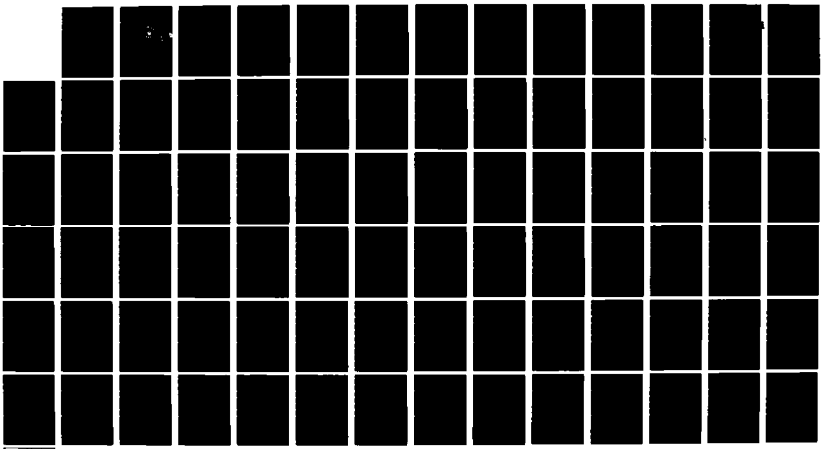
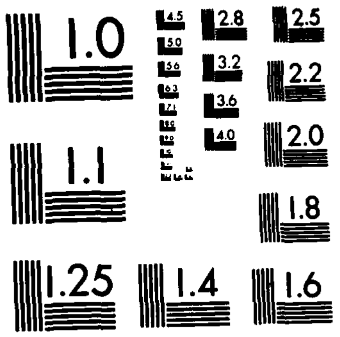
AD-A126 665   A STUDY OF QUANTITATIVE MEASUREMENTS OF PROGRAMMER   1/1
PRODUCTIVITY FOR FLEET MATERIAL SUPPORT OFFICE (FMSO)
(U) NAVAL POSTGRADUATE SCHOOL MONTEREY CA D J SPOONER
UNCLASSIFIED   DEC 82   F/G 9/2   NL



END


1.0
4.5
5.0
5.6
6.3
7.1
8.0
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.1
1.8
1.25
1.4
1.6

2

NAVAL POSTGRADUATE SCHOOL

Monterey, California

THESIS

A STUDY OF QUANTITATIVE MEASUREMENTS OF PROGRAMMER PRODUCTIVITY FOR FLEET MATERIAL SUPPORT OFFICE (FMSO)

by

Daniel John Spooner

December, 1982

Thesis Advisor: Dan C. Boger

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO. A126663	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) A Study of Quantitative Measurements of Programmer Productivity for Fleet Material Support Office (FMSO)		5. TYPE OF REPORT & PERIOD COVERED Master's Thesis December, 1982
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Daniel John Spooner		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940		12. REPORT DATE December, 1982
		13. NUMBER OF PAGES 82
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for Public Release; Distribution Unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Programmer Productivity, Software Development Productivity, Programmer Metrics

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The demand for software products has grown, but the number of quality programmers has not kept pace. Therefore, programmer productivity has become a major area of discussion throughout the software development industry. This paper examines the various measures discussed in the literature and used in selected corporations which develop software. It presents several methods for measuring programmer productivity. Included in the (Continued)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

ABSTRACT (Continued) Block # 20

discussion are the salient points where managers must devote special attention if they are to use programmer productivity measures effectively. This paper is part of a group of papers which together provide recommendations to the Fleet Material Support Office (FMSO) to enhance its software development organization.

A Study of Quantitative Measurements of Programmer Productivity for Fleet Material Support Office (FMSO)

by

Daniel John Spooner
Lieutenant, United States Navy
B.S., Pennsylvania State University, 1977

Submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN INFORMATION SYSTEMS

from the

NAVAL POSTGRADUATE SCHOOL
December 1982

Author: [illegible]

Approved by: Dan C. Boger
Thesis Advisor

Norman R Lyons
Second Reader

[illegible]
Chairman, Department of Administrative Sciences

W M Woods
Dean of Information and Policy Sciences

ABSTRACT

The demand for software products has grown, but the number of quality programmers has not kept pace. Therefore, programmer prodcutivity has become a major area of discussion throughout the software development industry. This paper examines the various measures discussed in the literature and used in selected corporations which develop software. It presents several methods for measuring programmer productivity. Included in the discussion are the salient points where managers must devote special attention if they are to use programmer productivity measures effectively. This paper is part of a group of papers which together provide recommendations to the Fleet Material Support Office (FMSO) to enhance its software development organization.

TABLE OF CONTENTS

LIST OF FIGURES

I. INTRODUCTION

In the past two decades, as computer hardware costs have fallen and software costs have risen, there has been an increasing interest in programmer productivity. This interest has become particularly intense during the last decade as the general purpose computer market has flourished. Customers are becoming much more aware of the flexibility that different software packages provide to computer hardware. They, therefore, are demanding more and more software products to upgrade existing hardware facilities. Withington [Ref. 1] of Arthur D. Little Inc., a Cambridge (Mass.) consulting firm, states that the throttling factor in the evolution of the data processing industry is the pace of software development. Revenues in the data processing industry are expected to reach $95 billion by 1984 but have the potential to reach $125 billion if the software development constraint did not exist. This software demand has precipitated a large demand for programmers. But programmers, especially skilled ones, are hard to find and take time to train. Since there has been such an astronomical growth in the computer software industry, finding sufficient numbers of well trained and experienced programmers is prohibitively difficult. [Ref. 2], According to Digital Equipment Corporation [Ref. 3], the biggest problem is identifying the few good programmers. Of the many applicants they receive, most are not capable of writing sophisicated software. Consequently, software developers are turning towards increasing the productivity of programmers in an attempt to keep pace with the demand for current and future software design needs.

There have been a number of papers written discussing productivity. Some discuss determinants of programming productivity [Ref. 2], others provide tools [Ref. 4], which purport to improve productivity. Interestingly, few of these studies discuss or make referance to others who have discussed how to actually measure this productivity. The philosophical approach for many years was that programming was an art. This made it virtually impossible to measure, for it would be similar to measuring the progress or productivity of a Picasso or Michelangelo as he was painting or sculpting. Obviously, there is no way to measure the progress of art aside from personal opinion. This, however, is not acceptable in an industry based on the profit motive. In the late 1960's the term "Software Engineering" was coined and with it came a number of ideas that served to pull programming out of the world of art and into the world of the engineer, a world where measurement is of vital importance. Software development was shown to be an area that required discipline and a process-oriented approach [Ref. 5].

Software engineering has grown through the 1970's to virtually become the rule for the management of programming. It has led to the development of new strategies for software development. These strategies, top-down design, bottom-up design, structured programming, modular decomposition and metaprogramming, have provided a better foundation from which software developers can attempt to meet the growing demand for software products. Although these development techniques have made software development easier and helped to control the cost growth, they have had little impact on productivity measurement.

To discuss the measuring of software development or programming productivity, one must first determine what the product is. From the first day of programming until the present, the predominant product of discussion has been the "line of code" (LOC). This is the product on which nearly all research and the database information are based. If one were a construction engineer one would not discuss a building or bridge based on the number of bricks and girders used. Instead, rooms or floors or spans might be much more appropriate. These items are integral but separately measureable components of the final product. So why, rhetorically, do researchers and data base information collectors continue to insist on LOC measures instead of an integral and separately measureable and meaningful component of software engineering? This not a question for this paper to answer but one for the reader to consider when planning his own research or data base collection.

The Fleet Material Support Office (FMSO) is experiencing the same problems as the rest of the software indusrty. It is faced with a huge demand for quality software from the organizations it is tasked to support. The tasking of the past five years is shown in Figure 1.1 below. These figures are only for the Central Design Agency, the primary mission of FMSO. The figures show an increase in FMSO maintained programs of 75.4 percent in this short period. These figures are expected to continue to rise at a significant rate as the Navy continues to automate more and more functions. Another problem facing FMSO is the salaries of the programmers. According to Business Week [Ref. 6] programmer salaries are rising at a rate of 15 percent annually and salaries for top systems analysts can reach $50,000 a year. This places an extreme burden on the personnel department to acquire top personnel when hiring new programmers and systems analysts. The productivity issue becomes increasingly critical for FMSO in the light of the hiring freeze imposed during the Carter administration and the drive to reduce the cost of government in the present Reagan administration.

```
                    CDA Program Growth
FY
77 XXXXXXXXX  5,389
78 XXXXXXXXXXXXX  6,420
79 XXXXXXXXXXXXXXXXXX  7,722
80 XXXXXXXXXXXXXXXXXXXX  7,938
81 XXXXXXXXXXXXXXXXXXXXXXXXXXX  9,030
82 XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX  9,454 (April)
```

Figure 1.1 FMSO Program Library Growth.

This paper attempts to present a number of issues related to the measuring of programmer productivity. It will show that there are a other factors that impact on how one interprets the productivity figures. The manager needs to realize there are several different levels of the organization, each with its own product or set of products. Therefore, each level has a productivity rating for which it must be responsible. In fact, the reader should note that the programmer is not the predominant link in the output of a programming project. The requirements of the Department of Defense and conscientious software developers throughout the industry has placed increasing importance on the reliabiblity and maintainability of software. This new emphasis has produced a whole array of corresponding products which must be accounted for and new productivity levels which must be examined.

II. WHOSE PRODUCT IS BEING MEASURED?

When discussing productivity, before one can consider who to measure, one must first determine what the product is and then who makes the product. Without a rational visualization of the product it is unintelligent to discuss the ability of a person's, group's or machine's ability to deliver that product. Depending upon the level of the organization at which one looks there will be a variety of goals, objectives and products. Both Kiser [Ref. 7, p. 244] and the IEEE Workshop on Software Productivity [Ref. 8] address this important issue.

Where the IEEE Workshop focused on the general area of productivity, Kiser was most concerned with software management productivity. She focused on the idea that the manager often has as much to do with a programmer's productivity as does the programmer himself or his tools. This is a nontrivial issue. She looked at the top three levels of

```
   corp. goals  <---> top mgmt    <---> corp.prod.
 product goals  <---> middle      <---> product
 project goals  <---> first line  <---> project
    task goals  <---> programmer  <---> task
```

Figure 2.1 Kiser: Levels of Note in Software Productivity.

management, shown in Figure 2.1 . Many managers have failed to understand why their people, being well-trained and provided with excellent tools, continue to produce at unsatisfactory levels. Quite often, from this researcher's experience and the experience provided by Kiser, the poor production level is caused by higher level managerial policies or actions. This can be understandable when one examines the concerns of the various management levels.

At the corporate level, top management is usually concerned with profit maximization and market share. FMSO, being part of the public sector, does not have this particular concern but there are comparable goals (Figure 2.2)

CENTRAL DESIGN AGENCY (CDA)
RETAIL NAVY STOCK FUND
OPERATIONS ANALYSIS
SUPPLY OPERATIONS SUPPORT
INTERNATIONAL LOGISTICS

Figure 2.2 FMSO Major Mission Areas.

which are fleet support and effective management of their approximate $3.8 billion, FY82, procurement authority. When one considers the impact of money management at this level it is understandable that concerns for individual programmer productivities can get lost. The interpretation of top level management polices by lower level managers can also affect productivity.

At the middle management level, managers become concerned with specific product development and resource allocation. For FMSO, in its primary mission area as a CDA, management is concerned with allocation of resources to

UNIFORM AUTOMATED DATA PROCESSING SYSTEMS (UADPS)
- Uniform ADP System for Inventory Control Points (UICP)
- UADPS Stock Points (UADPS-SP)
- Level II/III Stock Points
- Disk Oriented Supply System (DOSS)

HEADQUARTERS FINANCIAL SYSTEMS

MANAGEMENT INFORMATION SYSTEM FOR INTERNATIONAL LOGISTICS - (MISIL)

SPECIAL DATA PROCESSING SYSTEMS PROJECTS
- Requisition Material Monitoring and Expediting (RMM&E)
- Trident
- Naval Aviation Logistics Command Management Information System (NALCOMIS)
- Naval Automated Transportation Data System (NATDS)
- Naval Automated Transportaion Documantation System (NAVADS)
- Resolicitation

Figure 2.3 FMSO CDA Primary Product Areas.

respective product areas as shown in Figure 2.3 below. The allocation of the resources is tempered with the command goals and the budget provided by the various sponsors.

