Because the thrust of this research was on prediction, rather than productivity measurement, no standards of performance could be expected.

GUIDE Productivity Group (43)

Initially the objective of the GUIDE Productivity Group's project was to identify methods of measuring productivity as they relate to the design and programming functions of project development. A report was produced at GUIDE 41 in 1975. The scope of the project was increased to include system maintenance activities and a report on progress was published in February 1977. The full report reflecting measurement within the entire system development life cycle was presented in 1978.

The basis of this approach is that the program development phase of the system life cycle be sub-divided into a series of activities, each with a main method and alternative methods of measuring productivity within that phase.

Phase	Main Method of Measuring
Review technical design specification	Number of pages reviewed per actual time in man-hours and elapsed time.

Phase	Main Method of Measuring
Defining Program Structure	The number of logic units (flow-chart blocks, decision table entries, pseudo-code lines etc.) per actual time in man-hours and elapsed time.
Code and Compile the program	All submitted lines of code per actual time in man-hours and elapsed time.
Program Inspection	Lines of code submitted for inspection per actual time in man-hours and elapsed time.
Prepare test plan	Number of program options to be tested per actual time in man-hours and elapsed time.
Prepare test data	Number of test data elements created per actual time in man-hours and elapsed time.
Prepare job control language	Number of job control statements per actual time in man-hours and elapsed time.

Phase	Main Method of Measuring
Execute test run	Number of lines of code submitted for test per actual time in man-hours and elapsed time.
Test result inspections	Number of pages of results inspected per actual time in man-hours and elapsed time.

This approach does take cognisance of the multi-dimensional activities within programming. However, not only is there no indication of what can be regarded as acceptable performance in each activity, but some measures (e.g, in terms of the number of pages produced or read) do not appear to be meaningful.

Because of differences in different organizations in both the stages of systems development and the definitions of the activities and deliverables of each stage, the report suggests that:-

' - each installation measure those activities applicable to that installation;

' - no attempt be made to make comparisons between installations.'

(44)

Although in the main text of their report the GUIDE Productivity Group advocate sub-dividing the programming activity into various components, another approach is taken in Appendix 'A' of the report where methods of calculating ratios to reflect the rate of project development are suggested.

The ratio suggested to reflect the performance on the entire system life cycle is:-

$$\frac{\text{Number of Functions in the system}}{\text{Accumulated time for each activity}} \qquad (45)$$

A Function is defined in their Glossary as:-

> 'an action or activity at the lowest definable level that must be performed in order to meet specifications; that is, update, validation, print, read, write etc.' (46)

(see Chapter 3 for the definition of Function in the context of this research, and Section 10.1.1.)

Parikh (47)

In this article Parikh appears reluctant to make any dogmatic claims regarding the effectiveness of methods of measuring programmer productivity. He makes a number of suggestions, but is cautious about the use of most of the methods mentioned e.g.

> '... lines of code produced and time expended may indicate productivity'. (48)

Parikh cautions that unless lines of code be precisely defined, productivity measurements and comparisons may be misleading. Having said that, he suggests that with the increased use of structured methodologies the 'function' may become the basic unit for the system or the program. Obviously, though, he feels that the use of the structured metnodologies is not yet sufficiently wide-spread to discard the lines of code measure. (See Table 6).

Parikh suggests that a productivity index (calculated for each programmer, or programming team) gives a more comprehensive measure of performance than any single measure. (See Table 6).

This index is calculated by summing the average rating for each measure in the matrix. Although some of these measures are subjective - and may need more than one expert to help calculate the rating - Parikh suggests that this index enables the comparison of programmer performance within the same organization. In fact, he feels that provided the method is standarized, it could enable comparisons to be made between organizations.

Table 6 Sample of Programmer Productivity Metrics (from Parikh).

