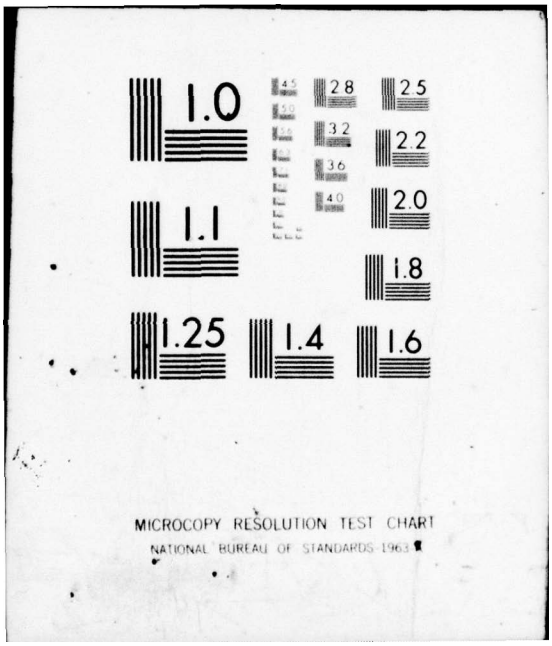
AD-A036 064 SYSTEM DEVELOPMENT CORP SANTA MONICA CALIF F/G 9/2
SOFTWARE DATA COLLECTION STUDY. VOLUME III. DATA REQUIREMENTS F--ETC(U)
DEC 76 M C FINFER F30602-75-C-0248
UNCLASSIFIED SDC-TM-5542/003/01 RADC-TR-76-329-VOL-3 NL

1 OF 2
AD
A036064


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963

12
NW

RADC-TR-76-329, Volume III (of eight)
Final Technical Report
December 1976

SOFTWARE DATA COLLECTION STUDY
Data Requirements for Productivity and Reliability Studies

System Development Corporation

ROME AIR DEVELOPMENT CENTER
AIR FORCE SYSTEMS COMMAND
GRIFFISS AIR FORCE BASE, NEW YORK 13441

This report has been reviewed by the RADC Information Office (OI) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public including foreign nations.

This report has been reviewed and is approved for publication.

APPROVED: Richard T. Slavinski

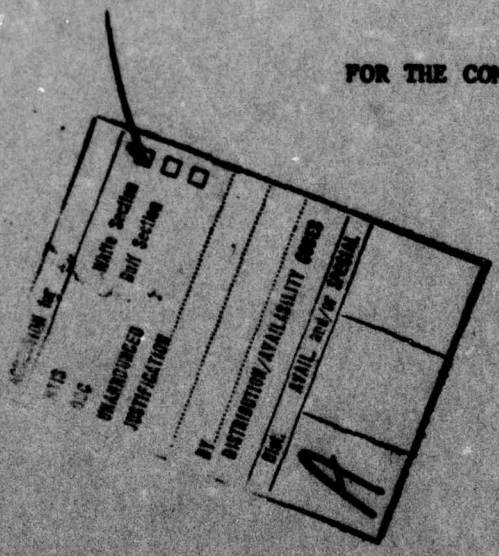
RICHARD T. SLAVINSKI
Project Engineer

APPROVED: Alan R. Barnum

ALAN R. BARNUM
Assistant Chief
Information Sciences Division

FOR THE COMMANDER: John P. Huss

JOHN P. HUSS
Acting Chief, Plans Office



Do not return this copy. Retain or destroy.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: RADC-TR-76-329-Vol-III (of eight) | 2. GOVT ACCESSION NO. 3 | 3. RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle): SOFTWARE DATA COLLECTION STUDY. Volume III. Data Requirements for Productivity and Reliability Studies.

5. TYPE OF REPORT & PERIOD COVERED: Final Technical Report. June 1975 - June 1976.

6. PERFORMING ORG. REPORT NUMBER: SDC TM-5542/003/01

7. AUTHOR(s): M. C. Finfer

8. CONTRACT OR GRANT NUMBER(s): F30602-75-C-0248 new

9. PERFORMING ORGANIZATION NAME AND ADDRESS: System Development Corporation, 2500 Colorado Avenue, Santa Monica CA 90406

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 63728F; 55580810; 98

11. CONTROLLING OFFICE NAME AND ADDRESS: Rome Air Development Center (ISIS), Griffiss AFB NY 13441

12. REPORT DATE: December 1976

13. NUMBER OF PAGES: 126

14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): Same

12 137p.

15. SECURITY CLASS. (of this report): UNCLASSIFIED

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE: N/A

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)

Same

DDC RECEIVED FEB 28 1977 C

18. SUPPLEMENTARY NOTES

RADC Project Engineer:
Richard T. Slavinski (ISIS)

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

Software Development
Data Collection
Productivity Studies
Reliability Data

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

Ever increasing costs of software development without a parallel increase in software quality has generated much attention on software reliability, project productivity and the overall problems inherent in the software development process. One of the objectives of the Data Collection Study is to recommend a set of parameters to be collected for the RADC Software Data Repository that will form an historical data base to support research and analyses requirements within RADC. Using past and present data collection systems and the results of a survey of the literature as guides, data parameters

OVER

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

339 900 LB

from contract award to software system installation are discussed.

A classification of data parameters results from the examination of significant software development factors. The first class consists of project environment factors including contract and customer relations data, organizational and personnel characteristics, hardware and support facilities parameters, and overall attributes of the software product itself, such as size, complexity, etc. The second class of data is project performance information reflecting the amount and quality of work performed for the duration of the project period. Class three data consists of automatically generated product measurements, which demonstrate the structure and behavior of the product through the application of analysis tools and testing procedures.

TABLE OF CONTENTS

TABLE OF CONTENTS (cont'd)

1. INTRODUCTION

The material herein presented is the final report for Task II, Data Requirements for Productivity and Reliability Studies for RADC Contract F30602-75-C-0248.

Data collection on software development projects is a complex process of obtaining a number of parameters at strategic points during the software development life cycle for the purpose of measurement and analyses. Comparisons of quantified data from differing projects, albeit collected in the same manner, can not be altogether valid unless the collected data is mapped to the set of conditions that uniquely produced it. It is the conclusion of this study that there are three major categories of data needed to meet the current research needs of the RADC Software Data Repository, including: 1) the project environmental data reflecting conditions under which the software is developed; 2) the progress and performance data current to the reporting cycle; 3) the product quality measurements of the software as demonstrated through testing and analysis tools.

The purpose of this study has been to determine what data items to collect with justification as to the reasons for selection. It is apparent that there may be a RADC research requirement at some time that encompasses a set of data items heretofore not considered important to the repository, but the collection process must be sufficiently flexible so as to allow collection of an expanded parameter set at a later time. The central research studies of project productivity, program reliability and software development costs have had the dominating influence in the study and selection of data requirements. Some attention has been focused on the accessibility and costs of collecting specific data parameters, but for the most part, selection of the data has been based on the utility of the parameter for the specific research needs stated above.

The data requirements are presented in a classification scheme that relates to both the research uses of the data and the content of the parameters themselves. The priority of data importance, established by both RADC and SDC personnel, is also considered in presenting the justifications for collection of specific data parameters.

2. SURVEY OF LITERATURE

Currently, world-wide attention is being focused on the problems besetting the software industry. These problems range from the tremendous costs incurred during the production of software to the growing need for secure systems for the protection of national interests as well as individuals' rights. One of the most significant problems facing the industry has always been, and continues to be, the reliability of the software produced. While software systems are growing in costs, size and complexity, and are invading all aspects of day-to-day living, the progress towards improving the reliability of the software has fallen short of the demands and needs for that reliability. Concurrently, the productivity of all phases of the software project are being discussed with the hope of finding the key to increase project performance, while developing techniques to improve reliability. When these goals are achieved, software development costs will be significantly altered.

It has been the intent of this study to determine the parameters contributing to software reliability, project productivity and costs of software production by reading past and present literature, surveying a sample of software managers, and examining established project control forms.

In the survey of the literature four principle areas of investigation were found to contribute to the conclusions derived in the course of the study. They include:

1. Productivity Analyses
2. Reliability Analyses
3. Advances in Project Performance Monitoring
4. Programming Technology Advances

Although these areas of study overlap and impact work being performed in each of the other categories, there are significant and independent conclusions to be made in looking at the progress and evolution of the work in each field. A brief summation of the findings of the literature search will be presented for each topic; many of the references will be examined in depth later in the report.

2.1 PRODUCTIVITY ANALYSES

In the 1960's several studies were made on factors affecting the growing costs of producing software. Two companies, System Development Corporation (SDC) and Planning Research Corporation (PRC), did extensive work in this field in order to determine the factors affecting resource expenditure for program development.

The purpose of the SDC study [26] contracted by the U.S. Air Force was to identify variables which were thought to seriously impact computer program development. The technique employed by SDC was to develop a questionnaire to collect quantitative data on variables affecting programming costs, and subsequently, statistically analyze the data. The collection and analyses procedures were refined over a three cycle period covering a large sample of completed software projects on each cycle. The most significant result of the iterative study was to indicate that cost analysis of empirical software experience data is a means for cost estimation using regression analysis predictor equations. It is also noteworthy that only 15 of the over 100 variables included in the questionnaire were used in the final predicator equation which can be reduced to:

$$\text{effort} = (\text{constant}) \times (\text{number of instructions})^{1.5}$$

From this study, SDC proceeded to extend the findings in a handbook for management to be used during the broader area of the life cycle of software development, including preliminary planning and cost evaluation, system analysis and design, program development, system test, installation and maintenance. The major productivity factors as determined by SDC in these studies are shown in Figure 2-1. It is interesting to note that the

YEAR	AUTHOR/ORGANIZATION	MAJOR PRODUCTIVITY FACTORS
1962-1966	SDC [38]	Stability of initial design Size/complexity of data base Frequency of operation Schedules Size of system Complexity of system Type of application Timing requirements
1968	Gotterer [52]	Communications
	Bemer [52]	Standardization
	Sackman [49]	Individual ability Environment, including computer system usage Familiarity with problem
1969-1971	PRC [52]	Number of I/O formats Size/complexity of data base Number/complexity of input records
1969	Fried [52]	Program size Number of programmers
	Krauss [52]	Program size I/O complexity Language Programmer skill
	Gill [52]	Organization
	Harr [14]	Communications
	Brophy [52]	Standardization Programming tools Training
1970	Brandon [52]	Standardization Inadequate functional descriptions Creativity syndrome Staffing shortages Personnel management Program psyche Management/Programmer training Technological change

FIGURE 2-1. SYNOPSIS OF PRODUCTIVITY ANALYSES

YEAR	AUTHOR/ORGANIZATION	MAJOR PRODUCTIVITY FACTORS
1971	Gayle [52]	Programmer experience Distance between computer and programmer Number of output formats Frequency of operation of object program
1972	Hirsch [52]	Work Environment
	Lassiter [52]	Employee morale Programmer experience Office layout
	Shell [52]	Type of programming application Programming language Computer hardware Program complexity Program size
	Weinberg [52]	Communications Programmer ego Perceived objectives
19??	Aron [14]	Complexity and size of programmer interactions with computer system
19??	Portman [14]	Estimations on productivity reflected in schedules should reflect actual hours spent in activity
1974	Wolverton [70]	Subjective and objective project dependent variables
1974	Scott-Simmons [53]	Quality of external documentation Programming language Availability of programming tools Programmer experience in data processing Programmer experience in functional area Effect of project communications Independent modules for task assignment Well-defined programming practices
1975	Brooks [14]	Complexity of task (Operating systems most difficulty; compilers second most difficult.)

Figure 2-1. Synopsis of Productivity Analyses (cont'd)

YEAR	AUTHOR/STUDY	PRODUCTIVITY MEASUREMENT PROJECTS
1969	ADPREP [52]	Resource estimations, for proposed tasks based on experience data.
	Harr [14]	Summation of productivity figures for four projects: productivity rate for: control programs - 600 words/man year; translators 2200 words/man year
1973-74	Patterson [42]	Programmer/project productivity estimations for proposed tasks based on experience data.

Figure 2-1. Synopsis of Productivity Analyses (cont'd)

work that SDC did in cost estimation techniques has been examined and reexamined in an attempt to use cost estimation techniques for proposal work, but the algorithms being used do not use the linear regression analysis equations previously mentioned.

The study performed by PRC [72] under a U.S. Air Force contract attempted to derive a correlation between development resources and independent variables. Although the PRC study was inconclusive due to the small sample used [45], a handbook was published. It attempted to provide a means by which the estimated project dependent variables were compared with existing data from 18 completed software systems in order to determine whether the proposed software system was reasonably similar to the established data base.