The first line level of management, project management, is where one first encounters the edge of software productivity, the area with which this paper is concerned. Here the project manager is concerned with meeting prescribed milestones within budget. The products at this level are the various "deliverables", such as functional specifications, conceptual designs, program design, test plans, etc., that are required in an effectively managed project with milestone requirements. These are the products one must measure against their respective costs.

At the line level itself there are two groups, project teams and the individuals who make up the teams. The team must be measured against its ability to deliver integrated software products. The individuals must be measured against their ability to deliver specific portions of the team assignment. This is the point where programmer productivity is discussed by most researchers. A special note is required at this point. While one usually assumes that the delivered products are of a specific quality, this seems to be missed quite often when discussing programmer products. The idea of quality in the product must always be considered. A person who can deliver five programs in one day that are incorrect or do not provide consistent results is not nearly as productive as one who delivers one product every five days but which is correct and easily maintained. Very few productivity measures take quality into consideration, as will be shown later.

After realizing the various products made by different levels in the organization, one must then consider who is viewing these measures, management or labor. The views and concerns of each are usually quite different unless there has been a considerable amount of education on each side.

Management must understand there is an overhead expense to developing, collecting and analyzing productivity measures which must be justified. Intuitively, one must have a set of measures before one can determine constant, "normal" or changing productivity. Also management needs to know how it intends to use these measures. The IEEE [Ref. 8, p. 341] sees four major uses for productivity measures: 1) motivation; 2) understanding; 3) evaluation; and 4) management.

Productivity measures can be used for motivational purposes in three ways which provide tangible benefit. First, researchers [Ref. 9,] have shown that by paying attention to a person or group, performance levels of that person or group will improve or change to what the observee perceives as expected performance. This is known as the Hawthorne Effect. When managers take the time to do productivity studies the Hawthorne Effect may occur, albeit temporarily. Second is the ability to focus attention on desired behaviors, events and objects or products. The measures selected will place relative importance on the areas being measured. For instance, if a series of measures are selected which include speed of production and maintainability the perceived relation between them by the programmer will determine which measure they emphasize. That perception of relative importance can have a profound effect on the final product. If programmers see speed being rewarded or emphasized more than maintainability, the manager should expect to see programs produced rapidly but which are hard to understand and have little documentation. If the reverse is perceived, then the manager should expect to see longer programming times with much easier to understand and better documented code. The third motivating factor occurs through feedback of results. The effective feedback of productivity measures can lead to changes in performance in several ways. Quite often performance will improve through the personal pride in accomplishment or competition with peers. Also if a corresponding and effective rewards and penalty system, either formal or informal, exists, performance normally will follow the system correspondingly.

Second, productivity measurements help managers to understand the factors underlying productivity. Measurement is fundamental to science in that it forces managers and researchers to conceptualize the area under study. Using various concepts will determine which measures to use as managers continue to try to model the environment in which they operate. Failure to develop a model will hinder managers in improving performance and will keep software development an art instead of a science.

Third, productivity measures help managers evaluate performance because they quantify performance. It is easier to evaluate performance over time within a single group or organization because the measures remain constant. It is also very important to track performance so that proper feedback to personnel can be provided. It is also important to evaluate between groups to see how one stands against an industry average. This has proven to be particularly difficult for software developers. Few groups use the same measures. Those that use similar sounding measures often have significantly different definitions for the individual parts of the measure. LOC, which will be discussed later, is a most common area of disagreement. Nevertheless, it is important for each organization to establish a baseline and to build a database of information. This information can then be used for measuring the evolution of methodologies and technologies used in software development.

Fourth, productivity measurement imposes a managerial discipline. Normally managers are concerned with tracking progress against a schedule and budget.The consistent use and taking of measurements can be extremely helpful in making projections of progress against schedules and budgets. The manager must remember that a productivity measure is only a snapshot. It must be analyzed in relation to its environment. In particular, managers must realize the difference in the learning curves of various projects. A "first-of-its-kind" project will have a much different learning curve than a simple modification to a generic project. The productivity rates will normally change proportionally to the learning curve.

The manager's need for measures and his goals can differ significantly from those of the workforce. Management often wants to use the measures to identify exceptional performers or those who need added training.

The workforce, however, may view the measures as a way to generate either more products from the same work effort or to generate the same number of products from a reduced workforce. When the workforce sees the second side there can be severe implications, particularly if they are organized.

The workforce will rapidly wonder what their benefits will be from all this new attention. Will the measures lead to more money for the same hours, the same money for less hours for the good performmers and/or lost jobs for the poorer ones? In an effort at job preservation, productivity may fall or stagnate at a predetermined level. This researcher has seen deliberate productivity stagnation by bricklayers, both in the housing and steel industries, and by electricians working for a telephone company, all at well below reasonable levels of capability. For one to think that programmers and their industry would not tend to act in a similar fashion is to approach this area with tunnel vision. This may become a primary concern for FMSO where some of their government employees hold specific GS ratings and incomes based on the number of personnel they manage. Command level management must take care in the introduction of the productivity metrics so that personnel in these GS ratings do not feel that their jobs or ratings are in jeopardy if there is significant increase in productivity which leads to a reduction in force (RIF).

III. WHAT IS THE PRODUCT?

This researcher has determined that the predominant measure of programmer productivity is the quantity of lines of code written. This leads to several interesting conclusions. First, the programmer only writes deliverable code. Second, the programmer is the single dominant entity in software development. And third, there are no other relevant products or by-products in a software development project. Anyone who has the opportunity to study or to work in the software development arena realizes the fallacy of these conclusions. Programmers do considerably more than write deliverable code. There are many other people involved, each adding important contributions to the project. There are several equally important products.

From the previous chapter it was noted that there are many levels of an organization whose productivity should be measured. Those involved in software development realize that various levels of the organization make contributions to the various products of each project. This chapter will look at the different products that this researcher feels are relevant to the measure of software development productivity. This discussion will begin with middle level management and work towards the individual. As we progress down the organization the product will become easier to grasp. The span of management control and resource responsibilities will decrease. Therefore, one must remember to ensure the product and the level of the organization match. All too often people are evaluated on their ability to produce a product which they were not assigned to produce nor had any role in producing.

Unfortunately the reader will find in this section several terms that have multiple meanings. This is inescapable because there has been no accepted set of standard definitions within the software development industry.

A. PROJECTS AS PRODUCTS

The "contracted project", generically, is a software development tasking for which an organization contracts another to produce. It may consist of a number of subprojects or programs. An example is the development of an operating system which includes a job scheduler, process scheduler and file manager, Figure 3.1 shows the various

contracted project	assigned project
milestones (1)	management/support (1)
design specifications	functional specifications
lines of code	modules
function (user)	function (computer)
test code	documentation

(1) not deliverable products

Figure 3.1 Software Development Products.

component products of a project. The project, an operating system, must integrate each of these various parts to be complete. Therefore, the question of productivity here is whether or not the project can be delivered <u>on budget and on schedule</u>.

If the contracted project is large, as in the operating system example, it will be broken down into several smaller projects, which I call "assigned projects" since there is little choice as to who will manage them once the contracted project is accepted. The assigned projects will be given to several project managers who will report to the central contracted project manager. The role of each of these project managers is to deliver a fully complete integrated operating product.

The question at this point is, "Are these good items by which to measure productivity?". Yes, they are, for several reasons. First, for <u>this level of management</u> they are the only products that are produced. Second, the reason for the manager to hold the particular job of project manager is for him/her to deliver a project on time, within budget and to the satisfaction of the customer so that the organization may make its profit. What about the difference in languages used or the sizes of various projects? These questions need to take their rightful place in the data base of information of the corporation. Each productivity measure has a set of parameters within which it can only be used. There is a definite need to know how capable a project manager is at: 1) developing any project; 2) using a specific language; 3) developing various sized projects; 4) developing machine dependent projects; 5) developing first-of-its-kind projects; or 6) modifying a generic project.

Each of these parameters gives added insight to a project manager's productivity rating. The first lets one know how productive he/she is relative to all the other project managers regardless of project specifics. Each of the other measures provide additional information on the relative productivity of a project manager within the different parameters. Use of all of these productivity ratings by the next higher level of management may improve both levels of management's productivity provided project managers are well matched to projects where their productivity is highest.

B. MILESTONES AND MANAGEMENT/SUPPORT

At this point it may be advantageous to discuss a management tool that many may consider to be or confuse with, a product. A "milestone" is a point in the life of a development project when a deliverable product, as listed in Figure 3.1 , should be completed. Many would think that the ability to meet project milestones shows great productivity. This is not true. For if it were true, first the milestone must, in fact, mean the production of a deliverable item. Second, the deliverable item must be something of value to the project. If the deliverable is, in fact, of significant value to the project then the production of that item is the basis for one's measure and not the meeting of a milestone. The meeting of the milestone shows only that the project is proceeding as planned. The milestone has no other inherent value. That is, one does not deliver a milestone as one would a program. The milestone is only another management tool just as is a productivity measure.

Like milestones, Management/Support is not a product but a management tool. However, the type, quality and quantity of the support must be considered very carefully. Management/support exacts a price in that it is an overhead expense. Its value is not as a product but as a tool. Nearly all presentations discussing productivity refer to the management/support tools. This is where the vendors and consultants make a great deal of money. They speak of productivity improvement and the aids that provide it.

There are two parts to this concept, management tools and support tools. The management side deals with systems that help predict costs and time schedules and those that track the progress against the predictions and plans. At FMSO, this function is under the auspices of the Management Department, Code 92 [Ref. 10] where PAC-II is used to track and DOD MICRO and SLIM are used to estimate software costs and time schedules. The value of this support can be very subjective. Often the value of the management aid is that it gives the manager much more confidence in his/her decisions. The effect of the use of these kinds of tools may also be seen on the ledger. If the systems help management, all else being equal, one would expect to see fewer cost overruns and better personnel management.

The support side has a miriad of tools that predict sure-fire ways to improve productivity dramatically. These tools include various design procedures (i.e. structured, top-down, modular design), on-line programming and provision for each programmer to have his/her own CRT terminal to mention a few. T.C. Jones [Ref. 11] discusses more of these tools and their respective limitations.

The fact that management/support is not a product does not minimize its importance. On the contrary, it is vital to effective software development. But the manager must realize that the addition of each piece of management/support costs money for which accounting must be made. Although, there are many management/support systems which may improve productivity, the indiscriminate implementation of their use will not necessarily lead to productivity improvements. The use and expansion of management/support is an area worthy of further study.

C. DESIGN AND FUNCTIONAL SPECIFICATIONS

Design specifications are usually thought of as a product of the contracting organization. They are used as the basis from which to make a contractual bid and to write the functional specifications. However, the design specifications, as delivered, often must be rewritten by the contractor in close conjunction with the contracting organization so that they are explicit enough to properly write the functional specifications.

Both Keider [Ref. 12] and Howden [Ref. 13] discuss the need for well thought out and well written design specifications. Keider's article, "Why Projects Fail", shows how poorly planned projects waste money and resources. Howden's article, "Life-Cycle Software Validation", discusses the need for project design specifications to meet five properties. First, the specifications must be consistent internally as well as in any related documents or other portions of the project. Second, the specifications must be complete. They must be examined for missing or incomplete information requirements and to ensure data properties are included. Third, the specifications should only include necessary items without redundancy (not to be confused with hardware redundancy to ensure reliability). Fourth, the system must be feasible with existing technology and hardware. And fifth, the specifications must use correct math formulas and decision tables.

