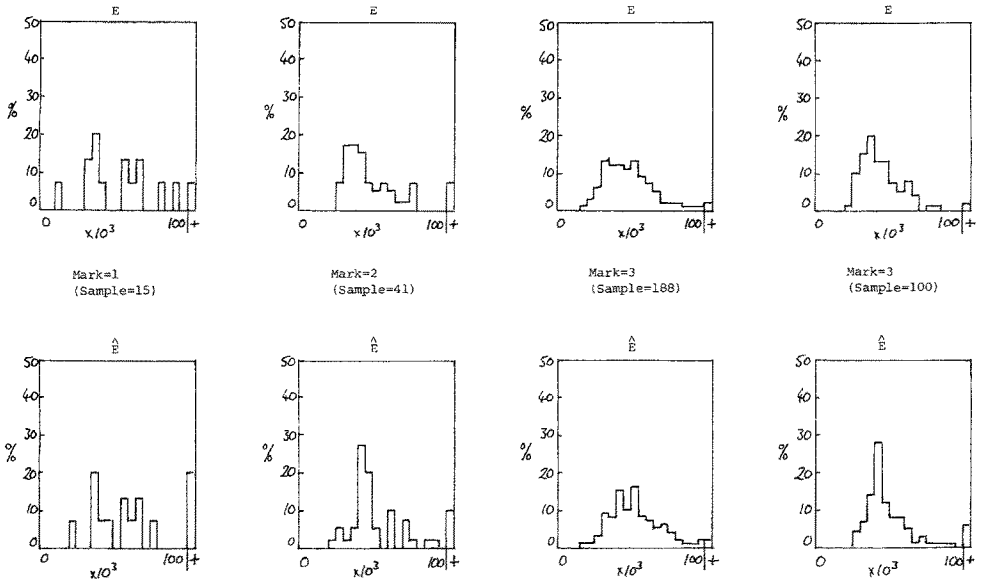

FIGURE 3 DISTRIBUTION OF EFFORT MEASURES OVER MARKS AWARDED (ASSIGNMENT 3)

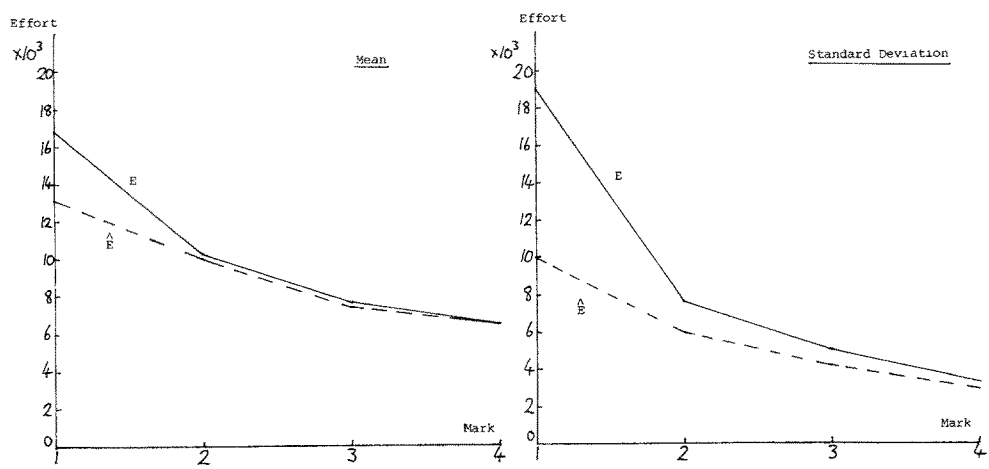

FIGURE 4 MEAN AND STANDARD DEVIATION OF EFFORT MEASURES AGAINST MARK (ASSIGNMENT 1)

