Discussed is the unit-of-measure situation in programming. An analysis of common units of measure for assessing program quality and programmer productivity reveals that some standard measures are intrinsically paradoxical. Lines of code per programmer-month and cost per defect are in this category. Presented here are attempts to go beyond such paradoxical units as these. Also discussed is the usefulness of separating quality measurements into measures of defect removal efficiency and defect prevention, and the usefulness of separating productivity measurements into work units and cost units.

Measuring programming quality and productivity

by T. C. Jones

Although it is not always appreciated, the great advances in chemistry, physics, and other scientific disciplines in the 19th and 20th centuries were preceded by advances in the measurement of physical attributes and the development of accurate measuring instruments in the 17th and 18th centuries. Indeed, it can almost be said that scientific progress of any kind is totally dependent on the ability to measure quantities precisely. Therefore, the work of men like Gabriel Daniel Fahrenheit, who made the first mercury thermometer in 1714; of John Harrison, who made the first practical chronometer in 1728, and many other measurement specialists, are the fundamental underpinnings of modern scientific achievements.

It is because of the vital significance of measurement to progress that so many common words and units of measure today are taken from the names of those who explored better ways of measuring new phenomena: Ampere, Celsius, Coulomb, Curie, Henry, Hertz, Joule, Ohm, and Watt were all researchers whose names have been applied to common units of measure, and there are others such as Faraday, Galvani, and Volta who have indirectly lent their names to measurement.

In the field of computer programming, the lack of precise and unambiguous units of measure for quality and productivity has been a source of considerable concern to programming managers throughout the industry. In 1972, there was established in the San Jose Programming Center of the IBM Corporation a study group to explore the interrelated topics of program quality, programmer productivity, and the units of measure that could clearly display trends in both areas.

One of the projects carried out by that group was a detailed analysis of common units of measure used to assess productivity and quality throughout the open literature of the entire industry. Some of the findings were surprising, and it soon became evident that a number of widespread units of measure were misleading and even paradoxical. For example, the unit *cost per defect* was discovered to yield the lowest values for the most defective programs, thus making obsolete technologies appear more favorable in some instances than modern ones. The unit *lines of code written per programmer-month* was found to consistently penalize high-level languages, and tended to favor programs written in Assembler language. These findings are of some importance to the industry, because they make it difficult to compare productivity and quality from program to program. In extreme cases, they can slow down the acceptance of new methods because the methods may—when measured—give the incorrect impression of being less effective than former techniques, even though the older approaches actually were more expensive. This paper attempts to describe the common units of measure, and point out the nature of the paradoxes that occasionally occur. When the paradoxes and measurement variables are clearly understood, it becomes possible to make reasonable assessments of programming quality and productivity, and to apply units of measure that behave in a predictable manner.

Counting lines of code

A fundamental problem of all measurement techniques involving computer programs is that of knowing exactly what is meant by the phrase "lines of code." This topic is discussed in References 1 and 2. The conclusions there and the one here are generally the same; namely, that programs consist of more than executable lines of code. Programs also contain commentary lines, data declarations, Job Control Language (JCL) in some cases, and macroinstructions. Some counting methods consider every statement to be a line, whereas other methods consider only a subset, such as executable lines and data declarations, in the counts of the program. Between the extremes of counting everything and counting only executable lines there may be more than a two-to-one variability for the same program.

This problem is not really too serious provided it is recognized. Counting methods become troublesome when productivity rates are being discussed without knowing the line-counting rules in effect. The convention used at one programming center calls for counting executable lines and data declarations, but not counting comments or JCL. Macroinstructions are counted once when expanded, and calls that invoke macroinstructions are counted once. There is no inference that this method is either better or worse than the other possibilities. The key is to state the counting rules when reporting on quality and productivity. Otherwise, there is no way of knowing what the results mean.

A more subtle problem occurs when counting lines of code for programs written in high-level languages. In PL/I, for example, a line might be everything that occurs between semicolons, or everything that is written on a single line of a coding pad. Here, too, it is important to state the conventions in effect for the data to be meaningful.

To avoid such variations associated with counting lines of source code, an alternative is to count bytes or object instructions. Although this method has much to recommend it and is often used successfully, it is not always easy to estimate the final, compiled size of programs written in high-level languages. Since compilers differ in efficiency, and since individual programming styles can interact with the compilers in astonishingly diverse ways,[3] it is incorrect to assume that X number of source code lines in a given high-level language compile into Y number of object code lines.

As an example of the difficulty of doing source-to-object code expansions consistently, a number of cost-estimating reference manuals recommend different ratios for expanding high-level source lines into object lines for a particular language. The lowest expansion is 1.6 object lines for each source line, and the greatest expansion is 6 object lines per source line. It is a simple matter to count object lines in completed programs, but it is not easy to estimate expansion factors before a program has been written, without knowing the programmers' styles and the compilers to be used.

Still another possibility is to accept the uncertainties of line counting and accumulate multiple counts of source lines, object lines, and bytes. Identify the compilers used. Then publish the line-counting conventions in effect for the programs being reported.

With any method, there are even more variables associated with counting lines of code than those so far discussed. For example, some programs are written in mixed languages. Other programs require scaffold code or throwaway code for testing and integrating the main program, yet that code does not become part of the final, delivered product. One might question the counting of such code. How should one count the changing of an existing program? Should only the new lines be counted, or should the base lines in the existing program also be counted?

In counting as in accounting, experience and practice require that whatever the counting method, it should be (1) documented and clearly understood by all who work on the program; and (2) the significant programming reality is the final version—its quality and cost—that is delivered to the users. The size of this final version, counted by whatever method is agreed upon, is the key to uniformly assessing productivity and defect rates.

Once delivered, of course, programs become candidates for changes and modifications, as defects are noted and the original requirements change. Here, too, the basic concept applies that what is important is the version of the program actually delivered after modifications.