Both the SDC and PRC studies made a significant impact on management's view of the software development process in showing a lack of conclusive results from the wide range of possible variables affecting software expenditures. It was seen that from the number of variables examined and used in projecting costing techniques, the key variables clustered around three main categories. These groups of variables include:

1. The familiarity of project personnel with the hardware, software, and problems to be solved;
2. The environment in which the software is being produced;
3. The type and complexity of the software project.

It is exactly these three areas which do not readily support quantification and measurability. The result, therefore, was that no usable prediction techniques became established as a means for cost estimation. Almost more important than this, however, was the attention that became focused on the complex process of software development. It became apparent that since the costs of producing software were rising at an extremely rapid rate, emphasis should be placed not on cost prediction techniques, but new, more efficient mehthods to be used in the software development cycle in order to reallocate for the rising costs and/or update original cost estimations.

By mid and late 1960's, the focus of attention had expanded to include innovative techniques in program testing capabilities, conceptually better programming languages, programming and documentation quality control standards, and more efficient programming disciplines.

The interest in productivity analysis remained, however, and is succinctly presented by Scott [52]. First, Scott examines what he calls the commentary literature-articles and books based upon authors' opinions and experience dealing with the problem of program management. Second, the research on predicting project expenditures is presented, including the SDC and PRC studies. Third, the application literature which explores attempts to apply past research to management practices. The survey presents a host of variables thought by various authors to contribute to programmer productivity, as shown in Figure 2-1. The commentary literature supports the subjective and human related factors, which are not only difficult to quantify, but are also difficult to even obtain a consensus of opinion on their definition. It has been suggested by A.M. Pietrasanta [45] that these subjective human characteristics inherent in the software development process might turn out to be the most important variables. The research literature supports the concept on which the RADC data repository is based: historical data covering a wide range of software products in all phases of development is necessary before adequate research can be accomplished. The application literature reviewed by Scott was primarily concerned with attempts to apply the data gathered in previous studies to obtaining a more meaningful estimation of project productivity. The main objective of the research covered by Scott's survey was to predict project costs, rather than quantify productivity variables.

One observation made by Scott, as well as J.W. Patterson [42], is that in the process of data collection, the differing collections are incompatible because of a lack of standardization of either data items, recording techniques, formats, or purpose. Patterson did an extensive study for IBM in 1973-74. As of now, the findings of the study, and subsequent techniques for utilizing the productivity data, have not been published. However, IBM-FSD

is attempting to support proposal cost estimates with data gathered from the Patterson study. It appears that project managers contributing to proposal cost estimates are eager to have a substantive method by which to tentatively validate or invalidate their estimated costs, according to Patterson.

Wolverton [70] has presented an excellent analyses of the factors impacting large-scale software development costs, including data supporting productivity rates. In order to arrive at valid estimation techniques, a data base consisting of experience data is thought to be necessary. This data can be obtained only by analyzing objective and subjective parameters characteristic to the given software development project. Utilizing "average productivity rates" is not meaningful unless the components of the productivity rate are known; even utilizing data from large samples are only useful in estimation if the estimator is certain of "comparing truly equivalent measures".

Frederick Brooks [14] also reviews significant work performed in estimating productivity. Some of the more important conclusions presented by Brooks are:

- The actual number of hours spent by members of a programming team account for realistic estimations of schedules. It is, however, important to note, that the members of the particular team were spending only 50 percent of the work week in the programming effort.
- The more complex and numerous the interactions between programmer and computer system during the design and programming activities, the lower the number of instructions produced per man-year.
- Productivity rates ranged from 600-800 debugged words/man-year for developing control programs; 2000-3000 debugged words/man-year for developing language translator programs.
- Productivity is affected by the language in which a system is written. "Programming productivity may be increased as much as five times when a suitable high-level language is used." [14]

In summary, much analysis has been done and continues to be done on productivity rates in the software development process. The parameters that are crucial in determining productivity are, to a large extend, the same parameters affecting costs and eventual reliability of the software system. Further work is needed in establishing a historical data base of estimated and actual parameters in order to provide sufficient data for detailed analyses on productivity rates. The eventual goal should be to establish technical and performance standards and criteria for the software development process, which, when understood and applied, will not only contribute to reliable predictions of productivity, but will also improve project and individual productivity.

2.2 RELIABILITY ANALYSES

The amount of work performed during the past decade on software development aimed towards improving software reliability is staggering. Many people have written extensive surveys addressing the major techniques used in validating programs, while others have concentrated their writing on one particular aspect of the reliability field. Conferences addressing the questions of software reliability are being held several times each year, and the attendance at the majority of these conferences has been most impressive. However, software remains predictably unpredictable and there appears to be no panacea for the problems of producing high quality, error-free software. A brief analysis of the work that has been done towards improving software reliability will indicate the techniques that are available, or under development, for improving the error rate of software, and the effect these have on the proposed data repository.

For many years, the major concern in the computer world was the reliability of the computer itself. Much engineering work was directed as this problem until today, hardware failures have diminished to the point that they are not even considered in the day-to-day coding and debugging activities of the average programmer. Attention is now being focused on software reliability. The Air Force in 1972 spent over 1.5 billion dollars on software development,

of which half was used for software verification; in contrast, they spent less than one billion dollars for computer hardware equipment [56]. Techniques for predicting hardware failures by mathematical models have advanced to quite a successful state. Consequently, many people have tried to apply the work done in hardware reliability to software reliability. The basis for the hardware reliability modeling relies on observing the rate at which a physical digital circuit fails over time. This type of analysis was applied to software errors in the Satellite Control Facility (SCF) programs by SDC personnel a decade ago. Hudson [30] and Ellingson [21] presented the error-detection and correction scheme as a birth-and-death stochastic process, using the data obtained as a basis for predicting the number of errors remaining in a software system during checkout. It was Hudson's conclusion that this data could then be used to: 1) define the point where the system development process was complete; 2) determine the cost of program quality; 3) set standards for the turnover of new system; 4) better evaluate systems during the design phase to ensure production phase quality.

What then has happen to Hudson's optimistic conclusion reached after reliability modeling of the SCF programs? The SCF models are based on data collected by an independent integration and test contractor, using delivered and contractor-tested software over a lengthy test-retest process. While this method is valid, it occurs later in the production period. Earlier, more rigorous test methods are desired.

Work in the reliability field has continued with goals for obtaining: 1) a standard definition of software reliability; 2) the data variables affecting software failure rates (as well as the critical parameters to be used in the modeling process); 3) a reliability model itself. The work of Shooman [57], Jelinski and Moranda [63], Schick and Wolverton [63], Littlewood and Verrall [33] all contribute to this growing field, but as yet, the application of a single reliability model to a large number of software systems with the intent of estimating the software reliability of each system is currently not feasible.

While individuals pursued the field of reliability prediction, work was concurrently being performed in the area of formally proving the correctness of programs. This work is based on providing a set of assertions representing program functions that can be analytically verified by formal mathematical techniques. In effect, by proving the correctness of program assertions, one can prove the correctness of the program itself. Floyd, Naur, London [37], and others have made significant contributions in the area of formal program proofs. However, the application of these mathematically complex and lengthy proofs to anything but a small piece of code makes formal program proving impractical and infeasible. The assertions and the assertion language themselves become yet another error producing construct.

The development of verification and validation tools has also proliferated in the past ten years. These tools range from simple debug snapshots and traces of individual program behavior to large and complex execution analysis tools. The majority of tools have attempted to provide information on the dynamic operational behavior of the program, or to offer test case information to be utilized by the programmer during the test phase. Such tools as PACE [40], PATH [41], RXVP [48], and QUALIFIER [46] are designed to analyze the execution frequency of the program, or parts of the program in order to indicate the statements which have, or have not, been executed. The principle behind these verification tools is that they provide the user with exactly the frequency that each program statement has operated for each test case. They do not improve the reliability of the object program, but they do provide information supporting the "testedness" of the said program. Reliability of the program is indirectly improved by these programs when the user has proof that every statement in the program was executed during the successful operation of a set of comprehensive test cases.

Another set of tools are being developed that attempt to provide the user with a set of test cases that guarantee the execution of every statement in the program. While these test tools are in their infancy for wide applicability to large scale software systems, the information they may provide will greatly reduce the amount of testing applied to a program. The user will be

able to examine the data values required for execution of all statements to determine the validity of the values. In addition, the execution of all program statements with a small, non-redundant set of test conditions will simplify the testing requirement procedures, while increasing program reliability.

The analysis used in the automatic generation of test cases is also used to determine the flow of control in the program. The concept of representing the program as a directed graph, whose arcs represent the potential flow of program control and whose nodes represent the basic blocks (a set of consecutive instructions) of program code, also contributes to the analysis of the complexity of the individual program. The fact that automatic test case generation is presently not feasible for the industry as a whole does not alter the needs addressed by this type of tool. Software errors result from the failure to adequately provide for all conditions and combinations of conditions relevant to the intended program operation. A method to represent the actual program operation, and/or combination of operations, can provide the user with information for more adequately devising and selecting test data.

Also contributing to the software reliability movement has been the emergence of a set of concepts or philosophies which contribute to the overall quality of the software produced. Structured programming is largely derived from the work done initially by Dijkstra [20], but more recently it has come to represent the collective name for a specific group of techniques that contribute to the reliability of software, while providing for greater understanding and maintainability of the software. Briefly, these techniques consist of a design methodology (levels of abstraction), which is used to divide a complex software system into a logical and compact set of modules; a programming discipline (structured programming, top-down development and the elements of programming style), which is used during implementation and testing of the modules; and a management organization (chief programmer teams), which supports the design, implementation and test methodology. While it is generally believed that structured programming techniques do

improve program quality, and there is some proof to support this believe [4], these techniques do not in themselves guarantee software reliability.

In summary, the needs of the software industry for reliability not only remain but are ever-increasing. There appears to be no one solution or technique for providing that reliability. An approach which combines new tools, technology and concepts, as well as an in-depth evaluation of historical experience, is a necessary first step towards producing higher quality software systems.

2.3 ADVANCES IN PROJECT PERFORMANCE MONITORING

Although practitioners in the industry were well aware of the problems of managing large scale data processing system development projects, Hosier's [29] 1961 paper on real-time system pitfalls was the first to make a real impact on the industry. In the post-Korean war, large schedule slippages and overruns of aerospace and naval systems were causing considerable furor in both Congress and the news media [43]. In data processing, experiences with SAGE and NTDS were equally disturbing. Hosier set forth the principles derived from those experiences much in the way they are viewed today. Developmental milestones, test plans, interface specifications, developmental measurement, debugging tools, concurrency, validation, and prototype programming were all mentioned. All this management paraphernalia, plus change and problem reporting, machine time recording and reporting, and other aids to management were developed on these large scale systems. Yet, project management tools have not had general usage until recently.

The Air Force and Navy in the late 1950's adopted PERT, or other critical path scheduling techniques, for large system development projects, and the techniques were soon introduced into software development. Ewart and Nanney [24] have evaluated the use and value of PERT techniques in weapons system acquisition from the time it was acclaimed as a panacea for the ills of system development to its downgrading in importance by the recent C/SCS regulations [1]. Besides being oversold, there was considerable user resistance; a

general lack of management support for the technique; and protests at the costliness of generating and maintaining the detailed plan required, as well as keeping both ordinary cost accounts and PERT/COST accounts. For project control, there is no denying that PERT scheduling and costing are probably the best techniques available. However, there is still a great deal of subjectivity in schedule and budget estimates. PERT is a tool for planning and control; not a substitute for good managerial analysis and evaluation. On most projects for which PERT is required, it appears that the technique evolves into a reporting tool for the client, and is not really used to plan and control the project. To be effective, PERT must be used on a day-to-day basis, not updated at the end of the month or quarter as a result of other control operations. Also, it cannot be the only tool of the project manager.

In the early 1960's, the Air Force devoted a considerable amount of effort to the development of project management concepts, culiminating in the issurance of AFSCM 375-1. This manual was best fitted for management of hardware acquisition. During 1964, and subsequently, considerable effort was expended interpreting and expanding the series to cover software and personnel subsystems acquisition. For instance, ESD sponsored an indepth study of computer program configuration management (partially reflected in Searle [55]), and computer program data requirements [54] that were eventually reflected in AFSCM/AFLCM 375-7 and AFSCM/AFCM 310-1. Although much of this effort was directed at specification practices, it also covered change control, documentation maintenance, configuration accounting practices, and configuration reviews. It was in these working groups that general models of the software development life cycle were worked out, and products and reviews identified for each developmental phase.