CHARACTERISTICS OF ATTRIBUTE / NAME OF ATTRIBUTE	METRIC UNITS	BEST VALUE (no value for better)	PLANNED (quite satisfactory)	% WEIGHT	MORE IMPORTANT THAN	REMARKS OF REFERENCES
Quantity of Work	Lines of code per day					
Errors found during inspections	Errors per 1000 lines of code					
Defects found during the first month of production	Errors per 1000 lines of code					
Documentation Quality	Grade points: 7 for best 1 for worst					
Compilations of program(s)	Number of compilations					
Core used by program(s)	k bytes					
Execution speed of program(s)	Seconds or minutes					
Program(s) completed within time schedule	7 for before schedule 1 for very late					
Program(s) completed within budget	7 for less than budget 1 for budget exceeded					
Requirements were met	7 for totally satisfactory 1 for unsatisfactory					
Number of statements in program(s)	Number					
Maintainability of program(s)	7 for maintainability 1 for unmaintainability					

In addition to this index, Parikh advocates that the program design phase should also be measured. This measure will depend on the design tool being used (e.g. pseudo-code or Warnier-Orr diagrams) and should be expressed in time units. Again the note of caution is evident, e.g.

> 'The measurement and comparison of design personnel's productivity in different situations may not be accurate. Use this method with great caution.'
> (49)

6.2.3.1.2 Measurement in terms of Multiple Variables whose summation reflects the size of the task, with a standard of performance for each activity

I was not able to identify any research done which provided results which fall into this category.

6.2.3.2 Measurement in terms of Multiple Variables whose summation reflects the size of the task, but which requires modifications to accommodate particular circumstances.

This is the final category to be described and it is characterized by a mixture of Input Factors and Output Factors for each program activity. By definition there can be no standard of performance within this category. Again I was not able to find research with results which could be placed in this category.

CHAPTER 6 - FOOTNOTES

(1) BERNSTEIN, M. I.: 'Hardware is Easy: it's Software that's Hard': Datamation: November 1978: p. 32.

(2) Ibid., p. 34.

(3) COOKE, L. H. Jr.: 'Programming Time versus Running Time': Datamation: December 1974: pp. 56-58.

(4) SACKMAN, H., ERIKSON, W. J., GRANT, E. E.: 'Exploratory Experimental Studies Comparing Online and Offline Programming Performance': Communications of ACM, Volume 11, Number 1: January 1968: pp. 3-11.

(5) COMPER, F. A.: 'Project Management for Systems Quality and Development Productivity': Proceedings of the Application Development Seminar, Monterey, California 1979, p.21.

(6) CASHMAN, M. W.: 'An Interview with Professor Dr. Edsger W. Dijkstra': Datamation: May 1977, p. 166.

(7) JONES, T. C.: 'Program Quality and Programmer Productivity': TR 02.764, IBM Corporation, San Jose: January 1977.

: 'Optimizing Program Quality and Programmer Productivity': Proceedings GUIDE 45: Atlanta, Georgia: November 1977.

: 'Measuring Programming Quality and Productivity': IBM Systems Journal: Volume 17, Number 1, 1978.

(8) BERNSTEIN, M. I.: op.cit., p. 34.

(9) REYNOLDS, W. J. F.: 'Programming Productivity', IBM Technical Report SETR 76-044, November 1976.

(10) SCOTT, R. F. and SIMMONS, D. B.: 'Programmer Productivity and the Delphi Technique': Datamation: May 1974: pp. 71.-73.

(11) BAKER, F. T.: 'System Quality through Structured Programming': Proceedings AFIPS Conference, Volume 41, Montvale, New Jersey.

(12) BROOKS, Jr. F. P.: 'The Mythical Man-Month': Addison-Wesley: Reading, Massachusetts: 1975, p. 93.

(13) Ibid., p. 90.

(14) CAMP, J. W. and JENSEN, E. P.: 'Cost of Modularity: Proceedings of Computer Software Engineering Conference: Polytechnic Institute of New York: April 1976: pp. 215-224.