As in building a house, in which the scaffolding is part of the cost, programs seem to be characterized by the same kind of thinking. For initial creation, the important aspect is the cost of the delivered program. Intermediate versions are significant only because they are part of the cost of producing the final program. After installation, the costs of program changes and extensions are important both as they occur, and because they are part of cumulative costs of the program. In other words, the size of the change and the cost of the change are important, as well as the size of the total program after the change and the cumulative cost of the program after the change. These concepts are discussed in detail later in this paper. Because of the uncertainties in counting lines of code, object lines, or bytes, one might come to believe that the entire issue is irrelevant. Programs, after all, are written to provide functions, and the number of lines of code it takes to supply a function is of much less importance than the cost of the function itself. In fact, it is not important that the function be supplied via a program. If microcode or an electronic circuit can supply the same function for a lower cost than programming, it might be preferable to measure *cost per function* rather than cost per line of code.

Although such an evaluative approach may become increasingly important, this paper does not explore these issues for several reasons. There is a great deal still to be learned about quality and productivity normalized against lines of code. We have not explored the limits of knowledge, and comparisons between different kinds of programs—with lines of code counted the same way for both—almost daily yield new insights and discoveries. It is premature to abandon this method, just when results are becoming encouraging. Also, to explore such things as cost per function, it is necessary to be able to define and count functions. At present, the methodology for doing this seems too uncertain, although some progress is visible.

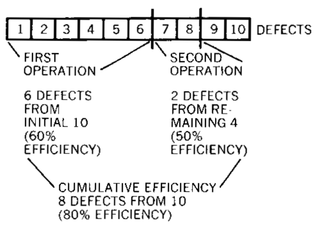
Measuring program quality

The term *quality* is used here to mean an absence of defects that would cause a program to behave unpredictably or stop successful execution. There are two fundamental ways of minimizing programming defects that significantly determine quality measurements. One is to prevent defects from occurring. The other is to remove defects that have occurred. Therefore, quality measurements are related to restricting the quantity of defects that come into existence, and their efficient removal by various kinds of reviews and tests used in programming.

a theory of defect removal

The pivotal concepts in the defect removal portion of quality measurement are those of *defect removal efficiency* and *cumulative defect removal efficiency*. Defect removal efficiency is the reduction of the defects that are present at the beginning of a defect removal operation by a certain percentage. Cumulative defect removal efficiency is the percentage of defects that have been removed by a series of removal operations, based on the number of defects that are present at the beginning of the series, or added while the series is in progress. The concept of cumulative defect removal efficiency is illustrated by Figure 1.

Figure 1 Cumulative defect removal efficiency


1 2 3 4 5 6 7 8 9 10
DEFECTS
FIRST OPERATION
SECOND OPERATION
6 DEFECTS FROM INITIAL 10 (60% EFFICIENCY)
2 DEFECTS FROM REMAINING 4 (50% EFFICIENCY)
CUMULATIVE EFFICIENCY 8 DEFECTS FROM 10 (80% EFFICIENCY)

The nature of cumulative defect removal efficiency is shown by the results of the second defect removal operation, and in combining the results of the first and second operations. Note that although the second removal operation has found two defects out of the total of ten in the hypothetical program, its efficiency is actually fifty percent (not twenty percent) because only four defects remain in the program at the time the operation is carried out. Since the sum of the effects of the two operations is eight defects out of ten removed, the cumulative efficiency is eighty percent. Even though the first operation had a sixty percent efficiency and the second operation had an efficiency of fifty percent, the cumulative efficiency clearly is not one hundred ten percent.

Bad fixes are another aspect of defect removal that require quantification when the efficiency of a given defect removal operation is being measured with accuracy. *Bad fix injection* is the introduction of a new defect, one not previously in the program, while repairing a defect in that program. It is useful to keep independent records of bad fixes so that separate statistics can be maintained for both detection efficiency and repair effectiveness.

estimating program defects

Given defect removal efficiency and cumulative defect removal efficiency, programming defect removal operations become more open to analysis and calibration. Indeed, it is largely through direct measurements of defect removal efficiency that improved forms of removal operations, such as design and code inspections[4] have come into existence.

Since defect removal efficiency statistics imply a knowledge of all, or almost all, defects found during the life of a program, the question arises of how such a theory of defect removal efficiency can be turned into a practical tool for everyday programming. We believe that this can be done as follows. As a program moves through the development cycle, records are kept of the quantities of defects found in all removal operations. Later, when the program has reached its intended users, records are also kept of the defects found in actual utilization of the program in its production environment. After several years, we sum all the defect records and find both the total quantity of discovered defects and the defect removal efficiencies of the series of reviews, inspections, and tests that were used to bring the program into existence.

We have been seeking trends based on experience that might prove to be useful in making predictions. Therefore, the procedure just given is not usually much help to the first few programs to be analyzed because too much time elapses before a sufficient quantity of data have been collected to carry out the analysis. Indeed, a program may already have been replaced or discarded before the data are sufficient. However, as more programs are measured and the results are analyzed, trends and problem areas become visible, and gradually significant improvements in cumulative defect removal efficiency become possible. Thus, real-time results may not always be possible, but long-term improvement in knowledge of removal efficiency eventually repays the effort. This is not unlike experience in other fields, such as medicine, in which the development of a cure for a disease is generally preceded by careful epidemiological studies on the causes of the disease, its outbreaks, the vectors that transmit it, and all other observations that relate to the disease.

Another problem in accumulating statistics of defect removal efficiency concerns major program changes between defect removal operations. A change in user requirements or a design change might take place between two defect removal operations. The result might be that the program that enters the second operation is quite different from the program that exited after the first operation. Our experience suggests two methods of dealing with this problem, a simpler method that tends to introduce distortions into the records (which may be tolerable), and a sophisticated method that preserves the accuracy of the data, but with added expense.