The reader should recognize that the validation of design specifications and functional specifications is a nontrivial task. The systems analysts who validate the design specifications and who write and validate the functional specifications must be held accountable for their resource use in the production of these products. The specifications need to be examined carefully, as discussed above, especially when one considers that approximately forty percent of a projects resources are used in the design phase [Ref. 37]. Poor quality here is very difficult and costly to try to overcome later in the software development cycle.

D. LINES OF CODE AS A PRODUCT

The line-of-code (LOC) is, by far, the predominant measure used throughout industry to discuss program size and productivity ratings for all levels of software development. Interestingly, though the entire industry uses LOC as a measure of product definition, few agree as to what a LOC is. One of the first questions asked is, "Do you mean a line of object code or source code?". The industry has had some success in distinguishing between them but not in choosing one or the other as a universal measure. Source code is that written by the programmer while object code is the compiled code stored in memory. Source code is more often used to describe programmer productivity than object code which is usually used to define the quantity of computer memory required to store the program code [Ref. 14].

Assuming one has settled on source code as a part of the measure, what determines a line of code? Some have said each line or statement written by the programmer regardless of length. Others try to force the line to have eighty characters. Still others try to define it by statement punctuation characters by language (i.e. periods in COBOL or semicolons in PASCAL).

If this weren't bad enough, the next question is, "Which of the lines are 'countable'?". That is, some want to differentiate between executable statements, data declarations, comments, nondeliverable debugging or testing aids, etc. Use of LOC each of these areas must be explicitly defined because studies have shown line count variations of more than two-to-one on the same program [Ref. 15].

After the LOC is well defined and published, one must watch carefully because, just as the measure helps management to rate personnel, so does it help personnel to promote themselves, often by manipulating the rules in their favor. Here are several examples. One company settled on every line written regardless of length. After some examination of several programs, lines were found not to be complete statements nor eighty characters in length, thus padding the true productvity levels. Another may decide to use eighty characters as the defined line. In this case it would not be unusual to find variables with extremely long names or use of the "blank" character to fill up lines and thus pad the productivity rating. Paradoxically, the programmers may be forced to have large numbers of blank characters if management requires the use of structured programming techniques. Another problem is that programmers may fight the use of higher level languages so they may program in a language in which they are comfortable and which requires more lines to accomplish the same task. Jones [Ref. 15, p.41-43] discusses the LOC measure more extensively than presented here.

Since the measure is so difficult to define and may lead to unacceptable programming practices, as stated above, or cause paradoxical conclusions, as discussed in the following chapter, this researcher feels LOC is a poor product measure. However, this does not mean to say that there is no use for LOC as a product measure. In fact it is the only measure available when one is performing maintenance on programs which entails changing individual lines in a program. Therfore, we must have a definition for a LOC.

There are many different languages in which one can program. Since each has its own rules of construction the definition of a LOC will necessarily be different for each language. This researcher prefers to view a line of code in the context of a complete sentence or phrase of spoken language. Each programming language has a defined equivalent of a complete statement or phrase. Just as Hemingway and Faulkner had different styles of conveying information, so will programmers. This is not a detriment to programming any more than it is to writing. Programmers will settle into standard line lengths with which each is comfortable. As long as management is satisfied that the style fits well into the structure of the language then there should be no problem. This does require management to supervise and to train those that are not consistent in their own programming or are far from the "average" line length of the rest of the programmers.

The countable lines should be those that are vital to the program quality and specific language. The lines that are niceties but which aid in the readability of programs have good reason to be in programs. They should be counted but not with full credit. The comment line is an example. It is necessary for readability but a one hundred line program does not need an additional hundred lines of comments. Contrarty to others, this researcher believes some credit should be given for comment lines. However, to keep verbosity out of programs due to comment lines and to be consistent with the credit given for reused code [Ref. 16], they should only count as twenty percent and then should be a full eighty characters long. Lines that are executable or data declarations and the like should be counted fully as one line.

If LOC is used as a measure for program length, it should be measured as a block of LOC, haing at least one hundred lines and not mire than one thousand lines per block. There are two reasons to do this. First, each block of LOC can have a time value association. This allows developers to speak in terms of time per block of code. This is valuable when trying to estimate the time required to develop a program estimated to be some number of blocks of code long. Second, code must have an intrinsic quality. It makes little sense to discuss one tested, debugged and documented LOC. But it does make sense to discuss a block of code with the same qualities. This tends to force the code to have some minimum level of quality. The quality requirement takes into consideration the time spent by the programmer in writing non-delivered test code and debugging aids and in correcting logic errors. When LOC are reused the count value should be a percentage of one original LOC. Basili and Freberger [Ref. 16] use twenty percent in their research. This researcher recommends starting with twenty percent and then adjusting it according to the percentage of time required to locate reusable code instead of developing original code.

E. MODULE AS PRODUCTS

A module is a single, intellectually managable portion of a program which is separately compilable but which must have connections to other modules. Its size is variable but it contains only one complete responsibility assignment of a program. It has only one entry point and one exit point and conforms to the permitted logic structures of structured programming. The responsibility assignments are determined during the design phase before any work on individual modules is begun. One of the key areas of modular design is the selection of module contents based on the probability of change during the maintenance phase. In other words, assign those portions of programs/projects that are likely to change due to hardware or technology to their own respective modules. The advantage gained by this bit of overhead is found in the cost avoidance which follows during the maintenance stage, where up to seventy percent of a project's costs lie.

There is a paradox concerning maintenance and well written code. If one measures productivity during the maintenence phase by cost per defect, a popular method, he/she will find that very poorly written code has a lower cost per defect than well written code. This occurs because poorly written code has many errors which programmers must spend much time correcting. They, therefore, become very familiar with the program. The initial costs of relearning the program logic are spread over many errors in poorly written code, and over very few errors in well written code. However, the total cost of maintaining well written code is usually much lower. If one were to take the same well written modular code and compare it to the same well written non-modular code one should find: 1) fewer logic errors because of the extensive analysis during the design phase; 2) it's easier to locate errors since they can often be traced to one module or at least to a branch of the program; 3) it's easier to relearn the logic because of the need to only learn one or a few modules instead of the entire program. If any or all of these points are realized, FMSO could save a great deal in resources and improve customer satisfaction. Since FMSO presently must maintain over 9400 programs and respond to over 3200 program trouble reports (PTR) annually, any reduction in the cost, in time or money, on a per item basis could lead to significant savings and higher productivity ratings for program maintenance personnel.

The use of modular programming allows two other areas to be explored. The first is Parnas' [Ref. 17] idea of program families. The idea is to look at similarities in programs before looking at their differences and write generic programs based on the similarities. Then one adds the modules that will make the programs individualistic. In this way programmers can reuse existing code which is well tested and with which programmers are thoroughly familiar. This helps to reduce initial project development time and costs and to reduce maintenance costs.

The second area is that which Zoll [Ref. 18 , p. 51] refers to as "metaprogramming". This is the use of data base libraries of modular code to build complete programs. The code is generic and the metaprogrammer merely researchs the data base and selects those modules which will meet the program logic. In this way programmers write much less original code. Lanergan and Poynton [Ref. 19] report that at Raytheon Company some new applications software have been developed forty times faster than by using traditional development methods. Reused modules have been averaging between forty and sixty percent of the total LOC on major projects. The probability of inducing logic errors is reduced significantly and the probability of textual errors is also reduced due to the reduced amount of original code required. Kendall and Lamb [Ref. 20], in their research at IBM, have reported data which shows that metaprogramming from a data base of modules should be seriously considered. Their study showed that eighty percent of the applications programming effort goes into production of programs whose used life is less than eighteen months. Therefore, any reduction in the effort to develop these programs and any reduction in the maintenance effort of these programs will provide a factor of four increase in the savings to be applied to the maintenance of the twenty percent portion of the programs with a siginficantly longer life cycle.

The added attraction of modular code is the idea of completeness of the task. For a quality module to be delivered for integration it must be: 1) documented; 2) coded in its entirety; 3) tested; and 4) debugged. These are much more difficult to attain with LOC as the product. In particular, it is very difficult to test a block of LOC since it relies heavily on the remainder of the code. Therefore, it can only be examined by inspection while modules can be inspected and machine debugged to a near zero defect condition prior to integration. Although the documentation is not vital for module delivery, it can be and should be an organizational requirement.

The idea of designing projects, especially large ones, by dividing them into subprograms or modules is a very old concept in programming. During the 1970's it became a topic of high interest as a way to improve program reliability and maintainablility. Ross et al [Ref. 21], Liskov [Ref. 22], Crossman [Ref. 23], and Parnas [Ref. 24] [Ref. 17] wrote formidable papers extolling the virtues of modular programming. Yet there are many software development organizations that do not understand the term, use or value of modular programming. The Department of Defense (DOD) appears to be one organization that does not fully understand the value of modularization and reusing code. Munson [Ref. 25] points this out in his short paper on reducing software costs by reusing code. Elshoff [Ref. 26] observed this problem at the General Motors Research Lab where modularization not only appeared foreign to analysts and programmers but was viewed as detrimental to the software life cycle. The unfamiliarity with modularity is also present at the US Navy's Fleet Numerical and Oceanographic Center in some analysts and programmers. While this does not appear to be a problem at FMSO at the present, internal training may be required because of turnover of software development personnel.

This section concerns quality modules. These are modules that are coded in their entirety, tested, debugged, and documented. Each organization will have to set up the requirements for a countable module. This researcher recommends these attributes. They ensure attainment of the organization's minimum quality standards and take into consideration the programmer's time in debugging and testing the module. When reused modules are a part of the delivered product they should be counted as a percentage of one module. Basili [Ref. 15] used twenty percent in his research. This is a good starting point. But if the organization finds that this is not an accurate percentage of the time required to develop original modules then the percentage should be adjusted accordingly.

F. USER FUNCTIONS AS PRODUCTS

The previous section dealt with functions based on program structure. This section deals with functions based on user requirements. While modules may vary in length by approximately one hundred lines of code, user functions can vary up to several programs. An example of this is a single entry accounting system. A company may want a system which performs several functions such as: ledger maintenance, invoicing, file maintenance, weekly reporting, etc. Each of these operations or functions, is a deliverable product to the customer as a part of the single entry accounting package. The quality of the entire package is determined by the customer satisfaction with each individual function. Albrecht, [Ref. 27] of IBM Corporation, uses this measure as the primary means of determining productivity ratings in the Applications Development Group. He points out that one must be careful when using this measure or any other measure by keeping the major project objectives in perspective: on time, within budget, and a satisfied customer.

The specific product measure is what Albrecht calls a function value. The approach to determine the function value is to count the number of external user inputs, inquiries, outputs and master files that the project must develop as a part of the user requirements. An external user input is a communication from the user to the computer such as data forms, terminal screens, keyboard transactions, optical scanner forms and the like. These do not include inputs from tapes and data sets, which are considered as internal and part of the file count. Each of these user functions is weighted by a value designed to reflect that function's value to the customer. Appendix I shows the details of determining the function value and Appendix II shows the details of determining the sizing and complexity of an entire project using function value components. Appendix II uses the same external user inputs and some internal inputs as components to compute the function points but also provides for the the computation of a development time estimation. It is important to note that Chrysler [Ref. 2] showed in an unrelated and independent study that these components were most significant in predicting development time.

Albrecht's function value concept has several advantages over those measures previously mentioned. First, it is the only measure that deals specifically and directly with user satisfaction. The other measures virtually ignore the user between the functional specification phase and the implementation phase. This method constantly works with the user. Secondly, since its focus is on user requirements and not on counting lines or blocks of code or modules, it tends to limit programmer gaming to improve his/her productivity rating artificially. Third, the measure breaks the project into user defined portions of importance. This focuses the effort towards teamwork since it requires the development group to work as a team toward the production of functions to which the user has placed a well defined importance. Lastly, the method provides more opportunity for a smoother evolution of change than the others. It focuses attention on the cost of each function and the effects on cost of mid-development changes. The constant attention to cost and user involvement provides a better mechanism to control the change process during development. It enables the planner to design for changes that may occur during the life cycle that may not be cost effective to include during the development phase.