CHAPTER 6 - FOOTNOTES (Cont'd.)

(15) FRANK, W. L.: 'The New Software Economics (Part 2)': Computerweek: Johannesburg: 5 March 1979: pp. 8-9.

(16) JEFFERY, D. R. and LAWRENCE, M. J.: 'An Interorganizational Comparison of Programming Productivity': Information Systems Forum, University of New South Wales, Australia, March 1979.

(17) JOHNSON, J. R.: 'A Working Measure of Productivity': Datamation: February 1977 pp. 106-110.

(18) LEHMAN, J. H.: 'How Software Projects are really Managed': Datamation: January 1979: pp. 119-125.

(19) MENARD, J. B.: 'Exxon's Experience with the Michael Jackson Design Method': Proceedings of Application Symposium: Monterey, California, 1979: pp. 143-148.

(20) WALSTON, C. E. and FELIX, C. P.: 'A Method of Programming Measurement and Estimation': IBM Systems Journal: Volume 16, Number 1, 1977.

(21) Ibid., p. 55.

(22) Ibid., p. 62.

(23) Ibid., pp. 64-65.

(24) ALBRECHT, A. J.: 'Measuring Application Development Productivity': Proceedings of the Application Development Symposium: Monterey, California, 1979, p. 53.

(25) CHRYSLER, E.: 'Some Basic Determinants of Computer Programming Productivity': Communications of ACM, Volume 21, Number 6, June 1978: pp. 472-483.

(26) Ibid., p. 474.

(27) Ibid., p. 475.

(28) Ibid., p. 480.

(29) HALL, G. E.: 'Programming Productivity': Proceedings Programmer Productivity Forum: Infotech International, London: October 1978.

(30) Ibid., p. 14.

(31) Ibid., p. 13.

(32) Ibid., p. 15.

(33) FITZSIMMONS, A. and LOVE, T.: 'A Review and Evaluation of Software Science : ACM Computing Surveys: March 1978: pp. 3-18.

(34) Ibid., p. 5.

(35) Ibid., p. 5.

CHAPTER 6 - FOOTNOTES (Cont'd)

(36) Ibid., p. 6.

(37) Ibid., p. 8.

(38) Ibid., p. 9.

(39) Ibid., p. 9.

(40) PEEPLES, D. E.: 'Measure for Productivity': Datamation: May 1978: pp. 222-230.

(41) BELFORD, P. C. and BROGLIO, C.: 'A Quantitative Evaluation of the Effectiveness of Quality Assurance as Experienced on a large-scale, Software Development Effort': Proceedings of ACM Software Quality and Assurance Workshop, 1978: pp. 173-180.

(42) Ibid., p. 178.

(43) GUIDE: Report of the Productivity Measurement Group: GUIDE 49, November 1978.

(44) Ibid., p. 13.

(45) Ibid., p. 71.

(46) Ibid., p. 77.

(47) PARIKH, G.: 'How to Measure Productivity': Computing South Africa, Number 1, Volume 40, 29 April 1981.

(48) Ibid., p. 14.

(49) Ibid., p. 15.

CHAPTER 7: SOME SIGNIFICANT PROBLEM AREAS

As a result of the literature survey (see Chapter 6) it is concluded that before a method of measuring programmer productivity will be viable in a development department, not only must it meet the requirements identified in Chapter 8, but certain problems must be resolved. These problems fall into 3 broad categories:-

- problems which result from the difficulty of measuring certain project variables;
- problems uniquely associated with the programming environment;
- problems related to the specifying, measuring and controlling of program quality.

7.1 The Difficulty of Measuring Certain Variables

In Chapter 5 some factors which influence programmer productivity were identified. It is difficult to assign a value to some of these factors because different people with different skills and different goals will place different values on the same factor. This lack of objectivity is problematic in the key areas of:-

- the programmers - their skills and motivation;
- the type of programming to be done, and its complexity;
- the influence of the work environment.