By the simpler method, cumulative defect removal is expressed by the following formula:

$$\text{Cumulative defect removal efficiency} = \frac{\text{defects found before release}}{\text{defects found before and after release}}$$

By this formula, if 100 defects are found in a program during its entire life—in both development and in production—and 90 of the defects are found before release, then the cumulative defect removal efficiency is considered to be 90 percent. We often find this coarse measure to be useful.

detailed method

The more sophisticated and detailed method requires the flagging of changes in the programming work products, including lines of code themselves. Although our goal is to arrive at the same formula as has just been described, the more detailed approach of flagging calls for analyzing the defect data and adjusting the quantities, depending on whether the problems were true defects or were caused by changing requirements. An adjunct of this method is that of analyzing the sources of programming defects. Given in Reference 2 are the following six causes of programming defects:

- Functional problems and the misunderstanding of user requests.
- Problems of logic and internal program design.
- Coding problems.
- Documentation problems.
- Incorrect repairs or bad fixes.
- Miscellaneous causes (a small category).

If a flagging system were utilized, and if records were kept against each work product during each defect removal operation, the resulting information could be displayed in a table that shows the contribution of each defect source to the overall total of defects, and the contribution of each defect removal operation toward the elimination of defects in each source. For simplicity, Table 1 gives an example display that is characteristic of the data we have collected, although this specific example is a hypothetical one. (All data are given in units of percent.)

Suppose that a program's defect causes fall into only two categories, A and B. Suppose also the program is known to have 100 total defects, of which 40 are caused by category-A problems and 60 are caused by category-B problems. Then it is clear that category A causes 40 percent of the program's initial problems, and category B causes 60 percent of the problems. Now suppose that the program is to be tested by a single defect removal operation that is known to have an efficiency of 50 percent against category-A defects and 70 percent against category-B defects. After the

Table 1 Sources of defects and defect removal efficiency by source

A. Percentages of defects by source

Functional design and misunderstandings	15
Logic design and misunderstandings	20
Coding problems	30
Documentation and others (not shown)	35

B. Percentage of defect removal efficiency by source

Activity	*Efficiency percentage against defects in*			*Percentage of incorrect repairs to defects*
	function	*logic*	*coding*	
Functional specification review	50	—	—	+1
Logic specification review	40	50	—	+2
Module logic inspection	60	70	—	+2
Module code inspection	65	75	70	+3
Unit test	10	10	25	+4
Function test	20	25	55	+5
Component test	15	20	65	+5
Subsystem test	15	15	55	+7
System test	10	10	40	+10
Cumulative Efficiency	*98*	*98*	*99*	
Net cumulative efficiency		*98*		

completion of the defect removal operation, the 40 category-A defects would have been reduced by 50 percent, so 20 undetected category-A defects are presumed to remain in the program for discovery by subsequent defect removal operations. The 60 category-B defects would have been reduced by 70 percent. Thus, 42 category-B defects would have been removed and 18 such defects would remain for subsequent removal. Since a total of 62 defects out of 100 have been removed, the cumulative detection efficiency against both sources is 62 percent in this example.

Of course, in operational situations it is necessary to adjust such elementary calculations as these to include bad fixes, to handle more than a single defect removal operation, and to recognize defects against more than two sources. In our experience, the necessary recordkeeping has been rather complicated, but the long-term value of the data and the insights that are gained have proved to be quite beneficial.

There is a great amount of record-keeping complexity associated with the more detailed flagging method of defect removal efficiency analysis. Therefore, the simpler method may prove to be useful initially, even considering the vagaries that this method introduces. These may be tolerable for crude calculations and rough analyses.

simpler method

To make the simpler method truly simple, however, it is necessary to normalize the data. Because programs vary widely in size and in other attributes that can cause defect quantities to fluctuate, it is not convenient to measure the raw quantities of defects alone. It is more practical to express defect levels in terms of some general unit, such as *defects per thousand lines of source code*. (Let the term "lines" be defined by local convention.) This method of normalization is useful for both quality and productivity measures, as is explained later in this paper.

In working with the simpler form of defect removal efficiency analysis, it is necessary to make the following two simplifying assumptions:

- All defects, regardless of source or of origin (whether design problems, coding problems, or some other) are lumped together and counted as the single variable, *defects*.
- The defect removal efficiencies of all reviews, inspections, tests, and other defect removal operations are lumped together and counted as the single variable, *cumulative defect removal efficiency*.

With these two simplifying assumptions and with the data for defect quantities in normalized form as defects per thousand lines of code, the results are both easy to work with and surprisingly powerful. Even though some imprecision is unavoidable, the value of this approach is that it breaks down the topic of quality into the two pivotal concepts of defect prevention and defect removal. The method also allows program data to be displayed as a matrix or table of data.

maintenance potential

The most basic display is that of total defects per thousand lines of code as one axis, and the cumulative defect removal efficiency as the other axis. Elements of the resulting matrix might be called the *maintenance potential* of a program. The maintenance potential of a program is the quantity of defects not found during defect removal operations. Undiscovered defects are sources of potential maintenance activity, if such defects occur during the actual use of a program. Table 2 is an example of such a matrix that is typical of values for selected ranges of the two variables.

Table 2 shows that if the total quantity of defects in a particular program ranges between 30 and 35 per thousand lines of code, and the cumulative defect removal efficiency of all reviews, inspections, and tests ranges between 90 and 95 percent, then the maintenance potential or quantity of undiscovered defects that might cause maintenance changes ranges between 1.5 and 3.5 potential maintenance problems per thousand lines of code. For example, the best case in this situation is 30 defects per thousand

Table 2 Maintenance potential* or undiscovered defects as a function of cumulative defect removal efficiency and initial total defects per thousand lines of code

Total defects per thousand lines of code	*Cumulative defect removal efficiency percentage*					
	90	*91*	*92*	*93*	*94*	*95*
35	3.5	3.15	2.8	2.45	2.1	1.75
34	3.4	3.06	2.72	2.38	2.04	1.7
33	3.3	2.97	2.64	2.31	1.98	1.65
32	3.2	2.88	2.56	2.24	1.92	1.6
31	3.1	2.79	2.48	2.17	1.86	1.55
30	3.0	2.7	2.4	2.1	1.8	1.5

*Maintenance potential = total defects − defect removal efficiency

lines of code, 95 percent cumulative defect removal efficiency, for a potential maintenance load of 1.5 problems per thousand lines. This is shown in the lower right corner of Table 2.