The AFSCM 375 series was replaced in the early 1970's by MIL-STDs 490 (Specification Practices), 480 (Engineering Change), and 1521 (Review Procedures). The Air Force carried most of the provision of 375 into MIL-STDs 483 (Configuration Management Practices) and 499 (System Engineering Practices). AFR 800-14 (Computer Resource Acquisition) May 1974 (Vol. 1) and August 1975 (Vol. II) is the latest in the sequence of regulations on configuration management.

In addition to progress reports, work breakdown structures (MIL-STD 881) and other management plans are normally required on military developed projects. The work breakdown structure (WBS) ties into both project management by providing the basis for tasking, and into configuration management, by breaking the system product into smaller segments. The WBS identifiers provide the basis for both cost and configuration accounting. While a WBS helps to organize and simplify the project, it is usually at too gross a level of decomposition to provide close control, and too inflexible to reflect the realities of a developmental project.

The notion of concurrency in software development preceded Hosier's report, and despite many difficulties, is still most appealing. It is reflected in the current notion of the "Software First Machine" [8]. That is, since software now represents the major cost item and the pacing factor in development, its development should begin by first using a simulation of the end machine for initial development. Concurrency has always been present in the simultaneous development of many programs and system segments, a now popular concept in top-down implementation and integration. That is, while simultaneous development of many portions of the system demands very close attention to interfaces to assure integration later, the top-down approach seeks to control proper interfacing by implementing control segments and operating interfaces first. This does not necessarily increase the degree of concurrent testing, but permits the detection of interface and control errors early, and avoids the rework necessary to correct the incompatibilities when discovered later.

Considerable attention has been given recently to proposed "innovations" in organizing for software production. In 1965, Willmorth [69] summarized the agreements for and against various organizations. A line (or assembly line) organization takes high advantage of job specialization, but has high communication needs as work is passed from one job stop to another. Appointing a "system manager" to control funds and coordinate work expedites the process and speeds decision processes, but has only moderate impact on reducing communication needs and increasing concurrency. A matrix organization in which the job specialists are organized into skill groups, but assigned to projects (with or without relocation and shift of supervision), ensures the advantages of job specialization. This considerably reduces communication needs, but introduces some conflict of authority and control into the management situation. The project organization is conceded to have the least coordination and communication problems, but it also takes least advantage of job specification. The present emphasis on Chief Programmer Teams [4,5] seeks to ameliorate the situation by specifying an internal organization for the team and calling for highly qualified team members. However, it would seem that the team would have a greater chance of success if the internal organization remains flexible to meet the demands of the particular project. It is unlikely that highly qualified individuals will be available for all teams. Greater success is likely to accrue if the highly qualified are assigned to the trouble areas and/or highly sensitive control and interface areas.

In summary, it is now recognized that in order to provide effective management, the manager must have a set of plans, standards, and a method to determine how closely actual matches planned performance. However, there are many levels of management, and information needs are somewhat different at each level. The first level supervision needs day-to-day, or hour-to-hour information, and top level management may be satisfied with monthly or even less frequent information, except on major problems. To improve management, further information is required on the way in which management is conducted, including the techniques and approaches used.

Management techniques are most apparent in the project management monitoring, configuration management controls, and quality assurance tools and techniques employed. To evaluate these, information is required on the plans laid and the performance realized, plus an evaluation of the project environment - customer relations, project characteristics, working conditions, stability of requirements, management and personnel, and such environmental stress factors as the adequacies of personnel skills, computing equipment, developmental tools, and developmental time.

2.4 PROGRAMMING TECHNOLOGY ADVANCES

The problems of software development are complex, and the complexity and interrelationships of these problems directly and indirectly affect reliability, productivity and costs incurred during software development. While modern computing power continues to increase, and the per unit cost of that power is decreasing, the size and complexity of the automated systems is also increasing. Although the technological skills of programming personnel is advancing, the problems to be solved continue to increase in complexity, requiring the application of new techniques. The effect of this dynamically changing field is that the newly developed computer systems become obsolete before the costs of producing those systems can be amortized over a period of time. In looking at the progression of the state-of-the-art in the software field, one can tentatively conclude that much of the programming technological advances has had its impetus from the problems of growing costs related to productivity and reliability. There is growing technology in all phases of the software production life cycle. The concepts, techniques and tools used during the life cycle reflect the attempts to alleviate the problems, but there remains no one panacea, nor little hope for one.

Perhaps one of the single largest contributions to the software field was the advent and development of higher level languages. This technological advance contributes to both increased software reliability and programmer productivity. Although it is not always feasible to select the language which best fits the problem to be solved, most higher-level languages of today incorporate the following attributes which significantly impact software production:

- Machine and operating system independence
- Object code optimization
- Ease of learning, reading, writing
- Increased power (fewer instructions to solve a problem)
- Built-in functions
- Self-documenting aspects

Since 1970 use of higher-level languages has become so extensive that machine coding is the exception rather than the rule. Even more specialized languages evolved, such as time-sharing, on-line languages as JOSS and BASIC, formula manipulation languages as FORMAC, simulation languages as GPSS and SIMSCRIPT. Along with these specialized languages, there evolved the development of a set of official standards for COBOL and FORTRAN. From the development of the specific languages, there evolved some definite concepts which directly impact programming style and the use of the languages. Formal syntactic notation is a concept which not only provides a method of defining syntax without ambiguity, but allows for a link between theoretical work and its application. Formal semantic definition techniques provide concepts currently being employed in the areas of formal program proofs. User defined languages are in their infancy, but much work is being focused in this area.

Although the directions that languages are evolving in the future are not of direct concern in this study, the higher-level language technology of today does offer the user a tool that increases both his productivity and program reliability. It is to be expected that future advances in computer languages in all phases of the software cycle will further productivity and reliability. For example, work on design languages is currently being done by several people. The objective of these projects is to aid the design process by converting the functional requirement specifications to a structure that can represent the software system and the environment it requires to operate. The work is motivated by the principle that the construction of reliable software systems is predicated upon the existence of a good design of those systems.

It must be noted that the subroutine library associated with most compilers of higher-level languages has also made a significant impact on the productivity level of the individual programmer. The subroutine library is a collection of standard, proven routines, through the use of which problems and/or parts of problems may be solved. The common routines obtained from the library can be combined with other programs, depending upon the specific user application. The library subroutines are on call to be loaded by the operating system when needed. The user's need for specific routines is communicated to the operating system through the compiled code. The use of the subroutine library allows for reduced costs by the sharing of common routines, increased programmer productivity by the availability of such routines, and increased quality by the reliability of the proven routines.

The development and refinement of large multipurpose operating systems has been another significant factor in increasing software productivity and reliability. Computer manufacturers have developed operating systems that satisfy diffuse and varied requirements for a diverse set of customers. To a large extent, these operating systems are a collection of tools, techniques and functions that may be used to construct a system independently of the hardware and associated interfaces. The programmer is relieved of the necessity to know and to create many hardware/software interactions such as optimizing loads, handling interrupts, devising paging algorithms and many other supervisory, memory management and I/O handling functions that once consumed a great deal of his time. This allows the manpower resource to be used elsewhere on important application problems, while the operating system provides reliable and efficient system interfaces.

The programming methodology, including concepts, techniques and tools, used in the development of software have significantly evolved over the years until today software projects have considerable choice in how they elect to go about software production. Proper selection of programming methodology is an issue of great interest in software development today. Examination of some of the techniques that have recently been developed have disclosed

significant data points to be collected. While there is a general move to use the new concepts in the software industry, proof that they improve the overall quality of software systems does not at this time exist.

It is the task of the system designer(s) to determine the basic programming approach to be used after the functional components and interfaces of the system have been defined. Even the process of system definition has available to it a variety of tools and techniques to aid in the alternatives and trade-offs inherent in the analysis process. Often, the decision to use specific concepts rely heavily on established corporate standards, existing methodologies, and languages available for the programming task.

Traditionally, software under development has moved through the developmental phases as a unit, the product of each phase being baselined before work officially began on the next phase. Exceptions to this lockstep approach were frequently made to admit incremental or evolutionary builds, and to give priority to programs, such as programming support tools or data base tools, that were needed in advance of the rest of the system. Once program design was complete, final development of the software advanced in a "bottom-up" mode - the most primitive system routines were identified and implemented first, with control structure and interface routines implemeted later. As routines were coded, they were first unit tested, when subsequently tested in more and more complete aggregations until the system ran as a unit. Such an approach requires the development of test drivers that define the control and data structures necessary for the execution of the bottom-level modules.

A number of problems can be identified in the traditional software development approach. Some of the problems, such as the complex internal control structures and flows and the late discovery of system inadequacies and errors, helped give rise to the alternative and more current development approaches.

In contrast to the traditional approach, a concept called structured programming evolved. One characterization of it by Bratman [12] includes:

- The hierarchical development of systems
- Modularization and simple control structures
- "Good" programming practices
- Design languages and proofs of correctness

Hierarchical development introduces the notation of "levels of abstraction." That is, the concept that a system is composed of successive layers of more and more abstract operations, beginning with very primitive operations at the machine level and ending with very abstract, very powerful operations at the user level. The interface interaction between levels is controlled such that lower level operations are "hidden" from higher level operations and the flow of control, or command, constrained to interfacing levels so that no level communicates with any level far removed from itself.

Hierarchical development also implies a top-down implementation, such that the total control and interaction structure is defined from the top (the highest level of abstraction) to the bottom (the most primitive level of abstraction). The system is implemented in the same fashion; the control structure being implemented and tested before application algorithms are written; proceeding in a like manner, level by level, to the most primitive. Thus, logic and flow errors in one part of the system should be limited by the interaction rules.

Structured programming also insists upon highly modularized programs in which the interactions among modules are limited to very simple control structures. The rules for modularization includes designing for program strength and independence. This includes programming modules to contain a single function, while minimizing external interactions, plus carefully delineating the system structure as stated above. Again, ease of understanding, verification, and maintenance is sought through the organization of such simple, easily understood modules.

Although "good programming practices" and the "elements of programming style" may be practiced anywhere, they have been most closely associated with structured programming. The structured modules are paragraphed, nested, and identified in such a way as to make program structure overt and the modules' functions easily grasped. Commentary and naming conventions strive to introduce understanding through much mnemonic and symbolic content.

Structured programming has also striven to extend language capabilities, both by normal programming language extensions and the introduction of design language and verification language features. More precise specification languages, coupled with verification features, promise much in the way of reliability, although most of these claims have not been substantiated. Language extensions, largely to reveal program structure and delimit control, holds similar promise.

Brooks [14] indicates that structured programming, specifically top-down implementation, improves reliability in many ways, including:

a. The structure supports a precise definition of the functions.

b. The partitioning and independence of modules of code contribute to clarity during test.

c. Excessive detail is missing and, therefore, design errors are more easily recognized.

d. Testing of completed levels can start in parallel with implementation of lower levels, allowing for early design error detection.

Another programming methodology, the software engineering approach, has been defined by Book, et al [9]. It is the application of the accumulated knowledge of the data processing sciences to the construction of computer program systems, especially in terms of creating optimally cost-effective systems using standard components and modular designs. The methods advocated by the software engineers stress reliability, management control, automatic engineering of design, and, of course, the structured programming approach to

developing additional modules. Since there is an emphasis upon standard components, equal stress must be placed on writing programs for maximum transportability and generalization. In the long run, the advantages of having transferable, general purpose routines that are easily modified and maintained ought to be more cost effective. This opinion is held dispite increased execution times, less machine-dependent optimizations, and increased developmental costs of individual modules in making them general and transportable.

In its detailed recommendations, software engineering differs little from structured programming. There is, however, a much stronger emphasis on standardization, system simulation, design and/or specification languages, and other system-oriented techniques than in the more limited programming orientation of the structured programming advocates.

The effort to specify the characteristics of languages and their processors might be considered an offshoot of either structured programming or software engineering. However, while these approaches take a more or less traditional approach to the software system life-cycle, the HOL approach seeks to short circuit most of the developmental cycle by going directly from a statement of a problem (requirements specifications) to a finished program. This is a most difficult proposition, and perhaps best approached through simulation programs that may be later polished for real world use.[1] On the other hand, steps are being taken toward higher level languages as evidenced by the Very High Level Language Symposium in March 1974 [62]. The features advocated there included:

- Associative referencing (accessing data on some intrinsic property of the data)
- Aggregate operators (operations over entire arrays or structures)

[1] At a recent symposium, Mr. John Lawson of Texas Instruments disclosed such a system, but no published data on the system exists at this time.