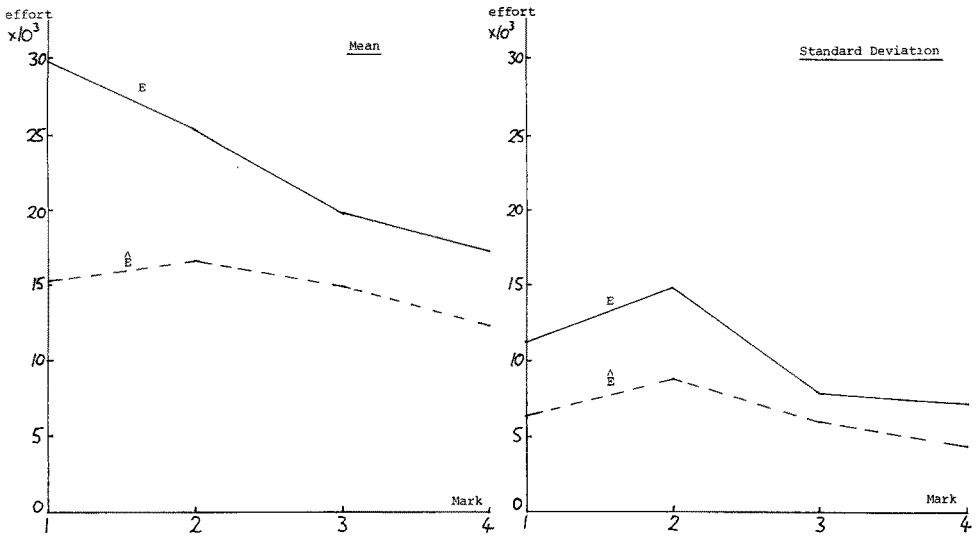

FIGURE 5 MEAN AND STANDARD DEVIATION OF EFFORT MEASURES AGAINST MARK (ASSIGNMENT 2)

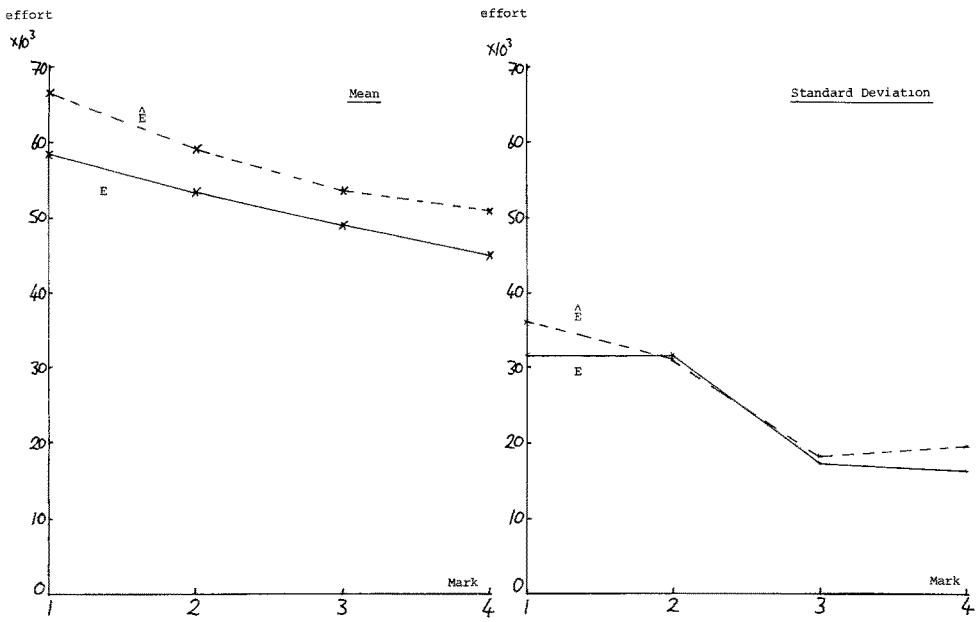

FIGURE 6 MEAN AND STANDARD DEVIATION OF EFFORT MEASURES AGAINST MARK (ASSIGNMENT 3)