As has been mentioned, a whole family of interesting and useful data displays can be constructed from the basic concepts already presented, when augmented by other kinds of programming data. For example, suppose there is uncertainty regarding how many lines of code must be written for a new program. All that is known is that the quantity probably falls somewhere between 15 and 20 000 lines. In such a case, it is possible to link a series of graphs together to display all key variables. For example, the best case for the program just cited in connection with Table 2 consists of the smallest quantity of lines, the lowest number of defects, and the highest defect removal efficiency. That means 15 000 lines multiplied by 30 defects per thousand lines, for a lifetime potential of 450 defects. If the cumulative defect removal efficiency is 95 percent for the program in question one would calculate a potential of 1.5 defects per thousand lines, or 23 defects in all.

The worst case for the program is 20 thousand lines multiplied by 35 defects per thousand lines, for a lifetime potential of 700 defects. If the cumulative defect removal efficiency is 90 percent in this case, then the program is estimated to contain 3.5 defects per thousand lines, or 70 defects in all.

probability rectangle

The general form for displaying the linkage of ranges of variables together is what I call a *probability rectangle* because it bounds the probable ranges within which the program is to be developed. A series of such rectangles, each based on at least one variable from a previous rectangle in the series, is what I call *linked probability rectangles.*

Table 3 Maintenance potential as a function of hypothetical program size range

Range in undetected defects per thousand lines	*Program size range (thousands of lines)* 15	20
3.5	52.5	70
1.5	18	30

Such rectangles take the form of a matrix, with the base of the rectangle indicating the range of one variable, and the height indicating the range of another variable. The elements of the matrix indicate the interaction of the two variables. The size of a matrix (or number of elements) depends on the ranges of the variables and the granularity—degree of coarseness—with which the data are displayed.

Table 3 illustrates a probability rectangle for the number of maintenance changes that might be expected in the hypothetical program. Here the size range is 15 000 to 20 000 lines and defects range from 30 to 35 per thousand lines. The cumulative defect removal efficiency ranges from 90 to 95 percent. Table 3 is the simplest form of the probability rectangle and shows only the extreme ends of the ranges, with no intervening values. In reality, more information is usually displayed. We have used this simple form to clarify the principle.

Note in this probability rectangle that although none of the intermediate variables used in its construction varies enormously, the difference between the best and worst case is quite large. Indeed, the best case is 18 potential problems (or defects remaining to be fixed) and the worst case is 70 defects, or 3.88 times the potential problems of the best case. When the variables that affect a program's defect rate are separated and analyzed independently, and then recombined, it becomes evident that small changes yield large results.

It is often said that quality cannot be tested into a program. The combined impact of improving the defect removal efficiency by even a few percentage points, if coupled with reducing the quantity of defects by another few percentage points, yields a large reduction of defects in the delivered program.

Before discussing productivity measurements, it is well to observe that the term *quality* has been used here to mean an absence of defects. Of course, there are many other attributes associated

Table 2 Maintenance potential* or undiscovered defects as a function of cumulative defect removal efficiency and initial total defects per thousand lines of code

Total defects per thousand lines of code	*Cumulative defect removal efficiency percentage*					
	90	*91*	*92*	*93*	*94*	*95*
35	3.5	3.15	2.8	2.45	2.1	1.75
34	3.4	3.06	2.72	2.38	2.04	1.7
33	3.3	2.97	2.64	2.31	1.98	1.65
32	3.2	2.88	2.56	2.24	1.92	1.6
31	3.1	2.79	2.48	2.17	1.86	1.55
30	3.0	2.7	2.4	2.1	1.8	1.5

*Maintenance potential = total defects − defect removal efficiency

lines of code, 95 percent cumulative defect removal efficiency, for a potential maintenance load of 1.5 problems per thousand lines. This is shown in the lower right corner of Table 2.

As has been mentioned, a whole family of interesting and useful data displays can be constructed from the basic concepts already presented, when augmented by other kinds of programming data. For example, suppose there is uncertainty regarding how many lines of code must be written for a new program. All that is known is that the quantity probably falls somewhere between 15 and 20 000 lines. In such a case, it is possible to link a series of graphs together to display all key variables. For example, the best case for the program just cited in connection with Table 2 consists of the smallest quantity of lines, the lowest number of defects, and the highest defect removal efficiency. That means 15 000 lines multiplied by 30 defects per thousand lines, for a lifetime potential of 450 defects. If the cumulative defect removal efficiency is 95 percent for the program in question one would calculate a potential of 1.5 defects per thousand lines, or 23 defects in all.

The worst case for the program is 20 thousand lines multiplied by 35 defects per thousand lines, for a lifetime potential of 700 defects. If the cumulative defect removal efficiency is 90 percent in this case, then the program is estimated to contain 3.5 defects per thousand lines, or 70 defects in all.

probability rectangle

The general form for displaying the linkage of ranges of variables together is what I call a *probability rectangle* because it bounds the probable ranges within which the program is to be developed. A series of such rectangles, each based on at least one variable from a previous rectangle in the series, is what I call *linked probability rectangles*.

The units of measure of programming cost I have called *cost units*. These units concern the program itself, rather than the human activities that go into creating the program. Examples of programming cost units include the following:

- Programmer-months of effort per thousand lines of code.
- CPU hours and connect hours per thousand lines of code.
- Dollars expended per thousand lines of code.
- Cost per page for documentation.
- Cost per defect for maintenance.

Here too there are variations, such as hours per line instead of months per thousand lines. Also bytes may replace lines in a definition. The general concept is the same, however, to normalize by the product rather than by the work of creating the product.

Lines of code per programmer-month

Both work units and cost units are needed in evaluating programming productivity. A basic difference between the two units is that each is the reciprocal of the other. A misunderstanding of this difference has sometimes led to one of the problems with lines of code per programmer-month. In such a case, for example, the work unit has mistakenly been pressed into service as a cost unit, where it has sometimes served unsuccessfully. As a general unit of measure, lines of code per programmer-month has a number of weaknesses to which industry-wide variations in reported programming productivity may be attributed.