The function value concept has three disadvantages. First, there may some question as to whether to call a component an inquiry or an input. These are not always distinct items. If the weighting factors are different for each this may significantly alter the final function value. Second, users play a large part in determining the weighting factors, as it should be. Users can be fickle, therefore, it is often extremely difficult to get them to admit "truthfully" what they desire most. It is not so much that they are hiding information but that they don't really know what they want. Therefore, it requires talented interviewers and designers to determine the true desires of the users. The third disadvantage is that this measure is so good that managers may tend to rely on it too heavily. This is not the ultimate or universal measure but it is a good one. The other measures can give insights on products and productivity that this measure can not. The function value is an aggregate measure and must be used as such. As Stevens [Ref. 28] of Performance Management Associates Inc. of Scottsdale (Az.) points out, there is no universal measure yet. We must use all the imperfect measures available in an effort to describe the programming activity.

G. TESTING, INTEGRATION, AND IMPLEMENTATION

One of the concerns of managers, when discussing programmer productivity, is how to incorporate non-delivered code in the calculation of productivity. The non-delivered code consists of test code, debugging aids and incorrect code. The incorrect code is a function of the programmer's skill and is a penalty to his/her productivity rating. The test code and debugging aids are not mistakes. They are used by skilled programmers to ensure coding quality and correctness. There has been some concern that the programmer should have this code included with the delivered code for productivity calculations. This researcher does not concur that the test code and debugging aids should be included. The programmer's job is to deliver code that meets the specifications. The only way to ensure the code actually meets those specifications is to perform some type of test. Test code and debugging aids are tools of the programmers just as milestones and management/support are tools for others in the software development arena. They are a necessary overhead which programmers must employ if they are to deliver the quality products discussed previously.

The integration, testing and implementation phase of software development utilizes approximately forty percent of the project's resources [Ref. 37 ,p.18]. Intuitively, one would think that an area which uses so much of the resources would be a prime place to do some productivity research. This, unfortunately, is not the case. One of the prime reasons has been the inability of the industry to determine the role these activities play. Specifically, there is a question as to whether testing is a part of development or a part of quality assurance. If it is part of quality assurance then it is an overhead and not a productivity concern.

If it is a part of development then the product is tested and acceptable code. But what determines how productive the testing is? The time expended in testing does not help to determine the productivity of testing because the time used in testing is a function of the test plan and the number of defects found. Defects found does help to determine productivity. It shows either poor design, poor programming, poor quality assurance practices or any combination thereof.

Integration is left with the same type of problems. This activity takes project portions, modules, LOC, or programs, and brings them together to form a cohesive and integrated product. But if there are major difficulties encountered are they the the fault of the integrators? Probably not. The fault probably lies with the designers or the programmers.

The manager must be aware of the problems that develop during this phase and keep records of them. Though there were no conclusive reports found on how to deal with the information, the consensus from the literature is that it must kept in a data base for later study and consideration. The science of software development has not progressed far enough to completely handle the test, integration and implementation problem. Most researchers are of the belief that if we get control of the development process in a scientific way these problem areas may disappear.

H. DOCUMENTATION

The primary belief in the industry and particularly in DOD is that software development projects have two separate products: program code and program documentation. This is an extremely short-sighted but understandable belief. As long as software development is viewed as having two products, this belief presents the opportunity to discard one. Since the program is what is wanted, all too often the documentation is reduced in an attempt to reduce development costs. The view that there are two products and the practice of reducing the documentation thrive on the belief that software development and software maintenance are not related. This is not true. The documentation is required to learn the program logic and coding structure. A software project that was poorly designed and poorly or not documented is extremely difficult and much more costly to maintain than one that was well designed and well documented. Nearly every other industry (i.e. automobile, electronics, machine tools, etc.) that produces a complex product provides documentation on the logic and design of that product so that maintenance personnel can provide quality and cost effective maintenance. There is no reason to believe that the software development industry should be any different.

This researcher believes that documentation is not a separate product but an integral part of all well developed software projects. This chapter consistently discussed fully coded, well documented quality software. It should be intuitively obvious that a program that does not operate properly is of little or no value. And that one that operates properly but is difficult to understand and maintain because of poor documentation is of much less value than one with superior documentation. Thus, the documentation has no specific measure of length only one of quality. It is a problem for the developers and quality assurance experts to ensure that the documentation is provided and adequate in describing the program logic and coding structure of the project.

IV. THE MEASURES

During the research for this paper it was noted that there is a great deal of misunderstanding both in the literature and in the industry about programming and software development productivity. The misunderstanding lies in the area that, when questioned about the product that is produced, one will receive quizzical looks or long spells of silence. People immediately want to jump to discussions on complexity, language, tools or the development environment. These have little to do with calculating productivity. Their roles are as parameters within which one must analyze the specific productivity rating. This is not to belittle the importance of these areas. It is simply a matter of organizing one's thoughts. One can not intelligently speak of improving productivity until one first has a quantitative measure and secondly a description of the environment. Too often people in the industry look at the environment not only first but exclusively. Without a product definition and the measure, the environment cannot be understood.

Productivity has two components: outputs and inputs. The outputs, loosely defined, are the products previously discussed: projects, programs, functions points, modules, and LOC. They are dependent on the corporate hierarchical level and the philosophy used for software development. The inputs vary considerably depending upon which productivity measure one is interested. The most common input used is the person-month, 160-175 hours. This can be broken down into its various parts by programmers, management/support, systems analysts, and program analysts. But there are other inputs that may be worth considering such as CPU time or terminal connect time. Though, these are rarely if ever, considered.

A. LOC PER PROGRAMMER-MONTH

The most common measure used for assessing productivity throughout the industry is LOC per programmer-month. Though a very popular measure, it is not very good. Since it is based on LOC it is subject to the line counting variations mentioned in the previous chapter. This variation can be limited, to a certain extent, by setting organizational standards as recommended earlier. This would permit consistency in the organization but not across the industry. Recall, one of the reasons for measuring is to make comparisons across organizational lines. As long as there are variations in the definitions of components no intelligent comparisons can be made.

LOC per programmer-month is ineffective for noncoding tasks. The tendency when computing this measure is to use programmer-month as the total development time which includes these noncoding tasks of design, documentation, testing and management/support. Since no coding is going on during these stages it makes little sense to include them in the coding effort. Therefore, that would imply that this measure should be used only for the coding phase. Of course, that focuses attention on the coding task exclusively, which is a minimal portion of the software development effort.

Finally, this measure tends to penalize high-order language (HOL) programs in favor of programs written in Assembler language. Jones [Ref. 29, p. 21] provided the example shown in Figure 4.1 . This is an example of the same program written in two different laguages. Two of the purposes of using HOL are to cut costs and improve productivity. But the example shows the paradox of this measure. It appears that Assembler language is more productive than the HOL even though the HOL program took one month less to

Activity	Assembler Language	HOL
Design	4 weeks	4 weeks
Coding	4	2
Testing	4	2
Documentation	2	2
Mgmt/Support	2	2
Total Effort	16 weeks (4 months)	12 weeks (3 months)
Lines of code	2000	500
LOC per prog-mon	500	167

Figure 4.1 Assembler Language vs HOL.

produce. Notice also that Jones used the term "programmer-month" to mean the entire program development time, a common practice, as mentioned earlier. The actual programming times were one month and one-half month for Assembler language and HOL, respectively. Even if this time frame is used, though, the Assembler language at 2000 LOC per programmer-month appears to be twice as productive as the HOL at 1000 LOC per programmer-month. This points out the problem of not being consistent about terms. Jones uses programmer-month to mean the entire development time which yielded an average productivity figure which included a period when no coding was being done at all. Using the term strictly and comparing it to Jones' usage leaves us with a four to one difference in productivity for Assembler language and a six to one difference in productivity for the HOL.

B. MODULES PER MONTH

This particular measure was presented in a paper by Crossman [Ref. 23]. Surprisingly, this researcher could not find any other references that have attempted to duplicate his findings. Yet he pointed to several advantages which this measure and its methodology of program development support.

Modular design programming tends to minimize the complexity of projects. Minimizing the complexity parameter allows the manager to reduce the number of variables he must consider when making productivity comparisons. The definition of a module appears to be more consistent throughout industry than LOC which gives it a potentially much better comparative capability between organizations, provided the other organizations use this measure. The use of modules as a product provides a consistency throughout the development cycle. It includes the design, coding, testing, documenting, and management/support phases. Yet it can also be broken down into its individual component efforts to determine which effort has the greatest impact on development time and the impact of each module on the project as a whole.

C. FUNCTION POINT DELIVERED PER WORK HOUR

Albrecht [Ref. 27] discussed the effects this approach has on showing the relative productivities between languages, project size and various programming technologies. The method focuses on the external attributes of a program and the work-hours contributed by both IBM and customer personnel assigned to work on the project. It covers all phases of the project. The goal of this method of measurement is to state development costs in terms of the work-hours used to design, program and test the applications project. Although there is not enough data available presently to give conclusive results, the report does indicate the capability to show the relative productivities of different languages and development technologies. This is a major advantage that is not possible with LOC and has not yet been explored using modules.

D. SELECTED INDUSTRY METHODS FOR MEASURING PRODUCTIVITY

The preceding sections of this chapter discussed various methods used in research to study programmer productivity. Each method mentioned uses a ratio of outputs (project, program, specifications, modules, LOC or function value) to inputs (person-months, programmer-months, or work-hours). Previous sections provided recommended definitions for selected output and input components. This section presents measures used by several prominant corporations that develop software.

1. IBM

Measurement of programs is still a fairly subjective process. We can measure size, based on 'lines of code' or 'number of statments', but acceptance of these measures is not universal. Acceptance of lines of code, as an example, seems to be based on the view that although lines of code may be an imprecise measure, it is something that can be enumerated, and until something better is discovered we will continue to use it. There is a veiled invitation here to find something better. [Ref. 30 ,p. 372]

This is the philosophy used to approach the measuring of programming activities at the Santa Teresa Laboratory of IBM. The "something better" that IBM has been trying to refine for the last three to four years has been the software science metrics developed by Halstead [Ref. 31]. Figure 4.2 shows the major elements in use by IBM [Ref. 32] [Ref. 30]. The philosophy for using software

Operands = values that are changed or used as a reference for change (constants, variables

Operators = elements that operate on or with operands (operation codes, delimiters, punctuation, arithmetic symbols, branches (DO WHILE, IF THEN, IF THEN ELSE))

η_1 = number of unique operators used

η_2 = number of unique operands used

N_1 = number of times the operators are used

N_2 = number of times the operands are used

Vocabulary (η) = the sum of unique operands and operators used in the program. It is a measure of the repertoire of elements a programmer uses to implement a program.

$$\eta = \eta_1 + \eta_2$$

Length (N) = the sum of the operator usage and the operand usage. It is a measure of program size.

$$N = N_1 + N_2$$

Difficulty (D) = a measure of the difficulty of writing code and, intuitively, a measure of ease of reading.

$$D = \frac{\eta_1}{2} \times \frac{N_2}{\eta_2}$$

Figure 4.2 Halstead Element Relationships.

science metrics is built on the following beliefs. First, in any given language, one type of program is no harder to code than another. The experience at Santa Teresa laboratory over the last five years is that the only things that affect productivity are the language and the tools used. They have found that HOL is about twice as productive as Assembler language. Second, aside from language, the development tools are what affects programmer productivity. To this end, IBM has consistantly added to the "workbench" of their programmers. They have provided on-line programming capabilities, given each programmer his/her own terminal in his/her office, provided a dedicated program development computer and various programming aids such as Script. Third, the definition of operators and operands is consistent across language barriers. This gives software science metrics a significant advantage over other measures. Additionally, IBM research has shown that the size metrics used by Halstead are as accurate as LOC for measuring program size.