- Elimination of arbitrary sequencing (sequencing based on data dependencies only, for instance)
- Nondeterministic programming and parallelism
- Pattern directed structures

Such systems have a long way to go before attaining a production status, but the promise is there.

Some sort of compromise among the competing approaches is sought by an approach like that described by Bratman [13] for the SDC Software Factory. This concept assumes a software engineering approach to software development, complete with structured programming methodology, but proposes that a standarized facility be used for all software development with cross-compilers used to bootstrap developed programs onto target machines. The Software Factory encompases standardized methods and procedures, many programming tools, and program production libraries as an inherent part of the Factory. Standard production facilities may be the wave of the future, and may provide many production economies and efficiencies, but these remain to be proven.

In conclusion, in looking at the state-of-the-art in software technology, the industry is moving toward a more standard, engineering approach to software development. In short: the costs of producing software, the programmer productivity, and the software quality have been most impacted by a collection of concepts, which form an organized and standard base from which to proceed in the software development process. It is in the application of accumulated knowledge of the data processing industry to the specific problem that optimizes the elements inherent in software production.

2.5 SUMMARY

After surveying the literature, analyzing management and resource utilization control forms and procedures, and questioning experts in the field of software development, the summation of the results of this work is as follows:

The data to be collected sufficient to support future analysis of reliability, productivity, and cost studies are of three types. First, there are the project environment and subjective data relating to factors which directly and indirectly influence human behavior and performance during program production. There must be an attempt to systematically collect and weigh these variables in the process of determining which factors truly impact productivity, reliability, and costs within the project specific environment.

Second, there are the data necessary for the proper management of a software development project of varying size and objectives, *including the objective real data relating* to all phases of software development. These data points are detailed and extensive, and must be collected in such a manner as to be representative but not overwhelming in their volume. A technique for reducing the volume of data obtained must be established before collection and analysis of both estimated and real data can be accomplished. These data must aid the project manager in his role of ensuring that the performance of the project meets the contractually specified objectives within the allocated resources with proper regard to configuration management and quality control.

Third, there are the data obtained from test tools, aids, error procedures, etc., that are used by project personnel to determine the quality of the emerging software product. These data can also be used to support specific analysis on productivity, reliability and costs depending upon the current objective of the repository. As the objectives of the repository evolve, more or less of the product quality measurement data may be obtained.

The data points analyzed by this study for the proposed repository should be viewed as an evolving set, subject to addition or deletion as analyses to their overall validity are performed. The data repository must become a means for standarizing collection criteria in order to relate uncommon data to common analyses. At the same time, however, the repository must remain dynamic to the extent of discontinuing collection procedures and collected data that proves to be obsolete or useless during analysis.

3. ANALYSIS OF PROJECT ENVIRONMENT ATTRIBUTES

This section will discuss project specific factors affecting all software develoment processes. The classes of data include:

- Customer/Contact Considerations
- Project Organization/Personnel
- Hardware Equipment/Computer Facilities
- Software Project Attributes

The data parameters that are discussed in this Section include estimated/actual values, and objective/subjective values. A major intent of the discussion is aimed at recognizing the human factors element involved in the software development process, and subsequently, attempting to quantify those elements inorder to provide data for future analysis and study in this area.

3.1 CUSTOMER/CONTRACT CONSIDERATIONS

Little attention is given to the overall effect of the customer/vendor relationship and its effect, directly or indirectly, on the individual programmer and/or the working environment. Generally, the vendor is so eager to contract to produce the software system that the customer interactions are not considered during the proposal phase. Scheduling and costing are predicted based on established in-house productivity algorithms, without regard to the extra liason with and accountability to an external customer.

Examinations of the contractual committments should be made prior to the actual signing of the contract. Many proposals are written without an in-depth examination and study of the actual end-times specified in the RFP (request for proposal). Many of the issues which should be addressed in the proposal phase are shelved until contract negotiations, or later, because of the nature of competitive bidding. Anderson [71] has termed bidding a "liars' contest", based on what he has observed in the defense industry. His opinion is that bidders agree to whatever the government has decreed as to schedules, costs and configurations, and that real costs and impending problems are discussed only when "the time is right" for those requests, such as a cost overrun or schedule delay.

Once the contract has been awarded, the contractor has the responsibility to deliver the product within the cost, schedule and performance requirements established in the contract. However, unless all conditions are properly established in the contract, the contract itself becomes a source of problems for both the customer and the contract management. Renogotiations on contract items are both time-consuming and costly.

3.1.1 Types of Contracts

Most common federally negotiated contracts fall into one of several standard classes. Special clauses and provisions make almost every contract unique. Although the project manager must understand his contract in its entirety, even complete familiarity does not indicate all the factors that may impact software production.

For instance, the type of contract awarded does not necessarily fit the work to be performed. There has been a move away from awarding cost plus fixed fee contracts for research and development software projects. Despite the high risk and uncertainty involved, industry attempts to make the contract fit the work to be performed. Often, the job is not compatible with the funding, and overruns become inevitable. If the conditions of the contract do not fit the work being performed, the effect may be that the vendor is penalized instead of rewarded upon completion of the work.

A brief description of contract types follows.

Fixed Price Contracts

1. Firm Fixed Price (FFP)	Price is set initially and is not subject to any adjustment. The contractor assumes maximum financial risk, and all profits and all losses are his. This type of contract is used where prices are established at the outset. Requirements are usually measurable and definite, and little innovation is required.
2. Fixed Price with Escalation (FP-E)	This type of contract provides for upward and downward revision of the stated contract price due to certain defined, measurable contingencies. This type of contract is used in cases where contract cost elements (such as labor rates, material costs, or component prices) are likely to be unstable over an extended performance period.
3. Fixed Price Incentive (FPI)	This type of contract provides for the adjustment of profit and contract price by a negotiated cost to target cost formula. The contractor may share in cost savings by higher profits, or may be penalized by over estimated costs, that can end in a loss. Other incentives may also be contractually specified which alters original cost estimates when savings are shared with the contractor.
3a. Fixed Price Incentive Fee (FPIF)	At inception of FPIF, estimated costs, profit price ceiling, and formula for sharing costs over and under estimation are established. This type of contract is used when there is a modest degree of innovation and the contractor's assumption of a degree of cost responsibility will give him a positive profit incentive for effective cost control and contract performance.

Cost Reimbursement Contractors

1. Cost	Provides for reimbursement of contractor's allowable costs, with no fee. This type of contract is usually used in research and development work with nonprofit institutions.
2. Cost Sharing	Provides for reimbursement of an agree-upon portion of allowable costs, with no fee. The contractor bears part of the costs. This type of contract is used for projects jointly sponsored by the government and the contractor, with other benefit to the contractor.
3a. Cost Plus Incentive Fee (CPIF)	Provides for reimbursement of allowable costs with provision for adjustment of fee in accordance with the relationship of final cost to estimated cost. At inception, maximum fee, minimum fee, and formula for sharing costs over and under the estimations are established. This type of contract is used primarily in development, where an estimated cost and fee formula can be negotiated that will provide the contractor with a positive incentive for effective management. This type of contract usually involves some amount of innovation in the work to be performed.
3b. Cost Plus Award Fee (CPAF)	CPAF and CPIF contracts are similar. In both contracts the amount of fee is based on how well the contractor performs. In the case of CPAF, a board of review determines how the contractor is doing and awards a variable amount of fee over some base fee.

Cost Reimbursement Contracts (cont'd)

4. Cost Plus Fixed Fee (CPFF)	Provides for reimbursement of contractors' allowable costs and payment of a fixed fee. These contracts are usually awarded for research, preliminary exploration, or study where the level of effort required is unknown.

Miscellaneous Contracts

1. Time and Materials (T&M)	Provides for payment of labor hours worked, plus the cost of the materials used. This type of contract is used where it is not possible to estimate the duration of the work and/or where the government feels surveillance and control is essential in the performance (This type of contract is generally used only when no other type of contract will serve, e.g., consulting work.)
2. Labor Hour	Same type of contract as T&M, except no material cost is included.

Wolverton [70] indicates that incentive fee schedule types of contract must be understood by all people performing work on that contract, and they must be conscious of the contribution they make in order to affect the fee that can be realized by the contract company. If either the incentive fee structure is misunderstood or not recognized, there is little chance of success in obtaining that incentive fee.

While the definition of the contract terms may be explicit, the terms are often agreed to by contract or legal people who have a limited knowledge of the software development process. The project manager may have a different opinion of the contractually specified statement-of-work than does the contract negotiation team. Or, the project manager may not even examine

the contract after award, and remain ignorant of the conditions of performance or deliverable end-items until unpleasantly surprised during the performance period. Even where the project manager is cognizant of the contractual committments, these are rarely communicated to the persons responsible for performing the work. To evaluate the actual contract constraints and/or requirements on the development of the software, the following data supports analysis:

- Overall quality of funding for contractual committments, including requirements for research development on fixed price, inconsistencies or lack of specificity on deliverable items, and provisions for redirection of effort without additional funding.
- Overall quality of stated specifications, including impossibility of performance, ambiguity, omissions, conflicting provisions, disputed interpretations, and mutual mistakes.
- Evaluation of customer supplied data and/or equipment, including completeness of manuals/directives/data, timeliness of equipment/data, and defective equipment/data.
- Evaluation of customer's method of approval or review plans, including overly stringent review criteria, reasonableness of review/inspection process, timeliness of review action, and penalties imposed due to non-compliance or disapproval.

It is important for the project manager to be aware of the direct or indirect results of the customer's action or inaction on contractually specified end-items. Results may include:

- An unpleasant working environment when the customer-vendor relationship is strained.
- Forced application of more manpower to the project, impacting the planned vs. actual manpower requirements.

- Resources allocated must be renegotiated because of escalated expense, schedule slippage, and higher costs of material services, and rentals due to customer imposed restrictions, changes or approvals.

3.1.2 Questions of Risk

Both the customer and the contractor assume some degree of risk in the procurement and development of software. At this point in the technology of sc ware development, there is no scientifically proven method for producing reliable software, and there is no amount of legally specified end-items that will guarantee that the customer will indeed receive exactly what he thinks he has contracted to buy.

The risks that both the customer and contractor must consider when purchasing and producing software products and/or services include:

a. Direct and indirect risk resulting from failure in current technology.

b. Direct risk of nonperformance or delayed performance in contractor's organization.

The data processing industry itself has not defined what falls into the category of technological risk. It does appear that the industry needs a technical and legal framework which defines the questions of technological risk in an evolutionary industry. The impact of technological risk cannot at this point be precisely evaluated by customer or contractor, and the effect of this problem on the production of software is uncertain. Perhaps the conclusion to be made is that there should be open recognition of the problems involved in taking risks of this nature, or that costs incurred for high risk projects be shared by both customer and contractor in a more equitable manner.

The number of software houses which have survived such problems as technological risk are few, and the trend towards establishing yet new software-producing companies has significantly decreased. The move away from large numbers of people concentrating on a coordinated effort to a small group of people solving the same problem was one approach to solving the problems of the management of large numbers of people, but it did not solve the problems in the technology.

Obviously, there is nothing inherently wrong with taking technological risk by either customer or contractor. These risks become a problem only when either party is unaware or unwilling to acknowledge or share the risk, and take contractual remedies for the possibility that the contract will not be performed as specified. The relationship between the risk involved in the contract and the effect it has on the contracting team appears to be most significant when the size of the contracting team is small enough to be directly influenced by the customer's satisfaction, or the contract has not limited the liability of the contractor or customer in some way.

3.1.3 Subcontracting in Software Development

It has often been stated that communication is one of the key elements affecting productivity in the development of software. If, indeed, this is true, the concept of subcontracting portions of the software development to one or more contractors, or being in the position of a sub-contractor, magnifies the communication problem, the management and control problems, and the other contractual considerations previously discussed. The dependency relationships existing between sub contractors and contractors impacts not only the human factors element but actual schedules for hardware and/or software deliveries. In many very large program development projects, the government has had a policy of awarding contracts to a number of organizations in order to distribute the funding, as well as technological resources. Although this policy has obvious benefits, the problems of communication, interface, competition, etc., will remain in the industry and should be examined.

A positive aspect of sub-contracting work directly relates to costs. Robert Patrick, consultant, addressed the Special Interest Group on Software Engineering (SICSOFT) in September, 1975 on the subject of common myths in the development of software. One point made by Patrick was that although sub-contracting software development appears to be an expensive way to produce programs because of profits, burden, etc., in the total cost, it may be less expensive than in-house programming where costs can be hidden, diverted, or at least, are not always carefully scruntinized.