Noted here are five problem areas that involve lines of code per programmer-month:

- Sensitivity to line-counting variations.
- Ineffectiveness for noncoding tasks.
- Tendency to penalize high-level language programs in favor of programs writen in Assembler language.
- Arithmetic awkwardness in accounting for subtasks.
- Attention focusing on the act of coding, which is a misdirection, since the coding of a program is but a small part of the total effort required.

line counting

Line counting variations have been discussed earlier in this paper and in Reference 1. We merely add that they can lead to perhaps a two-to-one variation in apparent productivity, depending on the line counting method used.

noncoding tasks

The problem of ineffectiveness in measuring noncoding tasks is summarized here from a fuller discussion in Reference 6. The complete job of developing a computer program requires more

Table 4 The paradox of lines of code per programmer-month

Activity	*Assembler program*	*High-level program*
Design	4 weeks	4 weeks
Coding	4 weeks	2 weeks
Testing	4 weeks	2 weeks
Documentation	2 weeks	2 weeks
Management/support	2 weeks	2 weeks
Total effort	16 weeks (4 months)	12 weeks (3 months)
Lines of source code	2000	500
Lines of source code per programmer-month	500	167

than coding activities and these activities must also be measured. Therefore, when lines of code per programmer-month is used on noncoding tasks, the results are apt to be questionable. Results may even approach being nonsensical, as illustrated by this scenario. With modern defect prevention and defect removal techniques in programming, it sometimes happens that no defects are discovered during testing because the program has no defects at the time the test is carried out. If testing is done by an independent group rather than by the programmers themselves this tends to introduce slack time into development. By normal program development practice, the programmer usually cannot be fully reassigned until testing is over, in case defects should be discovered. Since it is nonproductive, slack time does not contribute to lines of code per programmer-month. It is therefore inaccurate to say for example, that one's productivity is one thousand lines of code per month during testing when there is no coding, and much of the time is spent waiting for bugs that may never occur. It is reasonable to say that slack time has added one month to a project but it is not reasonable to say that slack has proceeded at a rate of one thousand lines of code per month.

high-level languages

The problem of penalizing high-level language programs has only recently been explored, and it has been found to be quite important. Many portions of a programming development project are language-independent, and take the same amount of time regardless of the programming language selected. Such things as understanding user requirements, writing specifications, writing test cases, and writing user documentation are not affected in any way by the programming language selected. We know that high-level languages require fewer source statements to program a given function than does Assembler language. But language-independent activities proceed at the same rate as in Assembler language programs, yet fewer lines of code are written in high-level language than in Assembler language. The result is an apparent productivity lowering for the whole development cycle with high-level languages, even though development costs have actually been reduced. This is one of the paradoxes of programming measurement.

Table 4 illustrates an apparent loss of productivity when a program is written in a high-level language instead of Assembler language. Note that the true cost for the high-level language version of the same program was actually lower. The paradox lies in the unit of measure itself. Lines of code per programmer-month often displays this paradox, if activities other than pure coding are included in the measurements.

Table 4 illustrates that although the high-level-language version of the program has actually required four weeks less time than the Assembler language version (both versions assumed to offer identical functions), the high-level language apparent productivity expressed in terms of lines of code per programmer-month is only about one-third as great as that of Assembler language.

Although the Assembler language version in the example in Table 4 has 2000 lines and the high-level-language version has 500 lines, this does not imply a general statement that one high-level language statement is equivalent to four Assembler-language statements. As was mentioned earlier in this paper, there is no reason to believe that any expansion factor for any high-level language can yield uniformly acceptable results. This is one of the reasons why it is important to define line-counting rules when discussing productivity rates. It is also one of the reasons why it is generally advisable to establish separate productivity targets for programs in each source language, and to use extreme caution in comparing productivity rates (for source lines, object lines, or bytes) from language to language.

subtasks

Another problem with lines of code per programmer-month is the cumbersome arithmetic it entails, when one tries to measure all parts of a programming development cycle. The point is illustrated by the following example. Suppose a program consisting of 1000 lines of source code has been developed. The development cycle consists of four separate activities, each of which has taken one month to complete and has yielded a total development expenditure of four programmer-months. The sum of four consecutive activities, each of which proceeded at a rate of 1000 lines of code per month, is not 4000 lines of code per month, but 250 lines of code per programmer-month. Although simple in this example, the concept is cumbersome if data for a number of programs are being analyzed, and each program is divided into a large number of subactivities.

coding

The fifth problem with lines of code per programmer-month is that it contributes to a mental set toward the coding, a task that

Table 5 Comparison of work units and cost units

Activity	*Raw time expended*	*Lines of code per programmer month*	*Programmer-months per thousand lines of source code*
Design	4 weeks	2000	0.5
Coding	4 weeks	2000	0.5
Testing	4 weeks	2000	0.5
Documentation	2 weeks	4000	0.25
Management/support	2 weeks	4000	0.25
Totals	16 weeks (4 months)	500	2.0
Lines of source code	2000		

is not always a major activity. The productivity measure of lines of code per programmer-month originated in the early days of programming, when writing a program was usually a one-person effort. This main activity may well have consisted of actual coding. Today, programs are often developed by teams of specialists, of which the coder is only one part. Further, in modular programming where programs are constructed from reusable modules, rather than being hand-coded, there may be no new coding to be measured.

Modern programming methods are moving rapidly in the direction of developing reusable modules that can be cataloged in a library, and then obtained from the library to create new programs with little or no additional coding. The trend of attention is now away from work units and toward cost units, as is discussed in this paper.

other work units

As has been previously mentioned, there are work units other than lines of code per programmer-month. The most common way of estimating and measuring machine time during programming projects is that of CPU hours and/or connect hours per programmer-month. This unit, however, shares the vagaries of other work units, and tends to fluctuate widely from person to person and program to program. Experience leads to the conclusion that it is wise to discard the work unit form of machine time measurement. A preferable measure is CPU hours per thousand lines, a topic that is discussed later in the paper.

Another work unit of questionable reliability is that of pages written per writer-month for documentation and publications. To be useful, it is obviously necessary to define the page, and even then the results tend to be erratic and of marginal utility. Here also the cost unit form, which might be expressed as documentation cost per thousand lines of code, seems more reliable as a way of gaining understanding about this important area.