Since programming productivity is believed to be constant for all programmers, given the same environment, IBM has looked primarily at the difficulty metric. Difficulty is defined as a metric that expresses the difficulty of writing code. It takes into consideration decision nodes, the repertoire of operators used and how concise the usage of the variables is. The measure, then, also appears to be one for ease of reading. It does not tell how difficult the program must be. It only tells how difficult the programmer made the program. High difficulty can come from poor programming skills, poor program structure, inexperience with the language or the complexity of the algorithm. The value of this metric is three fold. It tends to indicate error-proneness much earlier in the development cycle than traditional methods. Intuitively, the more difficult the program, the more error-prone it is. The measure can only be taken after coding has been completed but it can be calculated immediately following the first clean compile. There is no need to wait for testing. Secondly, it points out those programs which need rework due to high difficulty values. Third, it points out programmers who consistently have high difficulty values. This enables the manager to ensure that the programmer receives added training in the techniques available to reduce program difficulty. IBM has found that the difficulty measure tends to range from three to eight. When ever they see that a difficulty measure exceeds five, they call the programmer in to have him/her recode the program to reduce the difficulty measure to five or less. If the programmer consistently delivers code with high difficulty measures he/she is provided added training in techniques which can lower the program difficulty.

All this only gives measures of the program not the prodcuctivity of the programmer. For IBM to determine that all programmers had the same productivity, they had to test. The test measure was and continues, on a minor basis, to be LOC per person-year. LOC is defined as data declarations and executable statments. The use of this measure, now, is only to check for changes in productivity due to new tools and for reasonable production rate relative to the industry. IBM recognizes the comparability problem of the LOC measure. However, the IBM perceived industry average ranges between 800 and 2500 LOC per year, given the line counting variations. They continue to measure productivity using LOC per man-year to ensure that IBM remains wihtin this range.

2. <u>Amdahl</u>

a. System Software

Amdahl's approach to systems software development is different from most of the industry. As a manufacturer of IBM compatible hardware and software, their approach is to use IBM software products and modify them to operate more efficiently on Amdahl hardware. This means placing "hooks" into the IBM software to operate special Amdahl procedures. Since their goal is develop more efficient software, these hooks must be minimal in both length and interference with the existing software and logic. Amdahl places a much higher emphasis on quality than quantity.

In this light, none of the previously discussed measures apply. Amdahl uses a management by objectives (MBO) approach to measure performance. Their hiring practices aim towards acquiring those programmers who are experienced, skilled and senior in the industry. The programmers are organized into groups of two to three assigned to one team leader. Each group has its own area of responsibility for program development/modification. The assignment of tasks and the time constraints are determined by mutual agreement between the manager and team leader. The schedules are recorded and each programmer is evaluated on his/her performance. The evaluation is discussed with the respective programmer at the periodic performance review. Since each group has specific areas of responsibility and those areas are limited, any trouble reports received are easily assigned to the group and/or individual responsible. These are also included in the performance review. This scenario does allow any specific measure to quantify programmer performance. However, the programming section is a small organization, 50-75 programmers, so they track the type of modification against the time required and the quality of the programming. They do not use any particular measure outside of budget and schedule. [Ref. 33]

b. Applications Software

Amdahl's application program development is very similar to the systems software development in that they use MBO as the predominant measure. They do use LOC per programmer year to do some measuring but it has very little significance to the operation. LOC is defined as all programmer-original COBOL statments. No credit is given for reused code, although, they admit some credit should be given. This would appear to discourage reusing code but their incentive, reward and penalty system provides the necessary encouragement. How the system functions was not specified. Management does require programmers to use data dictionaries, and code libraries are kept in an on-line data base. The primary measure used to measure performance is a review of the programmer's schedule. The programmer submits a schedule of task accomplishment to the manager. The manager reviews it to ensure it is realistic and then compares the schedule to the task completion dates as the programmer delivers the assigned tasks. Here, as in systems development, the primary ingredient for measuring is programmer and manager experience. [Ref. 34]

The measure used to evaluate maintenance programming is built around the number of trouble reports received. Each programming group is responsible for maintenance of its assigned software. Team leaders must emphasize high quality in the software to avoid having to reschedule programmers onto maintenance from development. This does not prevent errors but it does cut them down. The main emphasis from the Applications Programming Manager is to ensure as rapid a response time as possible on the trouble reports. The required turnaround time for trouble reports, presently, is not to exceed six months. They use the turnaround measure because it tends to indicate to the users that the company is genuinely interested in the productivity of software maintenance. It also gives the respective managers an additional reason when requesting more resources. Finally, it gives a business value to organized maintenance because it forces the various managers to schedule resources for program maintenance.

Amdahl uses program packages predominantly in their applications programming section. These packages come with their own documentation which allows Amdahl to take take an approach significantly different from this researcher's view point. Amdahl believes program code and documentation to be separate and unequal products. This belief is made possible because they have programs that can analyze code and tell the programmer the structure of the code. therefore, they feel that program maintenance without the documentation is not as difficult one might assume. However, documenation is encouraged. The method used is to request documentation and to make it as easy to provide as possible. To make the documentation easier, it is all written on-line using Script and a variety of user-developed macros that provide some graphics to enhance the prose. The documentation quality is now much higher and the documentation is much easier for the programmers to deliver. [Ref. 34]

3. Systems Development Corporation (SDC)

SDC's cost estimating procedures use LOC and pages of documentation as the primary productivity inputs to compute costs. They categorize the various types of LOC (data definitions, executable statements, reused code, etc.) to determine the subtask cost for each activity. The LOC are weighted by an in-house complexity measure which includes parameters for program size, security, and reliability. Each productivity measure is computed relative to the type of program (real-time process control, interactive, report generator, data base control, etc.) that was produced. Documentation is mesured by pages produced per day per type of program. Although they call documentation a separate product, they consider all projects to be integrated packages of both software code and documentation. [Ref. 35]

4. TRW

TRW uses a weighted LOC per man-month method to measure productivity. They reviewed Halstead's metrics but concluded, as did IBM, that source LOC is equivalent to the size metrics developed from counting operators and operands, They do concede that the difficulty metric deserves more study but they have no resources avialable at present to conduct such a study. They have found that weighting the LOC with an in house factor for complexity and reliability is sufficient. The LOC is defined as a delivered well documented and well engineered line equal to a card image. The card image is an eighty character line. Comment lines are not included but all lines which hold "computing" information are (e.g. job control language, edit links, format statements, data declarations, executable statements, etc.). TRW defines a man-month, 152 hours, to include all personnel hours directly chargeable to the project.

At present, TRW does not measure maintenance productivity. However, the interview with Dr. Boehm [Ref. 36], recommended the method discussed in his book Software Engineering Economics [Ref. 37]. This method equates the annual maintenance effort to the annual change traffic (ACT) multiplied by the estimated development effort. ACT is the fraction of the software product's source instructions which undergo change during a typical year, either through addition or modification.

TRW includes documentation in its definition of a LOC. This corresponds with the philosophy of this researcher. TRW does not treat software code and documentation as separate products but as integral parts of the software project.

V. CONCLUSIONS AND RECOMMENDATIONS

This paper has attempted to point out the major areas which must be explored in order to measure and discuss programmer productivity or software development productivity. The manager must decide what level of the organization he wishes to measure. He then must determine what, specifically, the product is which that level is making. Before proceeding any further, he should examine the quality assurance procedures and practices to ensure that they are both in use and that they do establish and check for a minimum quality standard. From here the manager can select the various inputs which he feels are relevant to study. The productivity rates he computes need to be stored in a data base so that they may be used as comparators against time and other organizations. Finally, each measure must be kept in the context of its environment. This condition provides two functions. First, it keeps the measure meaningful. Second, by selectively changing one element of the environment at a time, the manager can determine cause and effect relationships that can lead to establishing the optimum software development environment.

The LOC measures are poor for software development and lead to paradoxical conclusions in many instances. Remaining with any measure that uses LOC will tend to bind the organization to technologies requiring the development of totally original code on every project. This will tend to prevent the use of metaprogramming and the development of program families. These programming technologies show significant promise to reduce development costs and improve programming productivity dramatically.

Modular measures provide the opportunity to explore and develop the metaprogramming practice. They also have overheads that must be accepted as development personnel learn the technology, the added effort required in the design phase, particularly for "small" projects, and the possible inefficient use of CPU time due to an increase in the number of LOC. These are small overheads to pay if the development time can be reduced by as much as Lanergan [Ref. 19] claims. The measure can be used in conjunction with any other measure to help define the programming activity better. It may be especially useful in conjunction with function points.

In closing, it is apparent for the literature and the discussions with the selected industry corporations that there is no perfect and correct measure or method for measuring programmer productivity. However, the vital point to understand is that nearly all organizations do measure programmer productivity in some fashion. Several organizations admit that their methods lack some possibly important inputs or parameters. However, each organization does attempt to measure productivity so that each can gain some understanding of the organization's particular environment. With an understanding of the environment, each organization and researcher is able to conceptualize the software development process so that the manager can make intelligent assertions about how it is affected.

LIST OF REFERENCES

1. Lewis, Tim, "Missing Computer Software", Business Week, pp. 46-53, 1 September 1980.

2. Chrysler, Earl, "Some Basic Determinants of Computer Programming Productivity", Communications of the ACM, vol. 21, pp. 471-483, 1978.

3. "A Rush of New Companies to Mass-produce Software" Business Week, pp. 54-56, 1 September 1980.

4. Azuma, M. and Mizuno, Y., "STEPS: Integrated Software Standards and its Productivity Impact", IEEE Computer Society Conference Proceedings (COMPCON 81), pp. 83-95, FALL 1981.

5. Wasserman, Anthony I. and Belady, L.A., "Software Engineering: The Turning Point", Computer, pp.30-39, September 1978.

6. "An Acute Shortage of Programmers" Business Week, pp. 49, 1 September 1980.

7. Kiser, Barbara C. Stewart, "Software Management Productivity - Understanding the Software Development Process", IEEE Computer Society Conference Proceedings Fall 1981, pp. 244.

8. Munsun, J.B. and Yeh, R.T. (co-chairman), "Report from the Measurements Workshop of the IEEE Workshop on Software Productivity", IEEE Computer Society Conference Proceedings Fall 1981, pp. 339-347.

9. Simon, Julian L.,is" Basic Research Methods in Social Science, pp. 287-291, Random House, New York, N.Y., 1978.

10. Pazur, Ron, interview on 30 September 1982, Fleet Material Support Office, code 9212, Mechanicsburg, Pa., AV 430-2434.

11. Jones, T.C., "The Limits of Programming Productivity" Proceedings of the Joint SHARE/GUIDE/IBM Application Development Symposium. Undated.

12. Keider, Stephen P., "Why Projects Fail" Datamation, pp. 53-55, December, 1974.

13. Howden, William E., "Life-Cycle Software Validation", Computer, pp. 71-78, February 1978.

14. Fox, Joseph M., Software and its Development, pp. 226-251, Prentice-Hall, Englewood Cliffs, N.J. 07632, 1980.

15. Jones, T.C., "Measuring Programming Quality and Productivity", IBM Systems Journal, vol. 17, no. 1, pp.39-63, 1978.

16. Basili, Victor R. and Freberger, Karl, "Programming Measurement and Estimation in the Software Engineering Laboratory" The Journal of Systems and Software, vol. 2, pp. 47-57, 1981.

17. Parnas, D.L., "Designing Software for Ease of Extension and Contraction", IEEE Transactions on Software Engineering, pp. 226-236, March, 1979.

18. Zoll, Peter F., "Measuring Programming Productivity" Computer Performance Evaluation Users Group 16th Meeting, (NBS-SP-500-65), pp. 49-52, 1980.

19. Lanergan, Robert G. and Poynton, Brian A., "Reusable Code - The Application Development Technique of the Future", Proceedings of the Joint SHARE/GUIDE/IBM Applications Development Symposium, pp. 127-136, October 1979.

20. Kendall, R.C. and Lamb, E.C., "Management Perspectives on Programs, Programming and Productivity", GUIDE 45, Atlanta, Ga., November, 1977.