Any lack of objectivity in these areas will prevent valid comparisons of programmer performance.

7.1.1 The Programmers

In this section variables difficult to measure, which are associated directly with programmers, will be identified.

7.1.1.1 Programming Skill

Possibly because it is acknowledged that there is a wide variance in programming skills (see Section 5.1.1), the concept of introducing levels of seniority in the programming ranks - which range from trainee programmer through junior programmer to senior programmer - has an element of appeal. Unfortunately there is an intrinsic weakness in this concept because it is difficult to measure the skill-levels of programmers objectively. This subjectivity results in the same name being given to people in different companies performing different activities which demand different levels of skill. To illustrate the point, the title CHIEF PROGRAMMER, has been used in each of the following circumstances:-

- in the context of the New York Times project in 1972 documented by Baker and Mills (1);

- As the programmer with the most experience at a small installation (e.g. Hubert Davies Heavy Equipment (Pty) Limited, who in 1978 had a development department which consisted of what they termed 'chief programmer' and four trainee programmers);

7.1.1 The Programmers

In this section variables difficult to measure, which are associated directly with programmers, will be identified.

7.1.1.1 Programming Skill

Possibly because it is acknowledged that there is a wide variance in programming skills (see Section 5.1.1), the concept of introducing levels of seniority in the programming ranks - which range from trainee programmer through junior programmer to senior programmer - has an element of appeal. Unfortunately there is an intrinsic weakness in this concept because it is difficult to measure the skill-levels of programmers objectively. This subjectivity results in the same name being given to people in different companies performing different activities which demand different levels of skill. To illustrate the point, the title CHIEF PROGRAMMER, has been used in each of the following circumstances:-

- in the context of the New York Times project in 1972 documented by Baker and Mills (1);

- As the programmer with the most experience at a small installation (e.g. Hubert Davies Heavy Equipment (Pty) Limited, who in 1978 had a development department which consisted of what they termed 'chief programmer' and four trainee programmers);

as in an advertisement in Computerweek 26 January 1982, for a chief programmer who needed a knowledge of COBOL and FORTRAN but because the successful applicant would control the systems development division of the department, experience in systems analysis was essential.

One reason for these discrepancies is that there is not an objective method of measuring the level of programming skills, which enables a person to qualify as a chief programmer, or any other level of programming.

In his research, Chrysler attempts to overcome this problem by measuring programmer competence in terms of Experience in Months. For example, he measures:-

- months experience at a particular facility;

- months experience programming business applications;

- months experience at programming,

etc. (2)

Of these, Chrysler concludes that the two factors which influence programmer productivity most significantly, are:-

- programmer experience (in months) at the particular facility, and

- as in an advertisement in Computerweek 26 January 1981, for a chief programmer who needed a knowledge of COBOL and FORTRAN, but because the successful applicant would control the systems development division of the department, experience in systems analysis was essential.

One reason for these discrepancies is that there is not an objective method of measuring the level of programming skills, which enables a person to qualify as a chief programmer, or any other level of programming.

In his research, Chrysler attempts to overcome this problem by measuring programmer competence in terms of Experience in Months. For example, he measures:-

- months experience at a particular facility;

- months experience programming business applications;

- months experience at programming,
  etc. (2)

Of these, Chrysler concludes that the two factors which influence programmer productivity most significantly, are:-

- programmer experience (in months) at the particular facility, and

- programmer experience (in months) at COBOL programming.

(3)

This raises another problem. Is it equitable to measure programmer skill in terms of the number of months experience spent in a particular environment? Chrysler is not the only person who appears to assume that this is valid. The annual salary survey of Data Processing staff conducted by the Johannesburg based company, Computer Personnel Limited (CPL), is also based on the number of months experience in each job category as a substitute for actual skills measurement.