Another advantage of sub contracting work is that often the requirements for visibility of project progress, quality, etc., are more stringent between contractor/subcontractor than for work being performed in-house. This increased visability contributed greately to the overall quality of the emer ing product. The impact of sub contracting work in data processing is unknown and it does need to be evaluated. At a minimum, the number of sub contractors, their responsibilities and their experience in the specific area of responsibility should be examined in relation to the entire development project.

3.1.4 Customer Data Processing Experience

The familiarity of the customer with the software development process is important for a number of reasons. The quality of the interaction between customer and vendor is demonstrable from the RFP to the acceptance testing. The more knowledgeable the customer is with the software development process in relation to the problem to be solved, the more likely it is that the customer will get what he has contracted to buy. Also, the customer's experience and knowledge can directly relate to the reliability of the system under development by the quality of the system requirements, the resource documentation, and official approval criteria. Perhaps equally important are the customer's specified standards for programming and documentation, as well as the technical support and direction, especially in defining system or acceptance test cases and procedures.

A human factors aspect of the customer's impact on software production is the rapport that is established between customer and contractor. Again, government agencies have established a policy, or contracting strategy albeit not official, that work is contracted to companies or organizations which have proven to have performed in an amiable as well as satisfactory way. The relationship that exists between the customer and contractor becomes a real and viable resource. (Although there are dangers of the contractor taking advantage of the existing relationship, the competitive nature of software contracting tends to ameliorate this situation.) The interactions between contractor and vendor can range from being strained and hostile to honest and openly communicative. In any case, the existing contractor/customer relationship may influence directly the reliability and indirectly the productivity of the software developed.

3.2 PHASES OF SOFTWARE DEVELOPMENT

The life cycle phases of software development were previously discussed in Volume I of this study. Further delineation of the phases will not be presented in this section, except to define the phases as consisting of the following:

- Preliminary Analysis and Feasibility
- Requirements Analysis
- System (or Preliminary) Design
- Program (or Detailed) Design
- Program Implementation
- Test
- Installation and Operation

The subdivision of a software development project into distinct phases is necessary for a number of reasons. First, allocation of resources, such as manpower and computer time have differing requirements during the software development process. Second, configuration control procedures require that

the developing product be baselined after distinct milestones. Even when configuration control procedures are not a requirement for the software project, quality assurance provisions, such as design and performance reviews, provide management with a mechanism by which to evaluate the emerging product. The subdivision of phases differs to some extent within all development projects. Although there are many opinions as to the optimum time allocation for each phase, the collection and examination of data of this nature may provide a definition of effective phasing and possible alternatives substaniated by a reservoir of historical data.

At a minimum it is necessary to collect data sufficient to identify the total elapsed time of a software project, and the allocation of the total time to each of the software development phases. Resources allocated and expended during the performance period are discussed in more detail in Section 4.

3.3 MANAGEMENT ORGANIZATION STRUCTURES

The organization and management of the personnel involved in the software development process remain one of the most complex problems of computer software development. The impact that the structure and people of the organization have on the productivity of the project, the reliability of the software, and the costs of development is undetermined at this time. Specific data parameters need to be collected that represent a wide range of differing organizational structures and personnel attributes in order to determine more effective management techniques and/or identifying and training individuals for data processing needs.

The management of computer system development is not a simple problem of cause and effect, but a complex function involving multiple variables, all of which have direct and indirect effects on each other. Effective management is closely tied to the manner in which the available resources are organized and deployed in solving the software developmental problem. Hence, management effectivity must be considered in relation to:

a. The system of people developing the product

b. The system of hardware and software tools and facilities utilized in the development process.

c. The time and funds committed to acquiring and utilizing the above.

There are a number of contributing factors which make the management of computer programmers and the software development process as complex as it is. Brandon [11] lists the following key problems:

- Ineffective and/or inadequate technical training
- Problems of communication
- Problems inherent in the management of "creative" personnel
- Lack of uniform management procedures and performance standards
- Shortage of competent personnel

This list can be extended by Clewlow [17]:

- Problems of determing manpower effectiveness, as well as general organization structure effectiveness.
- Evaluation of individual capabilities

To a large extent, tentative management solutions to software development problems have centered around a joint effort to develop management tools and disciplines that can be used to train personnel and to establish quality standards for project control and performance evaluation. (One major aerospace company is reported by Brandon [11] to be spending well over $300,000 for basic standards of this nature.) Little work has been done to develop objective measures of individual productivity, performance, capability or effectiveness within the federal structure because it lacks a set of "meaningful quantitative standards". Subjective personal judgment remains the prime means of evaluation [17].

As the size and complexity of software system increases, the factors influencing effective management and measurement increase. The ability to recognize the impact of adverse critical factors on the production process, and take action to adjust cost, schedules, and other plans contribute both to the human factors element and to the eventual quality of the end product. The effect of this management, which depends upon individual acumen, analysis and action, may never be effectively measured and analyzed.

The purpose of a management organization is to establish procedures and standards which allow for the monitoring of technical progress and resource expenditures, while ensuring that the contractually specified end-items are produced in the elapsed time period allowed. There are a number of organizational structures used to fulfill the responsibilities of management; all of which are designed to reduce the total amount of individual communication and coordination required, while establishing a responsible division of labor. The methods to obtain this division of labor are numerous, but the principle methods of organization include:

a. Project Organization. This type of structure consists of a manager who subdivides his responsibilities into project specific components. As projects are initiated or completed, reassignment of personnel to new projects is necessary.

b. Matrix Organization. This type of structure may consist of one of the following alternatives:

   1. A project manager who obtains the technical personnel required to perform his specific project work from an established organization, which consists of personnel with a varied mixtures of skills. In effect, this organization becomes a project organization until project completion, at which time the personnel return to their original positions.

2. A program office which, in effect, sub-contracts work to the appropriate technical personnel. In this case, the program office role is that of monitoring and integrating the functional components of the line organization while excluding other managerial responsibilities.

c. Line Organization. This type of structure consist of subdividing responsibilities by functional specialization. The mechanisms for division of work can reflect the type of application, the type of internal function, or employee skills and disciplines.

Data reflecting the type of organization, the organization's functional responsibilities, the number of units and sub-units in the structure, the number of people per unit, and the personnel skill level will support analysis on the effectivity of differing organizational structures representing the software industry. These data should be detailed enough to allow at least a partial assessment of an optimum structure for software development projects. Although there remains a basic problem of attempting to quantify and qualify unlike management characteristics and practices in order to obtain standard measures, data collection forms should provide a mechanism by which to capture a wide variety of differing organizational structures. A historical file of data of this nature will support research on organizational issues described in more detail below.

3.3.1 Project Organizations

From the experience at SDC and at other software development firms, the project organizational structure appears to be the most efficient means for management of the systems of resources present in software development. It has been found that there are a number of advantages to this type of structure, encompassing both human factor elements and real, measurable data. Some of these factors include:

a. Mission orientation. Assignment to a specific project often gives the project member a sense of mission and a set of specific goals with which he may identify. Morale, efficiency, and willingness to work over and beyond normal working hours and conditions may result.

b. Ease of Communication. A project organization tends to reduce formal communication requirements while improving the ease of information flow, and the currency and quality of the information passed.

c. Problem-solving Continuity. The project organization supports the continued use of concepts and approaches needed to solve the problem throughout the development process.

d. Skill Composition Team. The project team theoretically contains all the skills necessary for the development of a software product. While this is not always true and consultations with experts may be expected on high technology projects, the same situation may arise in other diversely organized departments.

The chief programmer team concept [4] has been one attempt to solve the problems of organizational structure of software projects while supporting project organization. In effect, it represents the management structure paralleling modularity of program design. Specifically, the concept includes a structuring of job assignments by individual specialization with a clear definition of the relationships existing between team members. The team is headed by a highly competent chief programmer, whose principal job is to

design, code and test the critical segments of the code. At the same time, he designates specific program stub assignments to the rest of the team members. A backup programmer, also well qualified, assists the chief programmer in the design of the program; and, generally, acts as an evaluator, but is not held responsible for the code. A program secretary is also included in the chief programmer team concept. This person has the responsibility of maintaining the project records, project notebook, and the program production library (PPL) in both the hardware facility and the listing records. Other programers and analysts are members of the team, and perform duties as designated by the chief programmer. The entire team usually consists of five to nine people.

One of the prime ingredients in the chief programmer concept is the visibility into exactly what work is being performed by whom. The software produced by the programming team is the public property of all team members. The quality of the work performed is the responsibility of the team as a whole.

The chief programmer team concept differs from traditional organizational methods by the placement of the individual programmers who are generally placed at the base of the management pyramid. The chief programmer teams can be insitituted as a part of this traditional structure by hierarchically structuring teams according to experience and/or compentency, or difficulty of assignments. In this structure, each team reports to a higher-level team, with the highest team reporting directly to the project manager [12]. Another major difference in chief programmer teams is the emphasis upon communication between team members. In the traditional organization, the assignment of a group of individuals into distinct problem and/or system oriented subdivisions may have the effect of limiting the flow of information between people with separate task assignments. Perhaps an even more important contribution to open communication in system development is the emphasis of shared responsibility and public programming existing within the team. Data reflecting the psychological factors involved in the chief

programmer team concept are difficult to obtain, but assessment is necessary. The chief programmer team concept without open communication, public programming, and shared responsibility may be nothing more than the programming units found at SDC on the SAGE program a decade ago.

3.3.2 Line Organizations

The line organization is based on the principle of work specialization and simplification in a technical or engineering environment, matching the formal disciplines and training of its personnel. Functional specialization breaks a job down into relatively small tasks, and permits the development of a high degree of skill in the task area specialty. Since technology is concentrated in an area, it encourages the development of techniques and tools in that area.

Line organizations appear to work well as long as a reasonably standard product is produced, and similar operations are performed in every instance of development by the specialized unit. The line organization becomes inefficient when the development of a product is accompanied by a degree of speciality, and relearning is necessary in order to perform the development task. Also, this organization is inefficient when a great deal of intergroup coordination and communication is necessary to preserve the integrity and continued compatibility of the product. Certain functional specializations have proven very efficient for system development, such as machine operators of all types, e.g., typists, keypunchers, computer operators. As long as independent program modules are produced, the programmer analyst who designs, codes and tests a unit of code representing 400-1200 source statements and four-size manweeks of effort is an effective specialization. However, in the concurrent production of many interacting modules, the need for communication increases and the product team (which may still be a unit in a functionally specialized organization with sub-specializations as in the Chief Programmer Team) may be more efficient.

3.3.3 Matrix Organizations

In effect, the matrix organization is a compromise between the intergrated, but short-lived project structure and the stable, but communication-bound line organization. Matrix organization seeks to take advantage of the high level of technical development and speciality skill of the line organization by retaining a "home" organization based on skill, discipline, or work speciality. At the same time, it attempts to retain mission-orientation, ease of communication, continuity of personnel, and flexibility of composition of the team. The "full-matrix" structure, which is a relocation of personnel into project organizations, permits the workman to retain identification with his primary discipline or speciality. The "semi-matrix" structure, which is a project office overseeing the efforts of line organized personnel, is more like the line organization. It places the burden of coordination and communication on the project office, and appears to work well only when the project office has full control over funds, technical direction, and quality assurance for its product. Lacking this control, the line organization involved may disregard the authority of the project office, assign its own work priorities, and use its own product standards without fear of reprisal.

3.4 PROJECT PERSONNEL

Programmers have often been thought of as a special breed of people, for any number of reasons. There is some evidence to support this contention, although if one looks at individual specialization across the country, indices on uniqueness can always be derived. Ershow [23] presented three reasons why he felt programming to be "the most humanly difficult of all professions involving numbers of men". These reasons are worth examining in the light of the following discussion:

> "• Programmers constitute the first large group of men whose work brings them to those limits of human knowledge which are marked by algorithmically insolvable problems and which touch upon deeply secret aspects of the human brain.

- A programmer's personal pushdown stack must exceed the depth of 5-6 positions, which psychologists have discovered to characterize the average man; his stack must be as deep as is needed for the problem which faces him, plus at least 2-3 positions deeper.
- In his work, the programmer is challenged to combine with the ability of a first-class mathematician to deal in logical abstractions, a more practical, a more Edisonian talent, enabling him to build useful engines out of zeros and ones alone. He must join the accuracy of a bank clerk with the acumen of a scout, and to these add the powers of fantasy of an author of detective stories and the sober practicality of a businessman. To top all this off, he must have a taste for collective work and a feeling for the corporate interests of his employer."