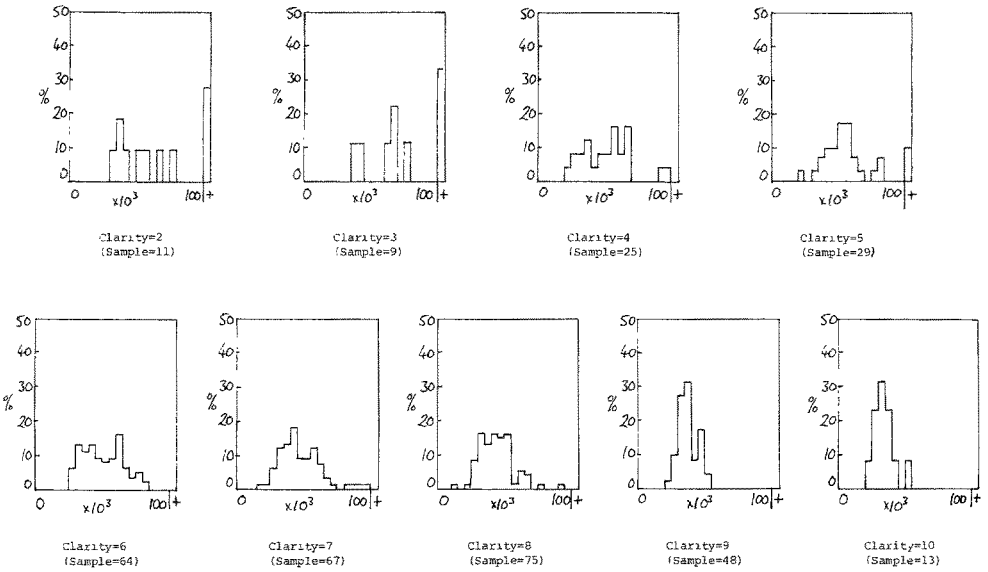

FIGURE 7 DISTRIBUTION OF E OVER CLARITY (ASSIGNMENT 3)

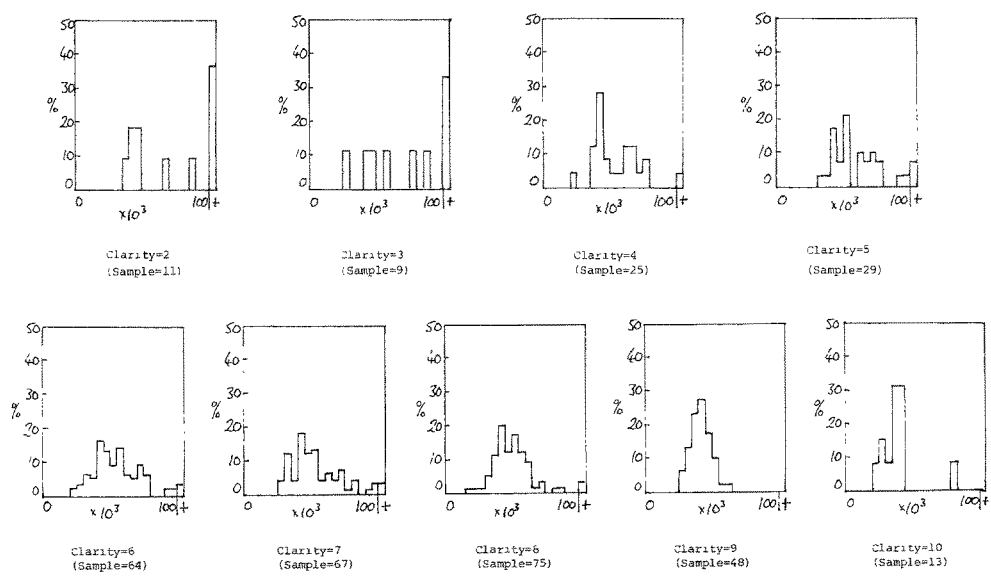

FIGURE 8 DISTRIBUTION OF $\hat{E}$ OVER CLARITY (ASSIGNMENT 3)