21. Ross, Douglas T., Goodenough, John B., and Irvine, C.A., "Software Engineering: Process, Principles and Goals", Computer, pp. 54-64, May, 1975.

22. Liskov, B.H., "A Design Methodology for Reliable Software Systems", Proceedings, Fall Joint Computer Conference, pp. 65-73, 1972.

23. Crossman, Trevor D., "Taking the Measure of Programmer Productivity", Datamation, pp. 144-147, May 1979.

24. Parnas, D.L., "On the Criteria to be used in Decomposing Systems into Modules", Communications of the ACM, pp. 220-225, March, 1979.

25. Munson, John B., "Improving Software Engineering Productivity", IEEE Computer Society Conference Proceedings (COMPCON 81), pp. 310, September 1981.

26. Elshoff, James L., "A Review of Software Measurement Studies at General Motors Research Laboratories", Proceedings US Army/IEEE Second Life Cycle Management Conference, pp. 172-173, August 1978.

27. Albrecht, Allan J., "Measuring Application Development Productivity", Proceedings of the Joint SHARE/GUIDE/IBM Application Development Symposium, pp. 83-92, October 1979.

28. Stevens, Barry, "Productivity: The First Step", Software News, pp. 28-30, March 1, 1982.

29. Jones, Capers, Program Quality and Programmer Productivity, TR 02.764, IBM Corp., General Products Division 5600 Cottle Road, San Jose, Ca. 93193, January 28,1977.

30. Christensen, K., Fitsos, G.P., Smith, C.P., "A perspective on Software Science", IBM Systems Journal, vol. 20, No. 4, pp. 372-387, 1981.

31. Halstead, M.H., Elements of Software Science, New York, New York, 1977.

32. Christensen, Ken, Interview, IBM Santa Teresa Laboratory, 555 Bailey Avenue, P.O. Box 50020, San Jose, Ca. 95150, 408-463-3127, September, 1982.

33. Patrick, Rich, Amdahl Corporation 1250 East Arquez Avenue, P.O. 470, Sunnyvale, Ca. 94086, 408-746-8916, September, 1982.

34. Berry, Mike, Amdahl Corporation, Sunnyvale, Ca., 408-746-6000, December, 1982. Interview.

35. Wong, Carolyn, SDC 2500 Colorado Avenue, M.D. 32-61, Santa Monica, Ca. 90406 213-820-4111, Interview, December, 1982.

36. Boehm, Barry, TRW-DSG 1 Space Park, R2-1076, Redondo Beach, Ca. 90278, 213-535-2184, Interview, December, 1982.

37. Boehm, Barry Software Engineering Economics, Prentice-Hall, Inc., Englewood Cliffs, New Jersey 07632, 1981.

BIBLIOGRAPHY

Albrecht, A.J., "Measuring Application Development Productivity", IEEE Computer Society Conference Proceedings Fall 1981, Washington D.C. pp. 232-241

Barakat, D.H., "Productivity and the Development Environment", IEEE Computer Society Conference Proceedings Fall 1981, Washington D.C.

Basili, V.R. and Freberger, B., "Programming Measurement and Estimation in the Software Engineering Laboratory", Journal of Systems and Software, vol. 2, February 1981. pp. 47-57

Basili, V.R. and Philips, Tsai-Yun, "Evaluating and Comparing Software Metrics in the Software Engineering Laboratory", Performance Evaluation Review, vol. 10, Spring 1981. pp. 95-106

Byars, L.L., "Solutions to Productivity Problems", Journal of Systems Management, vol. 33, January 1982. pp. 26-35

Chapin, N., "A Measure of Software Complexity", Proceedings of the National Computer Conference 1979, pp. 995-1002

Chen, E.T., "Program Complexity and Programmer Productivity", IEEE Transactions of Software Engineers, vol. SE-4, no. 3, 1978. pp. 187-194

Chrysler, E. "The Impact of Program and Programmer Characteristics on Program Size", AFIPS National Computer Conference, 1978. pp. 581-587.

Curtis, B., Sheppard, S.P., Borst, M.A., Milliman, P., and Love, T., "Some Distinctions Between Psychological and Computational Complexity of Software", Proceedings, U.S. Army / IEEE Second Life Cycle Conference, Atlanta, Ga. August, 1978. pp. 166-171.

Curtis, B., Sheppard, S.P., Borst, M.A., Milliman, P., and Love, T., "Measuring Psychological Complexity of Software Maintenance", IEEE Transactions of Software Engineers, March, 1979. pp. 96-104.

Curtis, B., Sheppard, S.P., Borst, M.A., Milliman, P., and Love, T., "Third Time a Charm", Proceedings, Fourth International Conference on Software Engineering, September, 1979. pp. 356-360

Fitzsimmons, A. and Love, T., "A Review and Evaluation of Software Science", Computing Surveys, vol. 10, no. 1, March, 1978.

Gilb, T., Software Metrics, Winthrop Computer Systems Series, Winthrop Publishing Company, Englewood, N.J. 1976.

Halstead, M.H., "Software Science - A Progress Report", IEEE / U.S. Army Second Software Life Cycle Workshop, August 21-22, 1978, Atlanta, Ga.

Halstead, M.H., Elements of Software Science, New York, N.Y. 1977.

Jeffery, D.R. and Lawrence, M.J., "Some Issues in the Measurement and Control of Programming Productivity", Information and Management, vol. 4, September, 1981. pp. 169-176.

Jeffery, D.R. and Lawrence, M.J., "An Inter-organisational Comparison of Programming Productivity", IEEE Proceedings of the Fourth International Conference on Software Engineering], 1979. pp. 369-377.

Johnson, J.R., "A Working Measure of Productivity", Datamation, vol. 23, no. 2, February, 1977. pp. 106-112.

Jones, T.C., "Productivity Measures", Proceedings of Guide 44, San Francisco, Ca. May 1977.

Jones, T.C., "The Limits of Programming Productivity", Proceedings of the Joint SHARE / Guide / IBM Applications Symposium, October, 1979. pp. 77-82.

Kirkley, J.L., "Programmer Productivity", Datamation, vol. 23, no. 5, May, 1977. pp. 63-69.

Lehman, M.M., "Programming Productivity - A Life Cycle Concept", IEEE Computer Society Conference Proceedings, Fall 1981.

Linger, R.C., "Human Productivity and Software Development", IEEE Computer Society Conference Proceedings, Fall 1981.

Markham, D., McCall, J. and Walters, G., "Software Metrics Applications Techniques", Proceedings of Trends and Applications 1981 Advances in Software Technology], IEEE; NBS. pp. 38-46.

McCabe, T.J., "A Complexity Measure", IEEE Transactions on Software Engineering, SE-2, 1976, pp. 308-320.

McCall, J., Richards, P. and Walters, G., "Metrics for Software Quality Evaluation and Prediction", Proceedings of NASA / Goddard Second Summer Engineering Workshop, September, 1977.

Mitchell, J., "Productivity and Software Tools", IEEE Computer Society Conference Proceedings, Fall 1981.

Parikh, Girish, How to Measure Programmer Productivity, Shetal Enterprises, Chicago, Il., 1981.

Patrick, R.L., "Probing Productivity", Datamation, September 1980, pp. 207-210.

Perlis, A.J., Sayward, F.G and Shaw, M. (eds), Software Metrics: An Analysis and Evaluation, MIT Press, 1981, pp. 137.

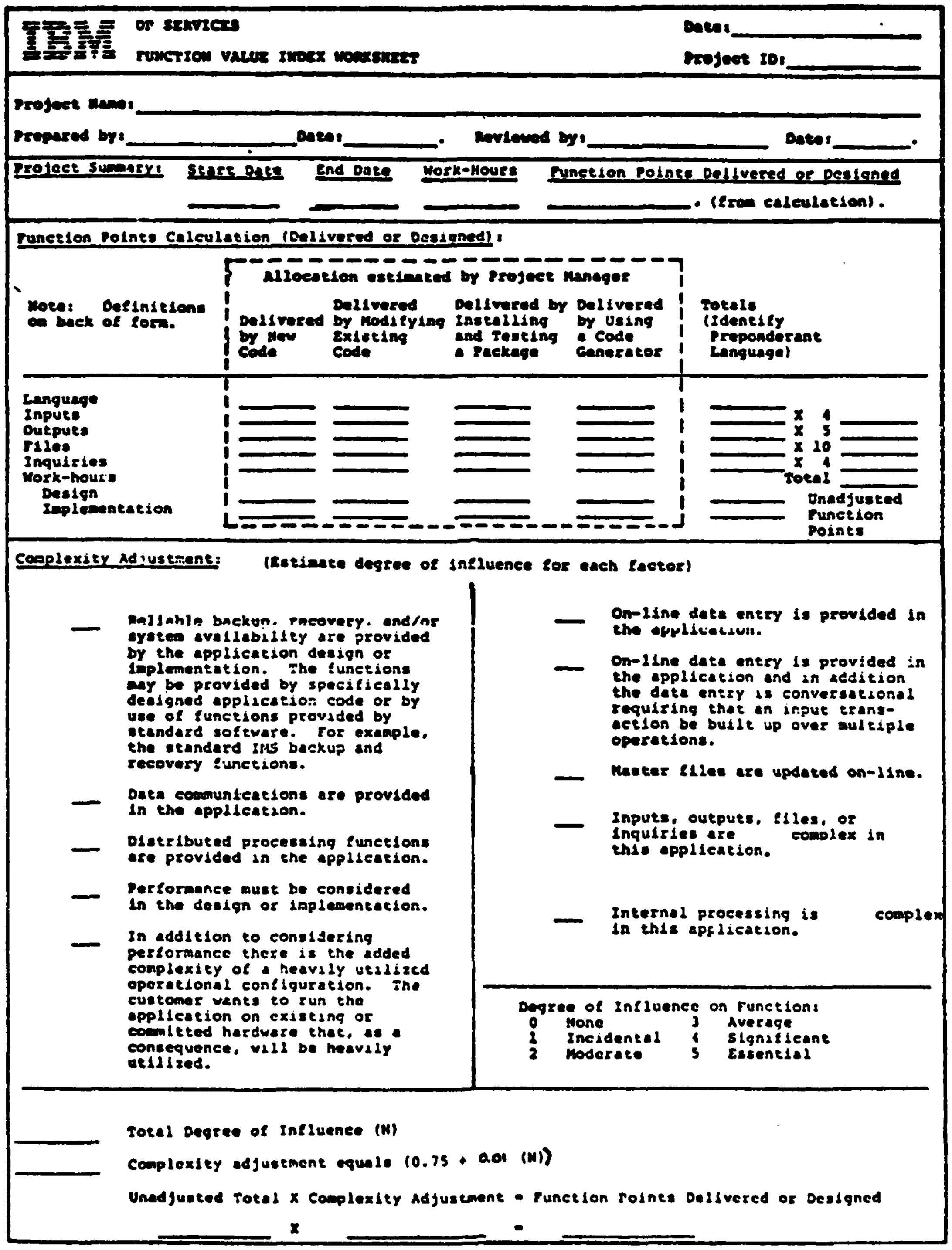
Putnam, L.H., "Measurement Data to Support Sizing, Estimation, and Control of the Software Life Cycle", IEEE Computer Society Conference Proceedings, Spring 1978, pp. 352.

Putnam, L.H. and Fitzsimmons, A., "Estimating Software Costs", Datamation, vol. 25, no. 10, September 1979, pp. 189-198.

Smith, C.P., "A Software Science Analysis of Programming Size", ACM 80, Proceedings of the Annual Conference, Nashville, Tn., October 27-29, 1980, pp. 179-185.

Walston, C.E. and Felix, C.P., "AMethod of Programming Measurement and Estimation", IBM Systems Journal, vol. 16, no. 1, 1977, pp. 54-73.

APPENDIX I



IBM DP SERVICES **Date:** ______

FUNCTION VALUE INDEX WORKSHEET **Project ID:** ______

Project Name: ______

Prepared by: ______ **Date:** ______. **Reviewed by:** ______ **Date:** ______.