If indeed, programming is such a difficult task, managing the system of people involved in the production of the task is equally difficult. Several factors are involved in the data processing people problem; the problems recommended for study by the data collection effort in order of importance include:

- Size of Project Staff
- Project Staff Experience
- Turnover Rate

3.4.1 Size of Project Staff

As the phases of software development are characteristically distinct with regard to functions performed, the requirements of manpower for the life cycle phases are also distinct. Manpower requirements build rapidly from the initiation of the project to module level/integration level testing where they more or less plateau at a constant level. Although the manpower

requirements are well recognized in the software industry, the total number of people required to do a given job appears to be a significant factor in both the manpower requirement build-up and the productivity of the project staff.

Cammack and Rodgers [15] have identified two factors that impacted productivity when a project was rewritten with improved programming and management techniques. The first and most important factor, according to Cammack and Rodgers, is the size of staff and schedule commitments. If the schedule demands a rapid and high level build-up of personnel, "productivity will suffer." The second factor is concerned with the learning curve of the personnel, a factor that every industry must recognize and deal with. There is a definite learning curve with each particular software application, which may be further complicated by a new programming process, and/or with the organization and methodology of the individual software project. When the software development project requires a large staff from onset, the time period necessary for the staff to become a productive unit must allow for the acquisition of qualified personnel, and the learning curve of each individual, which together forms the learning curve of the project.

A further consideration is the amount of training time required from the existing staff to bring the new comers to an anticipated productivity level. On most projects where a "buddy-system" is used to guide and instruct a new man, it is estimated that approximately 20% of the experienced person's time will be consumed in instructing each new person. (Obviously, more time is required initially but is then decreased as training progresses). Even if a separate training staff is employed, some "buddy-training" will remain, and the relatively "hands-off" training may stretch out the learning curve.

An observation made by Odgin [39] directly relating to project size was that the reasonable limit for manageable units was about 30 man-years of total effort. "Beyond this point the performance of personnel, managers and the infosystem itself deteriorate with startling rapidity." Besides the man-year effort, there is a critical staff-size of about 20 people, after which managerial control becomes impossible. The number given by Ogdin includes analysts, programmers and other system development personnel.

Peitransanta [44] claimed that "a linear increase in the elements of a system is accompanied by an exponential increase in the potential interfaces between the elements. Since system development requires both the development of elements and of interfaces, then one should expect a nonlinear increase in man-years as systems grow larger." The people involved in the system are one of the elements; the more people, the greater the exponential increase in interfaces between the people and the programs they are writing. Willmorth [68] noted that the process of developing a software system is an "interdisciplinary task, involving people trained in mathematics, engineering, human factors, and other disciplines depending upon the nature and purpose of the system". It is clear that as the task grows, so do the need for communication, coordination, and skills of the people grow. In order to evaluate the importance of project size and the effectivity of staffing requirements experience, the accumulation of data reflecting estimated and actual staff loading for each phase is necessary.

3.4.2 Project Staff Experience

Computer programming is a human task that has become an increasingly more demanding skill, requiring knowledge of complex machinery, abstract reasoning, and specialized tools, languages and techniques. The training offered to people interested in the profession ten to fifteen years ago does not resemble the computer science courses now offered in educational institutions of all levels today. Experience in the profession has become a much valued resource, but even that experience is not always applicable to the one-of-a-kind systems being engineered, designed and programmed today.

There are several considerations to be discussed when analyzing the data necessary to present the staff experience of programming projects. The evaluation of individuals is a human activity, filled with human fallibilities. Personal likes and dislikes, prejudices, measures of competency, and subjective performance evaluations all contribute to the problems of measuring the individual's worth to the programming project. Besides the lack of a common denominator for measuring individual skill in the software industry, the educational backgrounds of data processing people only contribute to the confusion of the problem. When programming came into existence, it attracted people from all walks of life, trained (or perhaps, untrained) in totally unrelated fields. Very many of these people were educated in data processing by the individual hardware and software firms after a cursory data processing test was administered and found to support a firm's acceptance criteria. As both secondary and college-level schools began offering courses related to the data processing field, formally and theoretically trained programmers and systems analysts began infiltrating the software industry. It has been said, however, that while the educational background may contribute to an individual's skill, the hands-on experience in data processing and in the specific application area are probably the two most important considerations in evaluating project personnel.

Until such things are unanimously defined and accepted by the industry as a whole, company assigned personal titles and salary grades lack uniformity and cannot be well utilized as a data point for demonstrating project personnel experience.

Because of the special demands of the management task, it would appear that experience in the field of management would greatly contribute to its overall effectiveness. This conclusion, however, has not been substantiated with real data. One possible reason for the lack of data is that the computer industry and technology have developed so rapidly that there is always the need for the manager to expand his awareness, involvement, and planning. The introduction of new ideas, and the actual implementation of those ideas, requires a great deal of skill, to say nothing of risk to the project manager if those ideas prove ineffectual or costly. Data supporting

the individual manager's total number of year's experience in software development, the manager's number of years experience in the management of of software development personnel, and the manager's number of years experience in the specific type of development project is necessary to attempt to evaluate the components impacting management effectivity.

In summary, data reflecting levels of experience in the data processing field, and the specific application area must be collected on management, systems analysis/design, programming and system test personnel.

3.4.3 Project Personnel Turnover Rate

It has been estimated that a project may expect a turnover rate from 20-30% per year of personnel, taking into consideration transfers, departures, and promotions [68]. (This percentage turnover rate has been experienced by large software develompent projects; data supporting analysis of the turnover rate of small software projects has not been found.) The turnover rate is an important consideration in both the estimation of resources and the organizational structure of manpower within the project. For example, one effect the turnover rate has is escalating software development costs by the need for continuous training of new personnel in the specific application area, in addition to time and money spend in recruitment procedures.

There appears to be an even higher turnover rate within the government sector of the data processing industry. Diesen [19] gives several reasons for this phenonmenon. The transfer of personnel from one government agency to another does not result in a loss of accrued benefits, such as seniority, vacation, retirement funds, and sick leave. The effect of the turnover on the individual software development project is the same, however. Departures due to retirement in private industry are just beginning to have an impact because of the youth of the industry and the age of the people it attracted in its infancy. This is not true for the Federal government employees for two reasons. First, it was possible to transfer into a data processing agency, without changing positions in stature. Second, retirement eligibility is based on length of service, resulting in the possibility of retiring at a relatively early age.

Turnover has always been recognized as a factor in productivity and product quality. Each time a key person vacates a work assignment some information is lost or must be relearned. Part of the emphasis upon documentation in the software industry is aimed at combatting the ill effects of turnover. That is, turnover indirectly increases costs by demanding that more documentation be done. If a person leaves an undocumented, or poorly documented, module behind, it is often more cost effective to recode the module than to try to interpret the old code.

Turnover in managerial and/or other key personnel may be especially critical events. A decision-making position left open for any length of time delays work because the decisions are not made. Even when the position is filled, a person new to the project may make some poor or uninformed decisions. In such a case, there is usually a period of inefficiency. Certainly, turnover rates of such persons may be an index to project performance.

In short, estimated and actual turnover rates for project personnel at each phase for each organizational unit are important data for analysis in the people problems.

3.5 HARDWARE EQUIPMENT/COMPUTER SUPPORT FACILITIES

A basic factor impacting project productivity and software development costs (and, perhaps, program reliability) is the hardware environment available to the development team. The computer hardware and support facilities are generally restricted to those dictated by the specific software firm or government contracting agency. The computer installation is a business enterprise in itself, and cannot always be available for fluctuating individual project usage requirements because of economic reasons. The computing equipment and support facilities are becoming more efficient, and do attempt to meet the needs of the individual user because of the competition in such service bureaus. However, the expense of maintaining and running the computing equipment and employing the necessary personnel for support activities requires that the computer facility be saturated with users in order to meet its resource expenditures.

While it has not specifically been stated that improving project productivity through analysis of hardware components and/or modes of computer operation is an objective of the RADC repository, there is a significant amount of work that can be done in this area. The analysis of computer usage data may include such factors as system throughput or individual programmer's frustration with poor response time. While the collection of data of this nature may be a potential and direct benefit to users of the repository, it constitutes a large bulk of data, and should only be collected to support specific studies. In any case, the structure of the data base should be flexible enough to allow collection and storage of this type of data when the need arises.

3.5.1 Computer Hardware Definition

The computer hardware components are project dependent variables greatly affecting the software product being developed. An attempt must be made to gather data on the factors that make each software project unique. When these factors are quantified, there will be a basis for comparison of the projects' characteristics. At a minimum, the parameters necessary to define the computer hardware include the type and name of equipment, the components comprising the configuration, storage capacity, and the processing speed.

3.5.1.1 Systems Throughput

The computer configuration and components are basic to the software development project. As such, many of the problems of the project personnel are predicated on the capacities of the hardware environment and how they are managed. The total productive work of a system from machine readable input of data through utilization and processing to the human readable output of data is defined as system throughput. Although it provides a measurement by which project hardware environments may be compared, the cost expended to obtain the measurement may be prohibitive.

The basic hardware configuration supplied by the manufacturer is generally accompanied by an equally basic operating system. Both hardware and software are becoming more and more modular in design, and can be expanded with increased costs as users' needs increase. An evaluation of the efficiency

of the hardware/software system in its entirity is a more difficult measurement to obtain. Generally, a compiler is used for performance evaluations at this nature. The data obtained from performance monitoring is not altogether relative to the data necessary to form a basis of hardware/operating system comparisons, even if the data were available from individual software development projects. This is due to the fact that the measurements are obtained on basic hardware/software components and do not take into consideration such things as simultaneous functions, inconsistant input loads to the system, and complex paging algorithms. Especially in a system supporting time sharing operations, measuring system efficiency and capacities with transactions reaching the system at random, the load on the system will vary from minute to minute. It would be extremely difficult to estimate the impact that this has on project productivity, if the data could be even be quantified in the first place. In order to compare system throughput across software projects, the measurement must be obtained in a like manner for each hardware/software environment. It appears possible to develop an algorithm of this type of measurement if further studies necessitate these data for comparative purposes.

3.5.1.2 Modes of Computer Operation

Many computer facilities have a variety of operational modes to offer the user. There are advantages and disadvantages to each mode of operation - mainly concerning costs and productivity. Therefore, project management must be aware of the choices available and the consequences of each choice. Much work in the past has been done in analyzing the characteristics of modes of operation. Figure 3-1 is a summary of the data found concerning modes of operation. (This data has been obtained from Aron [3], Erickson [22], Sackman [49].) Further, current attention on this problem has been addressed by Brooks [14]. He presents data supporting increased productivity by the use of interactive computer systems, including the data obtained from Harr which indicates "that an interactive facility at least doubles productivity in system programming".

BPS - Batch Processing System (Remote Job Entry)

DESCRIPTION	ADVANTAGES	DISADVANTAGES
A computer facility characterized by users submitting computer run jobs at pre-established remote locations. The user picks up the job generally at the same location after it has been run. The user has no direct contact with the computer hardware and his specific run.	1. Costs are significantly lower, especially in non-prime times. (Except when total user load is less than one-third system capacity.) 2. Greatest amount of throughput for non-interactive jobs. 3. User tends to check components of job more carefully when turnaround time is high. 4. User can submit more than one job at one time for different objectives. 5. Easy to introduce into a project as it requires no change in management or programming practice. 6. Supports program librarian's control of PPL and acquisition of job related data. 7. Supports "public" program concept of chief programmer teams.	1. Submittal and pick-up of jobs is dependent on preestablished times and places. 2. There is a queue for processing both input and output of the job. 3. No communication/ interaction with program during execution. 4. Programmer's productivity depends on turnaround time.

Figure 3-1. Modes of Computer Operation

MCB - Multi-Console Batch Processing or Remote Job Entry Terminals

DESCRIPTION	ADVANTAGES	DISADVANTAGES
A computer facility characterized by multiple consoles with individual input devices, such as card readers or terminals, and output devices, such as printers. Each console is operated by individual users running their specific job. Each job runs to completion in a queue of first in, first out. Many facilities of this nature use the traditional batch processing mode of operation for the bulk of the input, using the remote terminals for subsequent changes or execution commands. (Many of the advantages and disadvantages of the BPS apply to MCB.)	1. Programmer has some direct access with computer. It is not interactive as the computer stores input queues. 2. Reduces turn-around time as compared to BPS. 3. Can support more than one user per location.	1. Cost of terminals, card readers, printers and communication lines. 2. No communication/interaction with program during execution. 3. Costs of familiarizing personnel with equipment.