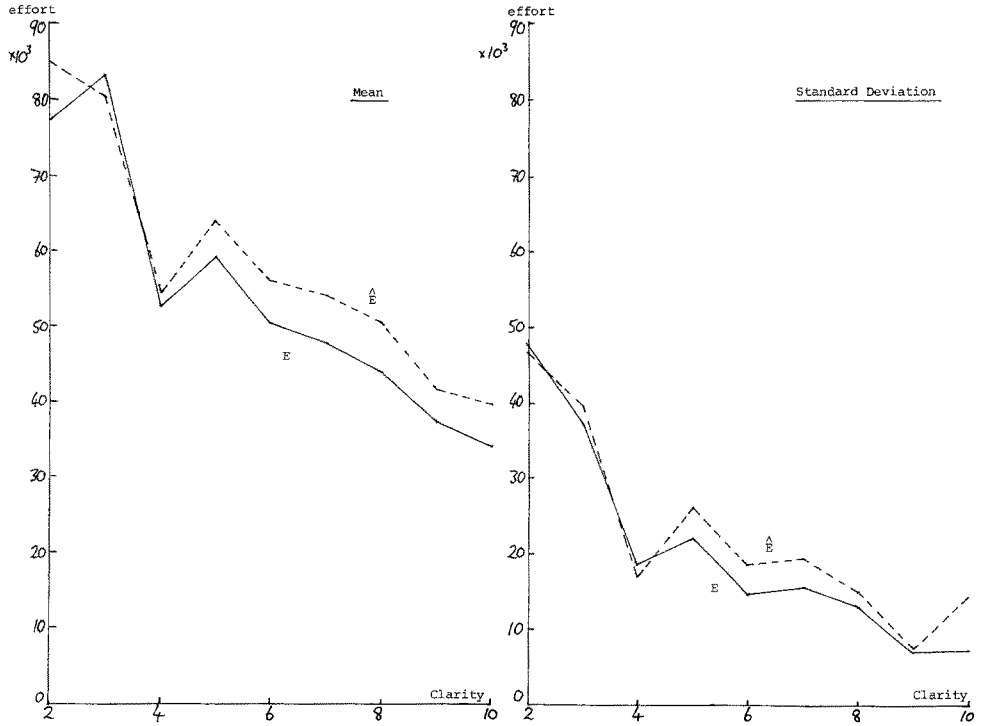

FIGURE 9 MEAN AND STANDARD DEVIATION OF EFFORT MEASURES AGAINST CLARITY (ASSIGNMENT 3)

However, one can detect an overall decline in the mean values of both E and $\hat{E}$ as the clarity increases, as is illustrated in Figure 9. Figure 9 also illustrates a decline in the standard deviation of both E and $\hat{E}$ as the clarity increases.

Figures 8 and 9 also indicate that Gordon's measure E_C $(=\hat{E})$ is no better indicator of clarity in this context than E.

5. SUMMARY AND DISCUSSION

The results of our experiment will probably disappoint the advocates and disciples of software science. There are a few observations which lend support to the theory, but most of our results are either negative or too weak to be useful. In terms of our original aims we can summarise them as follows:

(1) The correlation between the effort measures and the tutors' assessment of merit and clarity was small. We can say nothing more specific than that the better programs tended to have lower effort measures, and that these measures were less widely dispersed than for the poorer programs. There are certainly no grounds for using the effort measures as the basis for automatic grading of programs.

(2) The reliability of the length estimator $\hat{N}$ was good, at least in the mean. However, the variance of $N/\hat{N}$ indicates that there were a significant number of programs for which the estimate was not particularly accurate. In any case, for reasons indicated earlier, we do not regard $\hat{N}$ as a particularly important measure.

The level estimator $\hat{L}$ is far more important, since from it one can derive a value for E, even when V^* is unknown. Unfortunately this estimator did not prove very reliable - it showed an average discrepancy from L of nearly 40% on one assignment, and the average discrepancy over all assignments was between 10% and 20%. One reason for this may be that our values of L were themselves inaccurate, and we shall discuss this possibility in a moment.

(3) We could find no justification whatever for using λ to quantify the language level of PASCAL. Our observations show that λ was by no means constant and for reasons discussed earlier, we would not expect it to be. In our view a meaningful measure of language level can only be relative, and can be established only by comparing the volumes of a number of algorithms programmed in different languages.

We would stress that our findings, other than that relating to language level, should not be taken as an indictment of software science as a whole. Our experiment investigated the application of the science to an area where, as far as we know, it had not been applied before. The results indicate only that in this area software science has little to offer. Some possible reasons for this are discussed below.

Firstly, it could be argued that the failure to produce many positive results is due to weaknesses in the way the experiment was conducted. Some of these weaknesses derive from the fact that our data base was transient, in the sense that the only opportunity to measure each program was on the single occasion it passed through the compiler (we did not have the resources to store the source code of all 13,000 programs). This meant that it was impossible, for example, to see how the results would have varied with different counting schemes. Similarly, there was only one opportunity to make subjective assessments of programs - at the time of marking by tutors - before they were handed back to the students and disappeared into limbo. This meant that an anomalous measurement could not be traced back to the source and the reason for the anomaly adduced: all measurements had to be taken at face value.

Another major weakness lies in the derivation of V^* (and hence L and E). As described in section 4.2 we had considerable difficulty in deciding what the value of V^* should be, and felt obliged to use two values in all calculations performed. In our defence we might mention that we performed the same calculations with other, less likely, values of V^* (covering the range $6 \log_2 6$ to $10 \log_2 10$), and although the numeric results naturally vary, the same negative conclusions apply.

A third weakness may lie in our marking scheme for assignments, which was probably too coarsely grained. A finer grain may have shown up greater differences between "good" and "bad" programs. An attempt to provide a fine grain of assessment, and at the same time to eliminate spurious factors in the marking scheme, was made by obtaining the clarity measure for Assignment 3. In the event the results from the clarity measure were no better than those from the basic marking scheme.