Table 6 Work unit comparison of past experience and improved programming technologies

Activity	*Past experience*	*Improved programming technologies*
User analysis and requirements statement	3 programmer-weeks	3 programmer-weeks
Design	20 pages per week	20 pages per week
Coding	60 lines per day	110 lines per day
Testing/debugging	10 tests per day	20 tests per day
Documentation	5 pages per day	6 pages per day
Maintenance	4 hours per change	6 hours per change

Programming cost units

Of the several programming cost units mentioned—cost per byte, cost per line, cost per thousand lines, and others—from my experience, cost per thousand lines of code serves best. Here, lines of code means source lines of executable instructions and data declarations, but not commentary lines. Source lines are natural units for most managers and programmers, and selecting a thousand lines or bytes helps to visualize a realistic development cycle. Of course, for programs smaller than a thousand lines some other unit, such as a hundred lines, might be preferable.

The advantage of cost units as opposed to work units is that all development and maintenance expenses, including manpower, machine time, and dollars can be expressed in terms of this basic unit, and can be used to derive complete project costs by summing the subactivity costs. The summing of cost units is simpler than summing work units, and is one of the reasons why cost units are more useful and versatile than work units.

Table 5 illustrates the differences between a work unit (lines of code per programmer-month) and a cost unit (programmer-months of effort per thousand lines). The example is taken from the 2000 line Assembler language program shown in Table 4. In this example, the work unit data under lines of code per programmer-month do not add up directly. The net productivity at the end must be calculated by dividing 2000 lines of source code by the four months of effort. The data under programmer-months per thousand lines of source code can be added directly, and lead to a cost/value analysis that is discussed later in this paper.

example

Tables 6 and 7 give a hypothetical example comparison of work versus cost units wherein the data are typical of those found in the literature. Assume that a company is debating the merits of various improved programming technologies, and wishes to know whether they are cost justified. Suppose also that an experimental program is developed for comparison, using improved programming methods. Expenditure of personnel time is to be compared

Table 7 Cost unit* comparison of past experience and improved programming technologies

Activity	*Past experience*	*Improved programming technologies*
User analysis and requirements statement	0.24	0.24
Design	0.72	1.08
Coding	1.5	0.66
Testing/debugging	1.5	0.72
Documentation	0.48	0.36
Maintenance	1.56	0.72
Total	6.00	3.78

*Programmer-months per thousand lines of source code

to the experience of several past programming projects. On the basis of the information in Table 6, it is difficult to compare the two programs definitively because each method has advantages and disadvantages. Compared on the basis of programmer-months per thousand lines, as shown in Table 7, the cost advantage of the improved programming technologies stands out clearly.

productivity analysis using probability rectangles

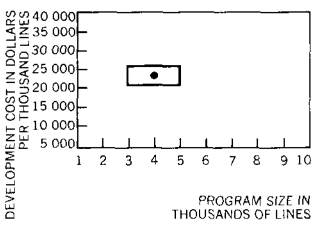
By using a cost unit, a series of useful productivity analyses can be made. To visualize these analyses, the probability rectangle approach, discussed earlier in this paper, is used again. In the particular probability rectangles used here, the cost units are dollars spent per thousand lines. Other measures, however, such as CPU hours per thousand lines or programmer-months per thousand lines, are equally possible and useful.

Programs have two attributes that lend themselves to a display of their cost of productivity ranges. They have size, which can be displayed in units such as thousands of lines of code. They also have costs and expenditures that can be expressed in such terms as dollars, programmer-months, or CPU hours.

The fundamental units of size and cost make possible the plotting of those parameters and the comparison of programs. Such data plots also highlight major uncertainties and the ranges of those uncertainties that confront a programming manager and cost estimator. Typical of the factors that such a person must estimate are the number of lines of code to be produced and the unit cost per line or per thousand lines.

In the following example, a company plans to develop a new program, the size of which is estimated to fall between three and five thousand lines of code. Previous unit costs for programs at the company have ranged between $20 000 and $25 000 per thousand lines (a typical cost range). In Figure 2, these estimates are plotted as a probability rectangle. The best case for the program being estimated is 3000 lines of code produced at a unit cost of $20 000 per thousand, yielding a total expenditure of $60 000, i.e., the lower left corner of the rectangle. The worst case is 5000 lines of code produced at a unit cost of $25 000 per thousand yielding a total expenditure of $125 000, i.e., the upper right corner of the rectangle. The center point of the rectangle is the mean of both variables. This point indicates that an expected size of 4000 lines has been produced at a unit cost of $22 500 per thousand, thus yielding a total expenditure of $90 000.

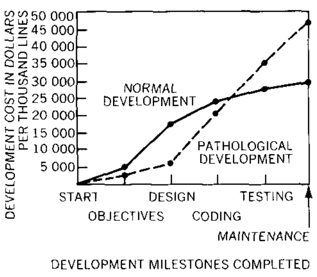
Figure 2 Probability rectangle for development cost and program size



While the programming project is under way, both the size of a program and the unit cost of a program tend to fluctuate independently. Therefore, it is helpful to be able to separate these variables, so they can be analyzed independently. Such information aids in business decisions about whether the project is worthwhile and should be continued. Such an analysis also provides feedback about potentially dangerous situations before they become pathological and cannot be corrected. The probability rectangle approach provides management with the expected boundary conditions of program size and program costs, and facilitates making decisions about whether to continue a project in the event that worst-case situations occur.

A probability rectangle analysis also aids in heading off what we term *pathological* programming situations. Generally, a pathological program is one where unit costs and/or size far exceeded worst-case expectations. Figure 3 illustrates the contrasts between a normal development and maintenance expenditure pattern and a pathological one. The curves plotted here are not a probability rectangle, but ones that have been derived from the concept of normalizing data to display various costs on a per-thousand-lines basis. This graph is one of the family of several possible data displays that use cost units and normalization.