Figure 3-1. Modes of Computer Operation (cont'd)

TSS - Time Sharing System

DESCRIPTION	ADVANTAGES	DISADVANTAGES
A computer facility which supports multi-users, running individual jobs with the same computer simultaneously through a remote console. On-line terminals provide the user with the capability of communicating directly with the progress and control of the job he has input via the terminal.	1. Least amount of user time required to run a job. 2. Least amount of computer idle time. 3. Greatest throughput for interactive jobs; resulting in less personnel idle time. 4. Immediate feedback from job execution. 5. Turnaround time can be reduced to immediate response. 6. Storage and retrieval of data is immediate. 7. Monitoring of programmer's time for productivity measures is more flexible because of terminal hours data. 8. On-line debugging and analysis contributes to a decrease in manhours required for program production. 9. Programmers appear to "like" T.S. and interactive communication with the computer. On this basis, productivity and product quality may improve. 10. Problem solving with interactive T.S. capabilities is available.	1. Highest computer system cost ($/hr); or higherst overhead per user transaction in the operational systems and support tools currently available. 2. Terminal acquisition and maintenance contributes to costs. 3. Costs of training users with tools and techniques of T.S. systems.

Figure 3-1. Modes of Computer Operation (cont'd)

PC - Personal Computer

DESCRIPTION	ADVANTAGES	DISADVANTAGES
A computer facility characterized by the user having full control of the computing equipment in running his job. (Comparisons of the personal computer will not be presented because of its limited applicability to the data repository.)		

Figure 3-1. Modes of Computer Operation (cont'd)

A tentative conclusion to be reached from the data presented is that the advantage of one mode of operation over another mode depends to a large extent on the conditions and resources of the particular software development project. Further and more current analysis of mode of operation on productivity and reliability measurements may be possible with the collection of detailed log reports of computer usage. It is recommended by this study, however, that collection of data of this type be initiated only in direct support of specific studies concerning the problems and alternatives of computer utilization and man-hour allocation.

3.5.2 Computer Support Facility

There remains another set of factors associated with the hardware environment that impact project productivity and performance. These factors are all related to the accessability of the people to the problem center, and the services offered by the computer support to meet the requirements of the individual project.

The optimum situation for project personnel is to have the computer center in close proximity to the people using the facility. The need for this is not as great when a time sharing system is being used; however, even with time sharing use there are sometimes high volume printouts which are only initiated in an interactive mode and subsequently printed off-line. Courier service, or other personal delivery service, serverely limits the turnaround time, which is perhaps the single most important factor affecting productivity during the unit and integration testing periods. Even when an efficient courier service can be obtained, the cost of the service is not insignificant. The wear-and-tear on project personnel required to drive some amount of distance to the computer center (regardless of whether they are reimbursed for the mileage or not) is hardly a better solution. A short time of inconvenience does not appear to seriously impact the overall performance of the project; a long term of inconvenience and delay produces ill feelings and resentment amongst project personnel and impedes productivity.

Project management does appear to realize the problems associated with a computer center remote from the project location, and appropriate steps are generally taken in making the service problem impact productivity as little as possible. Distribution of the work package in such a way as to allow (or require) programmers to be working on more than one work unit at a time has the effect of utilizing time during service delays. Also, emphasizing to project personnel the need to fully analyze test results between computer runs reduces the total number of computer runs necessary to debug the program.

The turnaround time delays caused by the computer's inaccessability, the additional costs to meet travel requirements of courier service or project personnel from office to computer site, and the length of time of inconvenience of computer inaccessibility are factors to weigh in certain analyses.

3.5.3 System Installation

It is often the case that the software system developed at one computer location must be installed and maintained at another location. The final acceptance of the product by the customer may be at the on-site location, or it may be transferred to the customer after the system has met a predetermined set of acceptance requirements, regardless of the location. The terms of software acceptance are generally specified in the contract agreement. Of interest to the analysis of software reliability and development costs are the problems that result in the transfer of a software system from one location to another, albeit the same hardware configuration. The resources required to install a computer system at a site are frequently underestimated. Metzger [35] sites an example where one project manager dismissed the costs of system site installation as being of little consequence. The result of this action was that "the final bill for site activities was about a quarter of a million dollars, which was almost equal to the original estimate for the entire project, excluding equipment costs."

There are generally two methods used for installing a new computer system. The first method is to install the new system in parallel with the old system, and compare the results. The second method is to immediately replace the old system with the new. In either case there are no standardized techniques for acceptance testing of the new software. When adequate acceptance procedures and criteria are formulated, both the software developers and acceptance personnel will have more confidence in the site operation of the product delivered. The acceptance criteria for software systems depend heavily on the individual problem to be solved in a specific environment. Modifications to software during on-site installation often involve a design change, which is a costly item at this stage of the software life cycle. Redesigning, recording, and retesting at a remote site may necessarily indicate additional man-power, increased traveling costs, etc.

Identification of the personnel required to install the software system is generally made well in advance of software delivery and installation. The personnel must be familiar with the software system, with the technical problems associated with installations of this nature, and with the customer. Often, project personnel will have the responsibility for training the customer in the system use. Resource allocation for travel and lodging facilities must be made with adequate resources available for extended stays. The problems of multi-site installation only magnify the original problems by the number of installation sites.

Data collected on installation costs, methods, personnel, number of errors, and amount of code modification incurred during the remote site software installation will support analysis of the conversion problems.

3.6 SOFTWARE ATTRIBUTES

At the center of the software development process is the nature of the problem to be solved and the mechanisms chosen to solve it. The process of solving the problem involves its analysis and complete understanding; a translation of the analysis to operational software, plus the associated activities of program documentation, design and implementation of the data base, and

storage and maintenance of the software system components. Many software attributes have been examined; from this study, the following have merged as being the most significant attributes examined:

- Type of Programming Application
- Program Complexity
- Software System Size
- Programming Methodology
- Programming Language
- Documentation Requirements
- Data Base Requirements
- Program Production Library

3.6.1 Type of Programming Application

It is generally accepted that a specific programming problem is one of the following four classifications:

a. Business applications programming: computer programs designed to solve predefined business oriented problems.

b. Scientific applications programming: computer programs design to solve problems in the field of research, engineering or science.

c. System programming: computer programs designed to solve the interface and control problems between the computer hardware and the application programs.

d. Maintenance programming: computer programming designed to effect changes in existing programs in any of the above types.

However, the classification of application does not necessarily indicate the scope of the problem to be solved. There is such a vast range of problems within any one of these classifications that the class alone does not characterize the programming problem. A more detailed description of the

software characteristics appears to be necessary. Hence, in addition to the general application area, the following statistics should be automatically collected:

a. Number of input/output formats

b. Percent logical instructions

c. Percent mathematical instructions

d. Percent input/output instructions

e. Percent control instructions

3.6.2 Program Complexity

There is currently no effective, non-subjective complexity measure available to the software industry by which one can determine the difficulty of a given program. Although much research work has been performed in this area, especially in the field of program structural analysis and data structure and relationships, a reliable method for quantifying complexity has yet to be developed. Factors such as the number of logical paths, realtime operational characteristics, the number of input/output transfers, number of decisions, etc., are all components of the complexity measure. However, none of these measurable program characteristics by themselves provide a consistant and acceptable, overall complexity indicator. Estimating the complexity of a computer program is a basic component to the entire resource expenditures of the software project, the overall project productivity, and the reliability of the software.

3.6.2.1 Program Complexity and Resource Estimation

Much analysis has been done on the implication of complexity to resource expenditures. Detailed study was presented by Weinwurm [65] using the Shaw Index[1]. Figure 3-2 and 3-3 indicate computer usage rates in different types of program applications. One may not be able to conclude that the different applications are more or less complex than one another from the data presented, but noting the variance in resources expended on computer usage and the productivity rates for given types of applications, allows for a basis of complexity estimation as it pertains to particular applications. One conclusion drawn by Weinwurm after analysis of the data is that increasing the amount of data collected to include organizations of all types and sizes in order to represent every kind of programming task will provide a basis for future economic analysis of computer programming. The benefits derived from a joint effort of such a data collection and analysis scheme proposed by Weinwurm would provide "an excellent change of eventually establishing economic control over the programming function."

Aron [2] suggest a complexity estimation technique to be used in the process of estimating resources for large computer systems. Complexity ratings are characterized by the following definitions:

a. Easy - Programs that have very few interactions with other system elements. For example, this includes modules that solve a mathematical or logical problem. "Easy programs generally interact only with input/output programs, data management programs, and monitor programs."

b. Medium - Programs that have some interactions with other system elements. For example, this would include utility programs, language compilers, schedulers, input/output routines, data management systems. Generally they are

[1] Shaw Index (Kilo babbages per hour) $= \left[\frac{3.6 \times 10^9 \times \text{Bits per Memory Word}}{\text{High-Speed Memory Cycle Time}}\right] \log_2 \left[\begin{array}{l}\text{Bits per} \\ \text{Memory Word} \\ \text{x Words in} \\ \text{High Speed} \\ \text{Memory}\end{array}\right]$

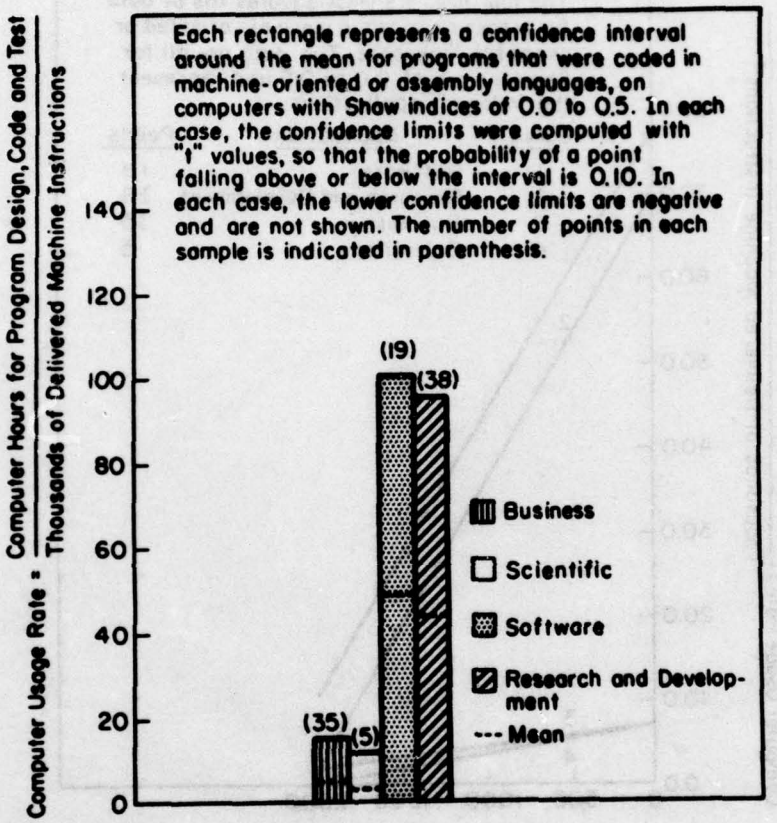


Figure 3-2. Computer Usage Rates for Different Applications [65].

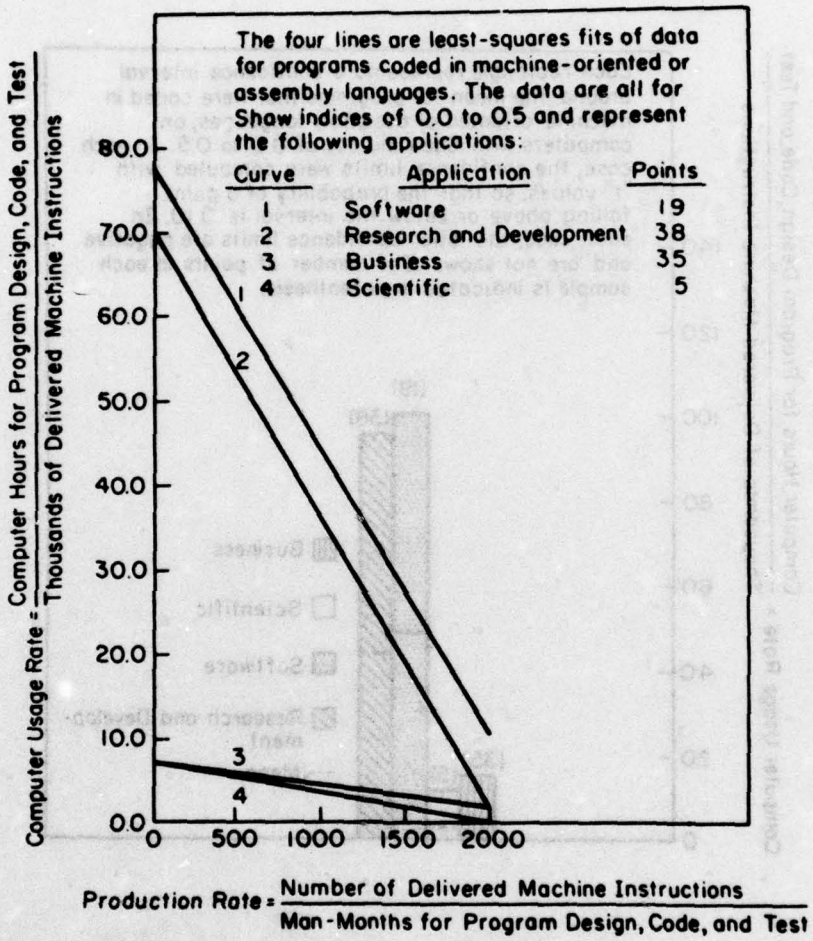


Figure 3-3. Computer Usage and Production Rates for Different Applications [65].

programs that interact with the hardware functions, monitors and/or application programs, and/or may be generalized to handle multiple or variable inputs or outputs.

c. Hard - Programs that have many interactions with other system elements. This category includes all monitors and operating systems, as well as some special-purpose programs.