Project Summary:	Start Date	End Date	Work-Hours	Function Points Delivered or Designed
	______	______	______	______. (from calculation).

Function Points Calculation (Delivered or Designed):

Note: Definitions on back of form.

	Allocation estimated by Project Manager					
	Delivered by New Code	Delivered by Modifying Existing Code	Delivered by Installing and Testing a Package	Delivered by Using a Code Generator	Totals (Identify Preponderant Language)	
Language	______	______	______	______		
Inputs	______	______	______	______	______ X 4	______
Outputs	______	______	______	______	______ X 5	______
Files	______	______	______	______	______ X 10	______
Inquiries	______	______	______	______	______ X 4	______
Work-hours					Total	______
Design	______	______	______	______	______	Unadjusted Function Points
Implementation	______	______	______	______	______	

Complexity Adjustment: (Estimate degree of influence for each factor)

___ Reliable backup, recovery, and/or system availability are provided by the application design or implementation. The functions may be provided by specifically designed application code or by use of functions provided by standard software. For example, the standard IMS backup and recovery functions.

___ Data communications are provided in the application.

___ Distributed processing functions are provided in the application.

___ Performance must be considered in the design or implementation.

___ In addition to considering performance there is the added complexity of a heavily utilized operational configuration. The customer wants to run the application on existing or committed hardware that, as a consequence, will be heavily utilized.

___ On-line data entry is provided in the application.

___ On-line data entry is provided in the application and in addition the data entry is conversational requiring that an input transaction be built up over multiple operations.

___ Master files are updated on-line.

___ Inputs, outputs, files, or inquiries are complex in this application.

___ Internal processing is complex in this application.

Degree of Influence on Function:

0	None	3	Average
1	Incidental	4	Significant
2	Moderate	5	Essential

______ Total Degree of Influence (N)

______ Complexity adjustment equals (0.75 + 0.01 (N))

Unadjusted Total X Complexity Adjustment = Function Points Delivered or Designed

______ X ______ = ______

Definitions:

General Instruction:

Count all inputs, outputs, master files, inquiries, and functions that are made available to the customer through the project s design, programming, or testing efforts. For example, count the functions provided by an IUP, FDP, or Program product if the package was modified, integrated, tested, and thus provided to the customer through the project's efforts.

Work-hours:

The work-hours recorded should be the IBM and customer hours spent on the DP Services standard tasks applicable to the project phase. The customer hours should be adjusted to IBM equivalent hours considering experience, training, and work effectiveness.

Input Count:

Count each system input that provides business function communication from the users to the computer system For example:

- data forms
- scanner forms or cards
- terminal screens
- keyed transactions

Do not double count the inputs. For example, consider a manual operation that takes data from an input form, to form two input screens, using a keyboard to form each screen before the entry key is pressed. This should be counted as two (2) inputs not five (5).

Count all unique inputs. An input transaction should be counted as unique if it required different processing logic than other inputs. For example, transactions such as add, delete, or change may have exactly the same screen format but they should be counted as unique inputs if they require different processing logic.

Do not count input or output terminal screens that are needed by the system only because of the specific technical implementation of the function. For example, DMS/VS screens, that are provided only to get to the next screen and do not provide a business function for the user, should not be counted.

Do not count input and output tape and file data sets. These are included in the count of files.

Do not count inquiry transactions. These are covered in a subsequent question.

Output Count:

Count each system output that provides business function communication from the computer system to the users. For example:

- printed reports
- terminal printed output
- terminal screens
- operator messages

Count all unique external outputs. An output is considered to be unique if it has a format that differs from other external outputs and inputs, or, if it requires unique processing logic to provide or calculate the output data.

Do not include output terminal screens that provide only a simple error message or acknowledgement of the entry transaction, unless significant unique processing logic is required in addition to the editing associated with the input, which was counted.

Do not include on-line inquiry transaction outputs where the response occurs immediately. These are included in a later question.

File Count:

Count each unique machine readable logical file, or logical grouping of data from the viewpoint of the user, that is generated, used, or maintained by the system. For example:

- input card files
- tape files
- disk files

Count major user data groups within a data base. Count logical files, not physical data sets. For example, a customer file requiring a separate index file because of the access method would be counted as one logical file not two. However, an alphabetical index file to aid in establishing customer identity would be counted.

Count all machine readable interfaces to other system as files.

Inquiry Count:

Count each input/response couplet where an on-line input generates and directly causes an immediate on-line output. Data is not entered except for control purposes and therefore only transaction logs are altered.

Count each uniquely formatted or uniquely processed inquiry which results in a file search for specific information or summaries to be presented as response to that inquiry.

Do not also count inquiries as inputs or outputs.

APPENDIX II

DP SERVICES DESIGN PHASE **Section 6.2**
SIZE AND COMPLEXITY FACTOR ESTIMATOR FORM **12-31-78**

Customer:______________________________**Project ID:**____________________

Project Description:____

____

Prepared By:______________________________**Date Prepared:**____________________

When Prepared: (check one block)

( ) Before any Phase Completion	( ) Coding Specs Complete
( ) Requirements Complete	( ) Integration Complete
( ) External System Design Complete	( ) System Test Complete
( ) Internal System Design Complete	( ) System Demo Complete

DESIGN PHASE
SIZE AND COMPLEXITY
FACTOR
ESTIMATOR FORM

DP SERVICES
DATA PROCESSING DIVISION
IBM CORPORATION

QUESTION DEFINITIONS

1. SCOPE OF THE INVOLVEMENT WITHIN THE COMPANY

a. Company Functional Organizations:

Identify the number of independent organizational entities which will be involved either directly or indirectly in the project For example, if the system includes two business functions inventory control and billing, at least two organizations probably would be involved. Direct involvement refers to actual participation in the requirement study or design. Indirect involvement refers to review and approval of the requirements or design. The organizations may be counted separately in each location. For example, if the accounting department has a subdepartment in each of three geographic locations, and if each must either be interviewed or included in the approval cycle, then the accounting function should be counted as three organizations rather than one. Always include the data processing organization.

b. Company Locations:

Identify the number of company locations that require travel for information, interviews or approvals. The primary location must also be counted. Each city involved would be a location. Where multiple locations exist in the same city, consider each as half a location.

c. Number of people in the organizations involved:

Identify the number of hundreds of people in each organization identified in question 1a) above. For example, a project involving two organizations, one with 135 people, and one with 50 people would count as three hundreds of people. This provides a definition of complexity of interviews and requirements definition.

2. FUNCTIONAL SIZE OF THE APPLICATION

a. Number of Major Subsystems:

1. SCOPE OF THE INVOLVEMENT WITH THE COMPANY

a.	Number of company functional organizations involved:	_____ x 4 = _____	
b.	Number of company locations involved:	_____ x 12 = _____	
c.	Number of 100 (s) of people in the involved organizations:	_____ x 2 = _____	_____ F1

In general, a major subsystem is equivalent to a major application or system function. Examples of subsystems within an Order Processing System might be:

- Order Entry
- Accounts Receivable
- Inventory Update
- Inventory Replenishment
- Shipping
- Recovery and Restart
- Invoicing
- Management Reporting
- File Administration
- File Conversion

If you think that a function is logically separable and reasonably significant in size then count it as a subsystem.

b. Number of External Inputs:

This question addresses all system input vehicles that provide business function communication from the users to the computer system (e.g., data forms, terminal screens, keyboard transactions, optical scanner forms). It does not include internal inputs such as tape and file data sets. These are included in the count of files. It should not include input screens that are needed by the design only because of the specific implementation (e.g., DMS/VS screens that are only provided to get to the next screen but do not provide input of a business function or business information for the terminal user.)

It should include the inputs associated with all the functions committed in the design. If such functions as File Conversion and Data Base Maintenance are to be supported their inputs must be counted even if they are used only once.

On-line inquiry transactions should not be counted here since they are included separately in a later question.

The objective of this question is to enumerate all unique inputs. An input transaction should be counted as unique if there is any possibility that it will require different processing logic than other transactions. For example, transactions which have exactly the same screen format and differ only in a code used to indicate transaction type (e.g., add, delete, change) should each be counted separately as unique transactions.

c. Number of External Outputs:

2. FUNCTIONAL SIZE OF THE APPLICATION

a. Number of Major Subsystems: _____x10 =______

b. Number of External Inputs: _____x 3 =______

As with the External Inputs this question addresses all system output vehicles that provide business function communication from the computer system to the users (e.g., printed reports, output screens, hard copy terminal output operator messages). On-line inquiry transactions, where the response occurs immediately on-line should not be included in this count. However, printed reports which are triggered by off-line or on-line inquiries should be included in this count. The count should not inlcude output screens that are needed by the design only because of the specific implementation (e.g., DMS/VS screens that are only provided to get to the next screen but do not provide a business function or business information for the terminal user.)

An output is considered to be unique if it has its own format which differs from other external outputs, or if it requires unique processing logic to provide or calculate the output data.

d. Number of Files:

This count should include each planned unique machine readable logical file, or logical grouping from the viewpoint of the user, that is to be generated by or input to the system (e.g., card types, data base files, disk files, tape files). This question is oriented toward logical files not physical data sets. For example, a customer file requiring a separate index file because of the access method chosen during design would be counted as 1 logical file not 2. However, a special alphabetical index file to aid in establishing customer identity would be counted separately.

This count should include all machine readable interfaces to other computer systems.

e. Number of On-line Inquiry types:

This question addresses conversational input/response couplets where the on-line input generates and directly causes an immediate on-line output. These couplets generally do not enter data except for control purposes and therefore alter only transaction logs.

In determining this count consider each uniquely formatted or uniquely processed inquiry (input/response pair) which results in a file search for specific information or summaries of groups of information to be presented as output response to that inquiry.

Inquiries should not also be counted as inputs or outputs.

c.	Number of external Outputs:	______ x 3 = ______	
d.	Number of Files:	______ x 7 = ______	
e.	Number of On-Line Inquiry Types:	______ x 4 = ______	______ F2

3. COMPLEXITY OF THE OVERALL DESIGN PHASE

a. Customer Capability:

Consider whether the customer has data processing or user capability that will provide a good environment for requirements definition and system design or whether his people will require more that normal explanation and justification for routine decisions.

On the other hand does the customer have so much expertise that his design convictions will complicate the job beyond that normally expected. (e.g., an application well suited to IMS but the customer wants to develop his own TCAM data base system.)

Both situations would hinder the project.

b. Existing Customer Function:

Does the customer currently perform the business functions that are to be included in the system or is this a new business area?

An example of a new function that would result in a "no" answer would be, an insurance company that does not currently handle group dental plans but wants to develop an automated system to process group dental claims so that they can compete for that type of business.

c. Existing EDP System:

If the answer to the previous question was No, then this question must also be answered No. If the customer currently is performing the majority of the business functions to be included in the system and a significant number of these are being supported by existing EDP System(s), the answer should be Yes. Otherwise, the answer is No.

d. First of a Kind:

Has this application ever been computerized before, anywhere? Is this the first attempt to automate a significant business function in the application? A Yes to either question should make this system the First of a Kind.

e. Hardware and Software Operational Environment:

This question is addressing the overall complexity of the estimated operational system. An example of a Simple system

3. COMPLEXITY OF OVERALL DESIGN PHASE

a. Will the customer's capability hinder:
No (0), Yes (10) _____

b. Existing customer function to be automated:
No (10), Yes (0) _____

c. Does an EDP system exist now to perform the function:
No (6), Yes (0) _____

d. Is this system the first of its kind anywhere:
No (0), Yes (10) _____

environment would be S/370 Models 115 or 125, DOS or DOS/VS and the IBM Standard TP and data base products that operate on that level CPU.

An example of an In Between system environment would be S/370 models 135 or 145, DOS, DOS/VS or OS/VS and CICS or DL/I or something equivalent.

Large computers or more sophisticated operating System (e.g., MVS) or TP or DB environment (e.g., IMS or TCAM) would be considered as Complex. Distributed processing and programmable terminals would also be considered complex.