Perhaps the most plausible reason for our largely negative results is that student programs are often badly written, particularly in the early stages of learning. They do not always use language features to best advantage, and they often contain impurities. The presence of impurities is often quoted in the literature as being the cause of anomalies in software science measures: it seems that in the case of student programs the impurities may be sufficient to render the science almost useless. This conclusion will certainly be drawn by those who prefer to attribute our results to deficiencies in the sample rather than deficiencies in the theory.

In summary we feel that our experiment, despite its weaknesses, indicates that software science has little to offer in the area of student programming. It may also have limitations in other fields: if the science is ever to emerge as a major tool these limitations need close investigation.

ACKNOWLEDGEMENTS

We would like to thank Dr. Jean-Louis Lassez for his comments on the original plan for this experiment, and Dr. Jim Welsh for his contributions to our discussions. We are also grateful to the tutors for providing their experience and to the students who unwittingly acted as guinea pigs.

REFERENCES

Elshoff J.L. (1978) "An investigation into the effects of the counting method used on software science measurements", ACM Sigplan Notices, Vol 13, No 2.

Fitzsimmons A., Love T. (1978) "A review and evaluation of software science", ACM Computing Surveys, Vol 10, No 1.

Gordon R.D. (1979) "A qualitative justification for a measure of program clarity", IEEE Trans. on Software Engineering, Vol 5, No 2.

Halstead M.H. (1977) "Elements of Software Science", Elsevier North-Holland, N.Y.

Van der Knijff D.J.J. (1978) "Software physics and program analysis", Australian Computer Journal, Vol 10, No 3.