Figure 3 Normal and pathological program development



In normal development, early expenditures are usually high because of the tooling up and necessary learning that accompany requirements, specifications, functional definition, and design. This spending pattern typically levels off during coding, testing, and maintenance. On the other hand, pathological development is often characterized by hasty requirements analysis, incomplete design, and the premature start of coding. The discovery of overlooked functional needs frequently triggers the rewriting and recompiling of much of the code. Such programs may be termed *rear loaded*, as illustrated in Figure 3 by low initial expenditures and by steeply increasing costs late in the project. One of the values of data normalization and cost units is that such patterns can be seen as they are developing, and corrective action can be taken.

Measuring productivity in a complex environment

We now explore more realistic program development situations that involve complex programs in which development and maintenance are intermixed at the same time. 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.

The concepts of normalization and programming cost units are useful in describing complex and realistic changing situations, as well as hypothetical cases. To do so, however, it is necessary to measure or evaluate the following costs:

- Costs of changing a program as circumstances change.
- Cumulative costs of program ownership.

Assume a programming system that has been developed and put into production status. Its initial size was 50 000 lines of code, developed at a unit cost of $40 000 per thousand lines. Thereafter, major changes were made to the system that added or deleted lines of code. Table 8 summarizes events in the life cycle of this hypothetical programming system.

The basic programming system entered production status with a unit cost of $40 000 per thousand lines, or $2 000 000 in total costs. Later there were two additions and one deletion. Although the additions and deletions were presumably made at different times for different reasons, and had varying unit costs, the cumulative cost of ownership always increases. Furthermore, after the additions and deletions, the unit cost for the whole system had risen from $40 000 per thousand lines at its initial completion to $50 000 per thousand lines after the third change. Although it is possible for the unit cost to decrease (such as when many lines are added for a very low cost) the general trend is usually upward with time, and the cumulative cost of ownership is always upward.

The main goals of productivity improvement are to lower the unit cost for development and the cumulative cost of ownership during the entire life of the program. It is important to be clear about these goals because technologies and strategies that tend to minimize unit costs and ownership costs are not always the same as those that lead to the most rapid coding or hand crafting of programs. For example, if a program were to be developed and there were a choice between writing the program from scratch or modifying an existing program, the following things might occur. Assume the new program to be 5000 lines of code in size and could be produced at a work-unit rate of 500 lines of code per month, or 10 programmer-months in all.

Table 8 Life cycle of a hypothetical programming system

Event	*Size in thousands of lines*	*Cost in dollars per thousand lines of code*	*Cumulative cost in dollars*
Creation	50	40,000	2,000,000
Addition	10	50,000	500,000
Deletion	− 5	40,000	200,000
Addition	5	60,000	300,000
Subtotal	60	50,000	3,000,000

To offer the same set of functions via modification might require 2400 lines of new code added to a base of 2600 lines of code borrowed from an existing program. Because of the difficulty of understanding or learning the base, productivity on writing the 2400 lines might drop to only 300 lines of code per month, or 8 months in all. Yet regardless of the apparent productivity rates, the costs are lower via modification. That is, if the delivered versions of the equivalent programs are contrasted in cost units, then the new program would require 2 programmer-months per thousand lines of code, and the modified version would require only 1.6 programmer-months per thousand lines.

This example illustrates the observation that on the average, productivity rates on new programs decline as size increases—with small programs of less than 2000 lines of code often taking in the vicinity of 1 programmer month per thousand lines, and large systems of over 512 000 lines often taking 10 programmer-months per thousand lines or more. When the cost of maintaining or changing a program is measured, however, a reverse trend is noted. That is, the smaller the change, the larger the unit cost is likely to be. This is because it is necessary to understand the base program even to add or modify a single line, and the overhead of the learning curve exerts an enormous leverage on small changes. Additionally, it is often necessary to test the entire program and perhaps recompile much of it, even though only a single line has been modified. This subject is discussed in somewhat more detail in Reference 2.

The cost saving that is often associated with reusing code that has already been written, rather than hand crafting it, is one of the main economic incentives leading to an increasing interest in modular programming and reusable module structures. It is in analyzing the potential cost saving that cost units as a means of comparison are showing their value.

If a program is being created from a library of precoded functions, the unit of lines of code per programmer-month has no meaning, since the work of the programmer has changed. Still, there are costs associated with assembling the products. Measuring with cost units leads to speculation about new ways of doing business, and about productivity gains similar to those in engineering and manufacture through the use of interchangeable parts.

Reusable code may significantly change one's perception of productivity. If, for example, one is developing a program function that is expected to be cataloged for reuse in many future programs, it might be well to invest in exhaustive testing, so as to approach zero program defects.

Problems of cost units

Although my experience indicates that cost units are more useful than work units in measuring programs at the present time, there are problems with cost units. Discussed here are limitations of two cost units, cost per defect for maintenance repairs and cost per page for publications and documentation.

In the context of programming, both units are in fact peripheral to the main concept of what a program is. With respect to programming, cost units aim at the product itself—lines of code or bytes. Thus cost per defect is a supplemental unit; the real indicator and true cost unit is defect removal cost per thousand lines or, alternatively, defect removal cost per line.

Similarly for documentation and publications, cost per page is a reasonable unit in a localized sense. However, it is preferable to measure documentation costs per thousand lines or documentation cost per line, so that these costs can be added to the other subactivity costs.

cost per defect

Of the two units of measure, cost per defect is likely to cause the greater misunderstanding. Cost per defect is a key unit because, as mentioned in Reference 2, about half the money ever spent on programming has been used for defect removal and repair. As it is commonly measured and used, cost per defect is one of the paradoxical units of measure, and tends to penalize high-quality programs because it often assumes its lowest values for the most defective programs. High-quality programs tend to be relatively free of simple defects, which are cheap to repair, and only have a residue of rather elusive problems. Also, cost per defect is a compound unit of measure, and one should understand both parts of the compound. All defect removal operations, such as testing, have two distinct expense elements. One element is preparation, which includes writing test cases, reading specifications, and many other activities. Preparation costs accrue whether a program has any defects in it or not, and these costs increase more or less as a function of program size. The other expense element is repair, which includes fixing bugs that are found and retesting after repairing the defects. Suppose, for example, that two similar programs are being tested, and we are interested in comparing their defect removal costs in some normalized form. Assume that both programs consist of one thousand lines of Assembler code, but one program has been written using improved programming methods, such as topdown design and structured code, for defect prevention. The other program, however, has been using older methodologies. Assume also that in testing only one problem is found in the modern program, whereas the old style program has ten problems reported. Preparation costs for the test are identical for both programs and run ten hours each. Defect repair costs for the modern program are only six hours, but total thirty hours for the old-style program. By adding the preparation and repair hours and dividing by the number of defects in each case the cost per defect is sixteen hours for the single modern program defect and four hours for the old-style program. The paradox lies in the observation that the greater the number of defects found in the program, the cheaper they are to repair. It might be thought that by separating the preparation costs from the repair costs the paradox would be resolved. This, however, is not the case. The low-defect program shows six hours per defect for repair alone, whereas the high-defect program requires only three hours per defect in repair costs.