4. SOPHISTICATION EXPECTED OF THE SYSTEM

a. In answering the availability question consider how important it is that the system be kept available to the users. The whole data processing system including communications and terminals should be considered. Can work be postponed?

Will components be duplicated to increase system availability? This can indicate critical availability. Will the system be designed to recover quickly from failure? This can indicate important availability.

A batch system usually requires normal availability. A data collection system with non-perishable inputs, such as paper claim forms, might justify important availability. A passenger reservation system or bank funds transfer system might require critical availability.

b. Will a major or important design consideration be, that each operation or function identified as critical have an alternate method. The alternate may involve manual operations and may take longer but the function is provided.

c. Will the system contain data that must be protected against loss? Will the function require special recovery design in either procedures or system? If so, the answer is yes.

d. Data Traffic Load or System Performance:

In some systems, the volume of data to be handled is not a design concern. Other systems require special design considerations such as: use of file access optimization, simplified input notation, or extensive use of exception reporting. Transaction rates may be a problem in either on-line or batch systems. Large

e. Hardware and software system operational environment to be required by the application:
Simple (0), In-between (5), Complex (10) _____ _____

F3

4. SOPHISTICATION EXPECTED OF THE SYSTEM

a. Availability is: Critical (8), Important (4), Normal (0) _____

b. Is an alternate method, for performing the functions of the system, non-routine consideration:
No (0), Yes (6) _____

c. Is system recovery or protection against data loss a non-routine consideration:
No (0), Yes (5) _____

volumes of data in short periods (peak loads) or volumes of data large enough to cause machine availability problems are all considered data traffic considerations.

System performance is often a significant design consideration in systems that are intended to handle large volumes of data. It can also be of major concern in the design of systems with relatively low transaction rates but with constraints (perhaps economic) in terms of the prescribed hardware and software environment. For example, there may be limitations on the size of main storage, control program multi-programming capabilities, or transmission line speeds.

e. Nature of the Application:

A batch system operates as a job shop, often scheduled. Transactions are typically batched external to the computer and periodically processed sequentially against the master files.

An on-line system generally requires a more sophisticated man/machine interface than a batch system. It is generally a system where transactions are entered as they are received with no opportunity for time saving sorting. The inputs are not perishable (i.e., they can be re-entered if necessary). An on-line order entry system, or an on-line stock location and inventory control system would be examples of on-line.

A real-time system is similar to an on-line system in that it is available on demand, but it has an additional requirement to not postpone its main line processing. Response time is exceptionally important. Immediate processing and response is necessary to meet the functional requirements of the system. Process control, production test stand control, and airline reservation systems are examples of real-time systems where degraded performance may cause lost production or lost business.

f. Processing Complexity:

This question addresses the internal processing logic required to provide the majority of the proposed system functions. Straightforward logic would involve simple transformations or mapping from the system inputs or files to the system outputs. For example, a transaction is read, verified to a limited degree and used to update a simple master file or to generate a simple report. Processing is a straightforward set of pre-specified rules. Few, if any, data transformations are done. Outputs are

d. Is data traffic load or system performance an important design consideration:
No (0), Either (10), Both (20) _____

e. Nature of the Application:
Batch (0), On-Line (10), Real-Time (20) _____

mostly collections in various sets, of established data from files.

Complex should be checked if the system has a preponderance of exception processing resulting in many incomplete transactions that must be resolved later or again. Complex logic would also be the answer if there are many interactions and decision points and extensive logical or mathematical equations. In-between is used if it fails to meet either of the above definitions.

g. Exception Correction:

Systems which are designed primarily to process correct data and to detect and present bad or unusual data for manual review and correction are manual exception systems. If the system is to be designed not only to detect, but also, automatically to correct a significant number of unusual conditions, the system is an automatic exception system. This is true even if the options selected or corrections applied are to be reviewed and verified manually.

5. KNOWLEDGE WE HAVE FOR THIS PROJECT

a. Consider the Services Area in general and specifically the people who may influence the project through:

- Project Management
- Proposal Preparation
- Systems Assurance
- Project Team Performance

Consider the Area's current knowledge and the available Industry knowledge. If none of the people in the performing Area have designed or implemented this type of application before, the answer should be Completely New. If informed consultation and review is available with people in the Area the answer should be Some Familiarity. If Services people, clearly expected to participate significantly in the proposal and project, are currently assigned to the performing Area and have recently performed on a similar project the answer may be Have Done Similar Job Once.

b. To answer Extremely Thorough the proposal should contain a technical baseline that shows excellent understanding of the tasks in the Statement of Work. The Customer User, IBM Branch, and DP Services must have contributed and concurred with the approach. Everything else should be moderate unless we lack

f. Processing complexity:
Straightforward (0),
Complex (30), In-between (15) ______

g. Exception Correction is mostly:
Manual (0), Automatic (20) ______ ______

F4

5. KNOWLEDGE WE HAVE FOR THIS PROJECT

a. How familiar is the proposed
Services Area with this Application:
Completely New (30), Some
Familiarity (15), Have done
Similar job once (0) ______

customer agreement either through lack of contact or because of direct disagreement.

6. READINESS TO PERFORM THIS PROJECT

a. Consider the location of the project with respect to the home location of the people expected to work on it. Unless local commuting habits and ground rules indicate otherwise, travel of more than one hour each way to the work location should be considered Significant Commuting.

b. Consider the proposed manning on the project. Normally the manning on DP Services projects comes from DP Services, the IBM Branch, or the Customer. If the manning is proposed with elements other than these, (i.e., subcontract or shop order) mark an equivalent answer from the viewpoints of Project Management control and the resource's ability.

c. All temporary or permanent moves of project team members should be considered whether they involve IBM people or customer people.

THE SIZE AND COMPLEXITY FACTOR COMPUTATION:

To compute the Design Phase Size and Complexity Factor that will be used to validate the task-by-task estimate follow these steps:

1. Review and sum up the weighted answers to the questions to determine factors F1 through F6.

2. Enter F1 through F6 and evaluate the equations on page 19.

3. Sum the results of (1), (2) and (3) to obtain the Design Phase Size and Complexity Factor.

ESTIMATE VALIDATION:

Use the Design Phase Size and Complexity Factor and the plots provided in Section 6.2 to determine the average number of hours that the standard tasks took on completed DP Services projects with similar Design Phase Size and Complexity Factors. Enter these hours in the appropriate blanks on page 20.

If the data is sparse, the information on each standard task may not be provided as a separate number. However, the hours spent on that task are in the totals and in the associated standard task. (e.g., the hours

b. Services Preproposal Analysis:
Extremely thorough (0),
Moderate (10), No customer agreement with
approach (20) ______ ______

F5

6. READINESS TO PERFORM THIS PROJECT

a. Where is project to be located:
No unusual commuting (0),
Significant Commuting (5),
Temporary or permanent moves
required (10) ______

b. Manning:
All Services (0), Mixed IBM Manning (5),
Customer and IBM Mixed (10) ______

c. Number of temporary and permanent
moves required ______ ______
_____x 5 =

F6

for implementation planning may not be separately identified, but they would be in the internal system design task and in the total hours.)

Map the task-by-task estimate into the same standard tasks and compare the estimates. The Proposal Manager should analyze and explain any differences or make the appropriate adjustments in the task-by-task estimate and the proposal.

FEEDBACK PROJECT RESULTS:

After the project is completed and the PCAR is available, adjust the Design Phase Size and Complexity Factor. The factor needs to be adjusted to account for changes (approved PCR(s)) that occurred during the project. This adjustment provides a factor that should be related to the completed project's results:

Original S & C -	The original size and complexity factor computed at proposal time on page 19.
Change Hours -	The total estimated hours of approved changes taken from the PCAR.
Total Hours Multiplier -	The current factor multiplier for the total hours plot in the design phase estimator.
Adjusted S & C -	The size and complexity factor used for project feedback of results adjusted for the approved changes.

$$\text{Adjusted S \& C} = \frac{\text{Change Hours}}{\text{Total Hours Multiplier}} + \text{Original S \& C}$$

The results of the completed project standard tasks and the delivered reports are also taken from the PCAR. If the project does not represent a complete design phase, the numbers must be used with care. (e.g., a requirements only design phase can give a good requirements number. It certainly won't give any design numbers. Less obviously, it won't give any management numbers or total hours numbers either).

THE SIZE AND COMPLEXITY FACTOR COMPUTATION:

1. Orientation Factor:

$$\underset{F1}{(\_\_\_\_\_\_)}\ (100 + \underset{F5}{(\_\_\_\_\_\_)} + \underset{F6}{(\_\_\_\_\_\_)})\ x\ .9/1000 = \_\_\_\_\_\_$$

2. Requirements Analysis Factor:

$$\underset{F2}{(\_\_\_\_\_\_)}\ (100 + \underset{F1}{(\_\_\_\_\_\_)} + \underset{F3}{(\_\_\_\_\_\_)}$$

$$+ \underset{F4}{(\_\_\_\_\_\_)}/10 + \underset{F5}{(\_\_\_\_\_\_)} + \underset{F6}{(\_\_\_\_\_\_)})\ x\ .6/1000 = \_\_\_\_\_\_$$

3. System Design Factor:

$$\underset{F2}{(\_\_\_\_\_\_)}\ (100 + \underset{F3}{(\_\_\_\_\_\_)}/2 + \underset{F4}{(\_\_\_\_\_\_)}/3$$

$$+ \underset{F5}{(\_\_\_\_\_\_)}/4)\ 1.7/1000 = \_\_\_\_\_\_$$

Size and Complexity Factor = $\underset{\text{Sum(1),(2),(3)}}{\underline{\underline{\_\_\_\_\_\_\_\_}}}$

ESTIMATE VALIDATION:

	From S & C Factor	From Task-By-Task	Comments
Total Hours	_______	_______	
System Design	_______	_______	
External System Design	_______	_______	
Internal System Design	_______	_______	
Implementation Plan	_______	_______	
Requirements Definition	_______	_______	
Orientation	_______	_______	
Management	_______	_______	
System Design Report Size	_______		
Requirements Report Size	_______		

FEEDBACK OF RESULTS:

Adjust Size and Complexity Factor: By:____________ Date:______

(______) + (______) = (______) Adjusted Size and Complexity Factor
(______)

Completed Project Results: By:____________ Date:______

Total Hours	______
System Design	______
External System Design	______
Internal System Design	______
Implementation Plan	______
Requirements Definition	______
Orientation	______
Management	______
System Design Report Size	______
Requirements Report Size	______

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22314	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93940	2
3.	Curricular Office , Code 37 Naval Postgraduate School Monterey, California 93940	1
4.	Dan C. Boger Administrative Sciences Department Code 54bk Naval Postgraduate School Monterey, California 93940	1
5.	LCDR John Hayes, SC, USN Administrative Sciences Department Code 54ht Naval Postgraduate School Monterey, California 93940	1
6.	Lieutenant Daniel J. Spooner, USN 124 Brownell Circle Monterey, California 93940	1
7.	Norm Lyons Administrative Sciences Department Code 54lb Naval Postgraduate School Monterey, California 93940	1
8.	Chairman Administrative Sciences Department, Code 54 Naval Postgraduate School Monterey, California 93940	1
9.	LCDR David F. Spooner, MC, USNR 6435 Wing Point Road N.E. Bainbridge Island, Washington 98110	1
10.	Carolyn Wong 2500 Colorado Avenue M.D. 32-61 Santa Monica, California 90405	1
11.	Dr. Barry Boehm TRW-DSG 1 Space Park, R2-1076 Redondo Beach, California 90278	1

12.	Rich Patrick Amdahl Corporation 1250 East Arquez Avenue P.O. 470 Sunnyvale, California 94086	1
13.	Fleet Material Support Office Code 92 Mechanicsburg, Pennsylvania 17055	2
14.	Fleet Material Support Office Code 92E Mechanicsburg, Pennsylvania 17055	1
15.	Fleet Material Support Office Code 92T Mechanicsburg, Pennsylvania 17055	1