APPENDIX

```
  1    PROGRAM ASS1;
  2
  3       VAR
  4           SUBJ, GRADE, CP: INTEGER;                    (*SUBJECT CODE, GRADE, CREDIT POINT VALUE*)
  5           I, N: INTEGER;                               (*LOOP CONTROL, NO. OF DATA PAIRS*)
  6           ERROR: BOOLEAN;                              (*TRUE IF DATA ERROR DETECTED*)
  7
  8       BEGIN
  9           READLN(N);                                   (*NO. OF DATA PAIRS*)
 10           FOR I := 1 TO N DO
 11               BEGIN
 12                   READLN(SUBJ, GRADE);                 (*READ NEXT DATA PAIR...*)
 13                   WRITE(SUBJ, GRADE);                  (*...AND PRINT IT*)
 14                   ERROR := FALSE;                      (*NO DATA ERRORS YET*)
 15                   IF (GRADE < 1) OR (GRADE > 7)        (*INVALID GRADE*)
 16                   THEN
 17                       BEGIN
 18                           WRITE('      ERROR IN GRADE');
 19                           ERROR := TRUE
 20                       END;
 21                   CASE SUBJ OF                         (*ASSIGN CREDIT POINTS FOR SUBJECT*)
 22                   100, 110:
 23                       CP := 8;
 24                   200, 201, 300, 301:
 25                       CP := 7;
 26                   390, 391:
 27                       CP := 5;
 28                   OTHERS:                              (*INVALID SUBJECT*)
 29                       BEGIN
 30                           WRITE('      ERROR IN SUBJECT');
 31                           ERROR := TRUE
 32                       END
 33                   END;
 34                   IF (GRADE = 1) OR (GRADE = 2)        (*FAIL GRADES*)
 35                   THEN
 36                       CP := 0;
 37                   IF NOT ERROR                         (*NO DATA ERRORS DETECTED...*)
 38                   THEN
 39                       WRITE(CP);                       (*...SO PRINT CREDIT OBTAINED*)
 40                   WRITELN
 41               END
 42       END.

NO ERROR(S) DETECTED
```

```
 1    PROGRAM ASS2;
 2
 3       VAR
 4           AH, AM, CH, CM: INTEGER;                              (*ACTUAL & CLOCK HOURS & MINS*)
 5           STICKING: BOOLEAN;                                    (*TRUE WHEN CLOCK IS STICKING*)
 6
 7       BEGIN
 8           AH := 0;
 9           AM := 0;
10           CH := 0;
11           CM := 0;                                              (*CLOCK STARTS AT ZERO*)
12           STICKING := FALSE;
13           WRITELN('            CLOCK TIME              ACTUAL TIME');
14           REPEAT                                                (*MAIN SIMULATION LOOP*)
15               AM := AM+1;                                       (*ACTUAL TIME ALWAYS ADVANCES BY A MIN.*)
16               IF AM = 60                                        (*ANOTHER ACTUAL HOUR DONE*)
17               THEN
18                   BEGIN
19                       AM := 0;
20                       AH := AH+1
21                   END;
22               IF ((CM = CH*5) OR (CM = 30)) AND NOT STICKING    (*CLOCK STICKS*)
23               THEN
24                   STICKING := TRUE
25               ELSE
26                   BEGIN                                         (*NORMAL CASE*)
27                       STICKING := FALSE;                        (*UNSTICK CLOCK*)
28                       CM := CM+1;                               (*CLOCK ADVANCES BY A MIN.*)
29                       IF CM = 60                                (*ANOTHER CLOCK HOUR DONE*)
30                       THEN
31                           BEGIN
32                               CM := 0;
33                               CH := CH+1
34                           END
35                   END;
36               IF CM = 0                                         (*PRINT-OUT REQUIRED ON THE HOUR*)
37               THEN
38                   WRITELN(CH, CM, AH, AM)
39           UNTIL CH = 12                                         (*STOP AT CLOCK'S MIDDAY*)
40       END.

NO ERROR(S) DETECTED
```

```
 1    PROGRAM ASS3;
 2
 3        CONST
 4            CHLIM = 8;                                  (*MAX. NO. OF CHARS PER WORD*)
 5            WDLIM = 20;                                 (*MAX. NO. OF WORDS IN TEXT*)
 6
 7        TYPE
 8            STRING8 = PACKED ARRAY[1 .. CHLIM] OF CHAR;
 9
10        VAR
11            CHCNT, WDCNT, I: INTEGER;                   (*CHARACTER COUNT, WORD COUNT, AUXILIARY*)
12            CH, KEY: CHAR;                              (*CURRENT CHARACTER, 1ST CHARACTER OF LAST WORD*)
13            FOUND: BOOLEAN;                             (*USED IN SEARCH FOR DUPLICATES*)
14            S: STRING8;                                 (*CURRENT WORD*)
15            TEXT: ARRAY[1 .. WDLIM] OF STRING8;         (*STORAGE FOR TEXT*)
16
17        BEGIN
18            WDCNT := 0;                                 (*INITIALISE WORD COUNT*)
19            WHILE NOT EOF DO                            (*INPUT & STORAGE LOOP*)
20                BEGIN
21                    REPEAT                              (*LOOK FOR START OF NEXT WORD IF ANY*)
22                        READ(CH)
23                    UNTIL (CH >= 'A') AND (CH <= 'Z') OR EOF;
24                    IF NOT EOF                          (*IF THERE WAS A WORD...*)
25                    THEN
26                        BEGIN                           (*...READ IT IN*)
27                            CHCNT := 1;                 (*INITIALISE CHARACTER COUNT*)
28                            REPEAT                      (*DEAL WITH EACH ALPHABETIC CHAR.*)
29                                IF CHCNT <= CHLIM       (*IF WORD NOT TOO LONG...*)
30                                THEN                    (*...THEN...*)
31                                    BEGIN
32                                        S[CHCNT] := CH; (*...STORE CHARACTER...*)
33                                        CHCNT := CHCNT+1
34                                    END;                (*...OTHERWISE IGNORE IT*)
35                                READ(CH)
36                            UNTIL (CH < 'A') OR (CH > 'Z');
37                            FOR I := CHCNT TO CHLIM DO  (*PAD WORD WITH SPACES*)
38                                S[I] := ' ';
39                            I := 1;                     (*INITIALISE TO LOOK FOR DUPLICATES*)
40                            FOUND := FALSE;             (*SCAN TEXT ALREADY READ*)
41                            WHILE (I <= WDCNT) AND NOT FOUND DO
42                                BEGIN
43                                    FOUND := TEXT[I] = S;
44                                    I := I+1
45                                END;
46                            IF NOT FOUND                (*NOT A DUPLICATE, SO...*)
47                            THEN                        (*...STORE THE WORD*)
48                                BEGIN
49                                    WDCNT := WDCNT+1;
50                                    TEXT[WDCNT] := S
51                                END
52                        END                             (*END OF DEALING WITH ONE WORD*)
53                END;                                    (*END OF INPUT & STORAGE LOOP*)
54            KEY := S[1];                                (*1ST. LETTER OF LAST WORD*)
55            FOR I := 1 TO WDCNT DO                      (*PRINT EACH WORD, FLAGGED IF NECESSARY*)
56                BEGIN
57                    WRITE(TEXT[I]);
58                    IF TEXT[I][1] = KEY
59                    THEN
60                        WRITE(' *');
61                    WRITELN
62                END
63        END.

NO ERROR(S) DETECTED
```