The overall conclusion is that cost per defect is not a reliable unit of measure, since it penalizes high-quality programs. A better method is to look at defect removal and repair costs per thousand lines. With this unit, the true expenses of high-defect levels are revealed, i.e., sixteen hours of test cost per thousand lines of code for high quality programs, and forty hours of test cost per thousand lines of code for old-style programs.

cost per page

The situation with cost per page of documentation is not quite as traumatic as it is with cost per defect, since page costs do not tend to favor high-defect work products. The problem with cost per page is that it tends to achieve its lowest values for pages with the greatest amount of white space. If white space is held constant, cost per page tends to be lower for documents with the greatest number of pages (although this latter point is not a definite rule).

The problems with cost per page can be shown by the following example. Suppose that two identical programs are being documented, and both are one thousand lines of code in size. In one case, the writer merely converts a specification into a publication, and produces a fifty-page document at a cost of $3000. This yields a cost per page of $60. In the second case, the writer works hard to condense the materials, and produces a thirty-page document at a cost of $2400, or $80 per page. Even though the cost per page favors the large document, the smaller publication is the less expensive of the two. If documentation costs per thousand lines of code is the unit of measure, this fact is clearly revealed. The small book costs $2400 per thousand lines of code, whereas the larger costs $3000 per thousand lines of code. With documentation as with programming, care must be used in selecting units of measure for the results to be truly meaningful.

ratios and percentages

Of all the ways to discuss productivity data, ratios and percentages tend to be the least reliable and the most likely to cause serious misunderstandings. Ratios show, for example, percentages of time, expenses, or CPU hours devoted to different aspects of development. It is extremely common—perhaps more common than any other method—to see reports that indicate such things as "design took twenty percent of the time and fifteen percent of the programmer-months while coding took thirty percent of the time and forty percent of the programmer-months."

The fundamental problem with ratios and percentages is that they assume that various development activities are connected in such a way that if you know one of the activities, you can derive the others. For example, there is an assumption (implicit in the use of ratios) that if you can estimate coding costs accurately, then you can derive testing costs by assuming that testing is some percentage of the coding cost. These basic assumptions are incorrect, and there are no known fundamental ratios between the various activities of programming. Consider the two activities of coding and testing. Coding expenses are a function of the completeness of the design, the skills of the coders, and the tools and methods used. Testing expenses are a function of preparation costs and defect repair costs. It is possible—in fact quite common—for two programs A and B to have virtually identical coding expenses, but very different testing expenses. The assumption that a ratio of coding costs to testing costs developed for program A will work for program B is a common misconception, and one of the key sources of estimating error.

The alternative to ratios and percentages is straightforward. Calculate the costs of each development activity on its own merits and then sum all the subactivity costs to arrive at the total programming cost. This way, even if one activity is grossly incorrect, the problem does not propagate itself throughout other activities, which might be the case if ratios had been used.

Without multiplying examples, it may easily be seen that ratios are extremely simplistic, and supply little or no useful information. Indeed, the only thing that ratios do well is preserve secret or proprietary information about how much time or money were actually spent.

Summary and conclusions

To a large extent, the units used to measure program quality and productivity tend to lead the mind along certain channels of thought. An analysis of the commonly used units of measure in programming has revealed deficiencies in some units that lead to incorrect and even to paradoxical conclusions.

This analysis has shown directions in which further progress can be made in understanding both programming itself and ways of measuring it. We have shown that by subdividing the general topic of quality into the subcategories of defect prevention and defect removal for separate analysis, great insights into programming productivity may result.

In the general area of productivity it has been shown that it is useful to distinguish between work units (which try to assess how fast programs are developed) and cost units (which try to assess how much will be spent). My experience so far indicates that this distinction is important, because techniques that reduce costs are sometimes quite different from techniques that increase speed.

Even though great progress is being made in programming and in measuring programming, it cannot yet be said that programming has fully reached the level of an exact science. In spite of this, however, the results are increasingly encouraging.

CITED REFERENCES

1. J. R. Johnson, "A working measure of productivity," *Datamation* **23,** No. 2, 106–112 (February 1977).
2. T. C. Jones, *Program Quality and Programmer Productivity*, TR02.764, IBM Corporation, General Products Division, 5600 Cottle Road, San Jose, California 95193.
3. S. L. de Freitas and P. J. Lavelle, "A method for the time analysis of programs," this issue.
4. M. E. Fagan, "Design and code inspections to reduce errors in program development," *IBM Systems Journal* **15,** No. 3, 182–211 (1976).
5. C. E. Walston and C. P. Felix, "A method of programming measurement and estimation," *IBM Systems Journal* **16,** No. 1, 54–73 (1977).
6. T. C. Jones, "Productivity measurements," *Proceedings of GUIDE 44*, San Francisco, California (May 1977).

GENERAL REFERENCES

1. B. W. Boehm, "Software and its impact: a quantitative assessment," *Datamation* **19,** No. 5, 48–59 (May 1973).
2. T. Gilb, *Software Metrics*, Winthrop Computer Systems Series, Winthrop Publishing Co., Englewood, NJ (1976).
3. M. H. Halstead, *Elements of Software Science*, Elsevier North-Holland Incorporated, New York, NY (1977).

Reprint Order No. G321-5063.