This approach is probably used, not because it is valid, but because it is possible to count months of experience. It is, in fact, invalid. Work-loads can be allocated in such a way that it is possible to use the skills developed early in a programming career during subsequent years, without any significant increase in skills (This is sometimes referred to as 5 years experience amounting to no more than 1 years experience repeated 5 times). I conclude that the passing of time and the level of skills acquired during the passing of time are not synonymous, and it cannot be assumed that they correlate.

If this approach to measuring programmer skills is rejected, the problem remains - how can programming skills be measured with a degree of objectivity?

7.1.1.2 Programmer Motivation

Limited output can be expected from a programmer with a high level of motivation but a low level of skill. The converse is also true. A highly skilled programmer's performance can be substantially degraded by low motivation. To measure motivation and its influence on productivity is difficult. One way to overcome the problem is to identify and create the environment in which each worker is equally (highly) motivated. This requires consistently high levels of managerial skill. Some of the reasons why it remains a problem in program development environment, are outlined in this section.

7.1.1.3 The Influence of Interpersonal and Group Behaviours

Even if programmers are not working in one of the programming team environments (see Chapter 3 for definitions), the programming activity requires sufficient interpersonal contact with people who are involved directly or indirectly with the project for behavioural factors to influence programmer performance.

It is interesting to notice that in his experiment, Cooke found that:-

> '....personality and attitudes of the participants affected the results somewhat, but we don't know how, or by how much.'
>
> (4)

The extent to which friendly, competitive, shy, aggressive, accommodating, etc., personality types - influence interpersonal and group behaviour - is difficult to determine.

7.1.1.4 The Impact of Cultural and Language Differences

Because the main thrust of computer technology has come from the English speaking Western world, programmers whose home-language is not English, or whose culture is not Western orientated, will be (at least initially) at a disadvantage. To measure the impact of this disadvantage on the performance of a programmer writing, say a COBOL program, is difficult to measure with any degree of certainty.

7.1.2 The Work Being Done

Programming work is often not developing new systems, or doing work towards getting new systems operational. A large percentage of programming activity is involved in maintaining (fixing) or enhancing (adding new functions to) existing systems (see Chapter 1).

The objective of this research is not directed towards the maintenance or enhancement environment (see Chapter 2). However, even in the single area of development work, the types of programs being written differ widely between installations and projects within a single installation (e.g. complex validates, simple extract and print, difficult updates, programs operating in a data base, data communication environment, etc.).

Determining the influence of these differences on programmer performance, is problematic.

Intuitively, it seems inequitable not to take the differences in programs into account when measuring programmer performance. More difficult programs surely will result in apparently poorer productivity figures, (even for the same programmer) and programmers with different experiences may find the same program has different levels of difficulty. To support this, any estimating (e.g. IBM (5)), requires factors to be introduced to represent the programmers general experience and particular 'know-how' for the type of program being developed. As was noted in Section 6.2.2.2 Chrysler (6) found that the number of output fields,
control breaks and totals and
input files
within a program, were significant factors which influence programmer performance. Each of these factors contribute to program complexity.

I suggest there are a number of ways of looking at program complexity. A program can appear complex just because programmers are not available with the necessary skills to develop it. (Compare the concept of Break-Thru Technology defined in Chapter 3). Program complexity can also be introduced into the program by the particular internal structures and interfaces chosen by the programmer at program design stage. It will be established in Section 7.4.2 that simplicity is one of the characteristics of a quality program. One of the ways of achieving simplicity is to write structured programs.

If programmers have developed programs without adhering to the disciplines and constraints of structured programming, complexity could be introduced into the algorithm by the particular style of programming.

These types of complexity must be contrasted with the complexity of a program which exists no matter who develops the program. It is this type of complexity which needs to be measured to enable programmer performance measurement to be equitable (see Section 8.2 and Chapter 9).

7.2 Problems Associated with the Programming Environment

Certain problems are introduced into any proposed method of measuring programmer productivity by the peculiarities of the application development environment.

7.2.1 Writing code is not the Programmer's only task

There is no direct relationship between the skills required to solve business problems, write syntactically correct code, set up appropriate test data, debug or accurately document a program. (One of the teams working in the data processing department of The Standard Bank gave the responsibility of program debugging to one specific team member because of his particular skill in that area). To measure programmer performance in terms of any one of these activities, to the exclusion of the others, is likely to be inadequate. This problem has been directly responsible for certain methods of measuring programmer performance being proposed which measure each activity separately (e.g. see (7)).

7.2.2 Programming is only one activity in the Application Development Cycle

The programming activity is part of a cycle of activities which are interrelated in such a way that events earlier in the cycle can influence later activities significantly. For example, the programming activity can be affected by the quality of the analysis and particularly, the design phases of development cycle.

The extent of this problem is illustrated by the approach to systems design advocated by T. de Marco (8). He suggests that the design activity continue until the individual program modules have been identified. It is only at this stage that de Marco recommends that the modules be grouped into appropriate programs. The distinction between systems design and program design activities becomes unclear. The more complete the systems design, the less program design is required, whereas incomplete systems design will force the program design phase to be extended. The extent and quality of systems design influence the duration of the programming phase to the degree that measuring programmer performance without controlling the quality of systems design will probably lead to suspect results.

7.2.3 The Quality of Program Specification

Ambiguous and incomplete program specifications will tend to extend program development time. To measure programmer performance equitably, therefore, requires a standardization of the quality of program specifications. To achieve this standardization, even within one installation, is difficult.

Using Reviews and Walk-Thrus (see (9) and (10)) may introduce a form of control, but the subjectivity of these approaches does not guarantee that each program specification is of equal quality. Judgement of programmer performance without this quality control will not be unbiased.

7.2.4 The Problem of Program Quality

In any production environment quality has a large influence on productivity. Bernstein writes:

> 'It is not true that higher quality follows from higher productivity. If we succeed in improving software development productivity without improving the quality of software, the results may well be lower cost, poor quality software ..... a sacrifice of quality in exchange for a gain in productivity.'
>
> (11)

Not only is it possible to sacrifice quality for an apparent improvement in productivity, sometimes too much emphasis is placed on quality, which results in low levels of performance.

So, the comparison of programmer productivity figures, or the calculation of efficiency ratios in relation to a standard of performance, will tend to be meaningless unless there is some form of program quality control.

Ideally, the quality requirements of a worker should be defined as part of the specified standard of performance. (That is, the standard is a certain number of output units of particular quality).

Unfortunately quality - not just program quality - is difficult to measure. Sometimes quality measurement is attempted in terms of the number of units rejected by a group of inspectors. However, this 'after-the-event' quality measure needs to be done by checking and possibly rechecking and there is a basic concern with this approach. There is a tendency to stop checking if there is a shortage of resources (people and money), or time. So checking is often done because time and resources are available rather than because quality control is essential or achieved.

In the programming environment this problem is compounded because:-

- there is a school of thought which holds that program quality is difficult to measure;

- there is an absence of a widely accepted definition of software quality;

- there is little consensus as to how program quality should be measured, even if it can be measured;

- there are few quantitative measures of the quality software product.

7.2.5 Most systems which are implemented are prototypes

In his report, published in Datamation, of a workshop on the state of the art in software quality assurance, sponsored by the Association for Computing Machinery, R. L. Patrick notes (12) that the design-development-production cycle for hardware is mature and well-tuned. In contrast, however, most software applications which are designed, developed and installed are prototypes. Most systems, in the form in which they are developed are unique to the extent that certainty regarding their appropriateness only comes after they have been implemented (see (13)).

This results in two particular problems:-

i) Any change to a system already implemented is potentially expensive and disruptive to the User (and will inevitably lead to a questioning of data processing staff's credibility).

ii) If most of the systems implemented were prototypes the developer's performance will be distorted by a learning curve for each application developed. Consequently there is a strong tendency to adjust performance figures to allow for this learning curve (for example estimating). This makes the comparison of programmer productivity figures misleading.

7.2.6 Each System shipped from the Development Department is an incomplete product

The maintenance of existing systems often demands a high percentage of programming resources. Opinion on just what this percentage is varies. C. P. Lecht estimated the figure at between 60% and 70% in 1977. He predicts that by 1982 about 80% of programming staff will be allocated to maintenance (quoted by (14)).

Lecht's assessment appears inflated compared with the results of a survey conducted in 1978 by IBM among some of their European customers (quoted by (14)). The IBM survey revealed that 10% of development department resources were being used for system conversions and 46% of resources on maintenance.

Whether 'maintenance' is regarded as correcting malfunctions to enable the system to perform as specified, or whether it also includes adapting the system to meet a new set of requirements, each system shipped from the development department requires maintenance. Jones writes:-

> 'One of the few items of wisdom in programming about which almost everyone seems to agree is that there is no such thing as a final program; changes always occur.'
> (15)

This creates the problem that even though a program passes all the required phases of reviewing and testing, when eventually it is executed in a live environment, it is still an incomplete product. If the program is incomplete, the programmer's responsibility for the program is also incomplete.

To measure the performance of a programmer in terms of a completed task when the task is still incomplete, is a questionable practice - but to delay measurement for months (or even years) because the program still requires further maintenance, may be equally unhelpful.

7.2.7 The Objectivity of the Data

Keeping accurate logs of the number of manhours spent on the development of each program within a system does not appear to be a common practice. Some development departments do not attempt to keep any record of time spent on developing each individual program. Other departments do keep a record of how programmers have used their time, but this is often done on a weekly basis. Genuine memory lapses on the part of the programmer will tend to make program development time suspect unless records are kept up-to-date on a short-term (daily?) basis. This is particularly so if the programmer is working on more than one program at any one time.

In reply to my question on how he controlled the quality of the data used in his research, Chrysler wrote:-

> '.... the amount of time devoted to each program was logged on a daily time sheet by the programmer responsible for developing the program. The accuracy of the programmer logs depends on the integrity of each programmer ... One is forced to assume that errors, either accidental or intentional, were randomly distributed across all programs and programmers'.
>
> (16)

7.2.8 Differences in the steps taken in the Systems Development Life Cycle

In the document prepared by the Productivity Measurement Group of GUIDE, the point is made:-

> '..... There is, however, a very basic rule an installation must adhere to if a successful measure of productivity is to be made ..... a definite effort must be made to be consistent in the System Life Cycle steps you take .....'
>
> (17)

Even in development departments like the Standard Bank, where the development life cycle is well defined and standardized, variations in the phases of application development occurred. For some valid reason no Feasibility Study would be carried out on one project, or no Detailed Design on another, or detailed Program Specifications would be produced on one project, but programming would be done from the Technical Specification on other projects. Although I am not able to substantiate the impact each of these situations will have on the performance of the development staff, unless the program development time is held constant (see Chapter 3 for definition), it is invalid to compare programmer productivity between installations, or within a single installation (see Section 9.1).

7.2.9 The lack of an Industry-wide standard of Programmer Performance

One of the requirements of a method of measuring productivity is that it provides a standard of performance (see Chapter 8). In Section 6.2 various attempts at identifying an industry standard were summarized.

The results of trying to identify a standard of performance which could be valid industry-wide are not encouraging. Diverse opinions are expressed in terms of suggested elements which should be measured and the performance level expected of the programmers.

For those who suggest that programmer performance should be measured in terms of lines of code produced, the ranges are almost too diverse to be meaningful (these figures are summarized from Section 6.2.2.1.2).

	Range
Low	3 to 22 lines of code per man-day
Average	4 to 26 lines of code per man-day
High	8 to 92 lines of code per man-day

Menard (18) claims that the industry standard is between 7,5 and 15 lines of code per man-da,. This seems no more convincing than any of the other levels of performance which are suggested. So wide a divergence cannot form a basis of programmer productivity measurement.

Because the use of lines of code as a productivity measure is suspect (see Section 9.3) perhaps standards of performance which are expressed in terms of other elements may be of more value. For example, the concept of measurement used at Boeing Computer Services of 1 module per hour, has appeal, but the formula needs tighter definition. What is a module? What is an hour?

It is concluded that there is no standard of performance which can be accepted industry-wide and this automatically excludes the possibility of across-organization comparisons, or the calculation of variances and efficiency ratios.

7.3 The problem of the influence of working conditions on Programmer Productivity

Intuitively one would expect working conditions to influence the actual performance of programming staff. However, it is difficult to measure the impact on programmer productivity of factors such as:-

- open-plan or individual offices;
- differences in management style;
- the enforcement of appropriate standards;
- different methods of application development;
- the method of utilizing personnel resources;
- the particular programming langauges used;
- implementation of quality control procedures;
- the use of program development and debugging aids;

etc.

Without being able to measure programmer performance first and then experimenting with different work environments, the effect of any of these factors on programmer productivity will remain speculation.

7.4 The Need for an expanded discussion

From the list of problems identified in Sections 7.1.2, and 7.2.4 there are two areas which I feel require further attention. The first is the differences in complexity from program to program, it is a point raised so often in discussions on programmer productivity. The second is the measurement and control of program quality, because of the direct influence of program quality on programmer performance (see Section 5.2.2.2.).

7.4.1. Program Complexity

Both topics to be covered in this section concern two ways of looking at program complexity and complexity as it is related to the whole system.

7.4.1.1 Two ways of assessing program complexity

I suggest that it is possible for program complexity to be regarded as both an input and an output factor. (See Chapter 3 for definitions of Input and Output Factors). If a competitor, developing the same system, would also have to come to terms with complexity, because it is part of the product no-matter who was involved in its development, than that complexity is an output factor.

However, if the program appears complex just because there are no programmers available with the necessary skills to do the job (compare Break-Thru Technology in Section 10.2.2.5.3) then that complexity is an input factor.

Program complexity is also an input factor if the complexity were introduced by the particular internal structure created by the programmer at program design stage. To ensure that standards of performance can be established, any input factors are ignored. (See Sections 8.1 and 8.2). It will be established in Section 7.4.2.4 that simplicity is one of the characteristics of a quality program. One of the criteria of simplicity is a structured program. If programmers have developed programs without adhering to the disciplines of structured programming, they could be introducing complexity into the programming task by their very programming style. This form of complexity is an input factor which can be identified by comparing the individual programmer's performance with an accepted standard (see Section 10.3).

7.4.1.2 Complexity, individual programs, and the whole system

It is expected that the differences in program complexity will influence the rate of programmer performance (compare the writing of a simple extract and print program with a complicated restart recovery program).

However, looking at the development of a whole system with a mix of more and less complex programs, the influence of the complexity factor is probably significantly reduced. Until the influence of program complexity on programmer performance can be measured accurately (see Chapter 11) the productivity of programmers based on individual programs, must be assessed with caution.

7.4.2 Program Quality

In this section some of the opinions in the industry regarding program quality are identified. After establishing that there appears to be a consensus that program quality is difficult to measure, an attempt will be made to define program quality and to identify some methods of measuring that quality.

7.4.2.1 Apparent consensus that the Measurement of Program Quality is difficult

During a private conversation in Johannesburg on 10 March 1981, Dr. Herbert Grosch made a single comment on this area of research:-

'That's a tough nut to crack!'.

This comment seems to reflect a widespread opinion that the measurement of program quality is difficult. The GUIDE report on programmer productivity states openly (concerning program quality):-