The productivity rate of a given software development team is directly related to the complexity of the software under development. Data presented in Figures 3-4 by Aron [2] summarizes an informal IBM estimate on historical data for the time period 1960-1969 from IBM systems programs (1410, 7040, 7030, System/360 OS/PCP) and application systems (Mercury, SABRE, banking and brokerage, FAA). Aron indicates that the productivity rates appear to be the same for 1969 as 1960, without an apparent reason. Aron gives a possible explanation for the apparent lack of any increase in productivity in the industry at the time the data is presented: "Programs are getting more difficult at the same rate that programmer skills improve. This implies that as computers become more powerful, we use them to tackle harder problems."

Since the time Aron presented the IBM data, Baker [4] has presented more recent data showing increases in productivity rates using Aron's measures of complexity, as shown in Figure 3-5. (Baker attributes the increase in productivity and the high quality of the software of the information bank system to the chief programmer team approach of functional organization in the project, and not to the complexity of the software. It is presented here as a contrast to Aron's explanation).

	Duration / Difficulty	6-12 Months	12-24 Months	More Than 24 Months	
Row 1	Easy	20	500	10,000	Very Few Interactions
Row 2	Medium	10	250	5,000	Some Interactions
Row 3	Hard	5	125	1,500	Many Interactions
	Units	Instructions per Man-Day	Instructions per Man-Month	Instructions per Man-Year	

Figure 3-4. Productivity Table [2].

Work type	Staff time (man months)											
	Programmer											
	Chief	Backup	Analyst	1	2	3	4	5	Technician	Manager	Sec'y	Total
Requirements Analysis	2.5	1.0	8.0	0.5	–	–	–	–	–	–	–	12.0
System design	4.0	4.0	4.5	1.0	–	–	–	–	–	–	–	13.5
Unit design, programming, debugging, and testing	12.0	14.0	10.0	13.0	4.5	2.8	3.7	4.5	–	–	–	64.5
Documentation	2.0	2.0	4.5	1.5	0.2	0.2	0.3	0.3	–	–	–	11.0
Secretarial	–	–	–	–	–	–	–	–	–	–	7.0	7.0
Librarian	–	–	–	–	–	–	–	–	5.5	–	2.0	7.5
Manager	3.5	2.0	–	–	–	–	–	–	–	11.0	–	16.5
Total	24.0	23.0	27.0	16.0	4.7	3.0	4.0	4.8	5.5	11.0	9.0	132.0

Analysis of project staffing by time and type of work

Difficulty	Level		Total
	High	Low	
Hard	5034	–	5034
Standard	44247	4513	48760
Easy	27897	1633	29530
Total	77178	6146	83324

Lines of source coding by difficulty and level

Organization	Source lines per programmer day
Unit design, programming debugging, and testing	65
All professional	47
With librarian support	43
Entire team	35

PRODUCTIVITY RATES

Figure 3-5. Productivity Data using Chief Programmer Team Concept [4].

3.6.2.2 Program Complexity and Program Reliability

Cheng and Sullivan [16] stated that the quality of the software is a function of a set of independent variables contributing to the construction of the software. According to these authors, one of the independent variables is the algorithm, or problem, to be implemented. The resources and the manipulation of those resources needed to mechanize an algorithm is one component of complexity. However, some programmers have the facility for transforming even the simplest algorithm into a complex, multi-path routine. The converse is also true. "Good" programmers can transform a complex algorithm into a straight forward, easily understood program. One must conclude with Cheng and Sullivan that the algorithm itself is only a part of the total complexity problem.

Complexity in a program makes that program less understandable, not only to code-readers or testers, but also to the programmer who originally coded it if he has been away from it for a time. This contributes to making the program less reliable in that testing is incomplete due to the difficulty of devising adequate test cases. Also, modifications to the program because of errors or changes are less readily made. One reason for the current attention being focused on modularity is that when programs adhere to a set of rules restricting data relationships and module interfaces, the complexity level of the programs is decreased. This, in turn, makes program development, testing, and maintenance easier to accomplish, while increasing the overall productivity and reliability.

Collection of complexity estimations before and after the process of converting an algorithm into a program is important, as is estimation of complexity of the same program by different people, e.g., the originator of the code and the personnel testing the code. Because complexity must be an estimation until there is an automatic and acceptable method for determining it, a dual estimation process will serve to authenticate the original complexity estimation.

3.6.3 Software System Size

The size of the project staff is obviously related to number of instructions necessary to solve the programming problem, i.e., system size. System size is not the whole determiner, however, since the number of staff members required will also be related to the difficulty and complexity of the problem, the amount of interaction and coordination necessary with the customer, as well as organizational and managerial considerations previously mentioned. The number of instructions, whether MOL or POL, that must be written to solve the problem is probably the bottom line in estimating resource requirements.

The estimation of the number of lines of code needed to deliver the software system usually begins in the proposal stage. What is necessary to manage the resources of the project effectivity is a method for updating the size of the system on an incremental basis throughout the development process. Aron [2] states that when the software project reaches the design phase, reestimates of system size can be made based on the level of detail in the design process. "The key to estimating system size is found in the design. Since the system is an aggregate of elements, its size can be determined by counting the number of elements and multiplying the result by the average element size. The number of units is estimated from the design by carrying at least one package design down to the unit level."

Initially, estimates of software system size should be collected prior to work initiation. Depending on the validity of these estimates, incremental updates may be obtained as guidelines for estimation became established. With this data, analyses can be made on the estimation process which may result in considerable improvement in the estimated/actual variances.

3.6.4 Software Development Methodology

Examination of some of the programming concepts, techniques and tools currently being employed by the industry was presented in Section 2, Survey of the Literature.

There are probably many additional methodologies currently used in the analysis, design and programming process. It is necessary to identify as near as possible the method, and characteristics of the method, employed by each particular project in order to determine the effect that methodology has on the project productivity and software reliability. Identification of the software development methodology does not necessarily indicate that all characteristics of the methodology were consistently employed by project personnel. Therefore, the specific characteristics that project personnel generally employed, such as top-down development, must be identified.

In order to collect project specific data on the employment of software development methodologies, it is recommended that a set of data be obtained for each class of technique used. In this manner, information can be obtained on all techniques contributing to the project's methodology, but which are not necessarily identified with one specific programming approach. For example, the use of decision tables may support structured programming but are generally not considered to be a part of structured programming methodology. The classes of methodology proposed include:

a. Analysis Methodologies - Tools, techniques and/or concepts used in the analysis phase of software development, such as modeling, simulation, trade-off studies.

b. Design Methodologies - Tools, techniques and/or concepts used in the design phase of software development, such as top-down, modeling, proofs of correctness, informal design, bottom-up, hierarchical structure, modularity, decision tables.

c. Implementation Methodologies - Tools, techniques and/or concepts used in the implementation phase of software development, such as modularity, control flow restrictions, programming standards, bottom-up, proofs of correctness, program production library.

d. Management Methodologies - Tools, techniques and/or concepts used by management during software development, such as chief programmer teams, configuration management, program production library, build approach, automated schedulers.

e. Quality Assurance Methodologies - Tools, techniques and/or concepts used to assure software quality, such as static and dynamic test tools, test teams, top-down testing, bottom-up, program production libraries, formal design walk-thru, automatic test case generators.

f. Notational Mechanisms - Tools or techniques used in documenting the software components, such as HIPO, programmer's notebook, decision tables, flowcharts.

A significant problem exists in identifying methodologies used in the development process in that common terms lack common definitions. Even within a given project, there is ambiguity in definitions. If sufficient data is obtained to support studies of the impact of specific methodologies employed by projects, the elements of that methodology as applied by the projects are sure to differ. Because of the lack of standardized software terminology, a glossary of terms containing descriptions of programming methodologies may be useful in obtaining data that is actually representative of what is used by differing projects. Although this may be a large task, it appears to be necessary if collection of data on this level is to be made and be meaningful.

Additional information concerning the programming methodologies will support reliability, productivity and cost studies, including the degree of mechanism involved, resource expenditures for acquisition, training and maintenance. Also important are subjective evaluations as to the effectivity of the methodology as it relates to the specific application, and the independence of the methodology as it is supported by other project characteristics.

3.6.5 Programming Languages

Programming languages have become an important attribute to software production because of their practicality. They are suitable for a large variety of problems and are an economical means of solving those problems. Although there have been over 200 higher level languages developed in the past 20-25 years, only 10 to 20 of those languages have had wide spread use and applicability. Much work continues in the field of language development, assessment of the "right" language for the particular application [31], and extensions of programming languages to support work in requirements specification and the software design process.

The method by which a programmer communicates a specified problem to the computer is formulated in a rigorous set of preestablished rules inherent in the programming language. Early communication between man and machine were so rudimentary as to almost guarantee unreliability, as well as seriously affect productivity. As languages developed and became more sophisticated, an increase in reliability and productivity was an immediate reward. However, the use of a specific language for the problem at hand does not guarantee increased productivity or reliability. The language used is sometimes imposed on the project by the contractor, and/or may be ill suited for the application. Structured programming techniques call for a language that can represent specific control constructs, top-down program implementation, and data communication relationships. At this point, the major procedure-oriented languages do not contain all the attributes necessary to support structured programming methodology.

Another consideration in the selection and use of a programming language is the debugging aids associated with the compiler. Program modification and checkout time is affected by the reliability, the optional control and data information listings, and associated language documentation. Machine and operating system independence is another attribute impacting reliability and productivity, especially when software is being developed at multi-locations. (More than one face has turned red when a "debugged" software

system is delivered to the customer's installation and the source file of that system will not even compile.) The familiarity of the project personnel with the language selected directly affects the costs incurred during the implementation phase if one considers the training and "break-in" time required of programmers to effectively use a new language.

The preceding discussion has been primarily aimed at the considerations of the use of a procedure-oriented language in the programming project. It is not always feasible to use a POL. Although procedure-oriented languages are recognized by most people to be preferable to assembly or machine-oriented languages, the need for MOL's and associated debugging tools will probably continue for the foreseeable future. The current trend for providing users with source-level debugging facilities is a move away from the needs of the MOL users. It must be recognized that the software systems written in a MOL may not be supported by debug tools other than core dumps, which may further limit programmer productivity and software reliability.

There has been much discussion on the advantages of using a higher-level language over a machine-level language in the software industry. Brooks [14] states definitively that the "chief reasons for using a high-level language are productivity and debugging speed...there is not a lot of numerical evidence, but what there is suggests improvements by integral factors, not just incremental percentages." Discussion at the Software Workshops [27] sponsored by ESD in 1974 indicated that the participants' experience with assembly language or machine-oriented language "was about twice the cost per source instruction in a higher-order language such as COBOL or FORTRAN. The dollar figures were derived from an estimate of 15-30 HOL source instructions per man-day and the typical figure of $35,000 per burdened man-year for software manpower."

In summary, the programming language used on the software development project, whether a MOL or POL, must be evaluated not in relation to the attributes that the particular language has, but in relation to the project problem and programming methodology used.

The information to be collected on the source language used in performing the programming task may contribute to the analysis of the overall contribution that that specific language made to the software development process. A more detailed analysis of what language attributes specifically relate to errors incurred during program implementation, or how language constructs increased programmer productivity, may be possible if one were to collect error data from first compilation through maintenance programming on the specific task. Because of the volume of data generated by a collection procedure of this sort, it is not recommended that data collection on compiler error statistics and construct usage be done unless an in depth study of this nature were being supported, and there are automatic methods to gather this type of data.

3.6.6 Software Documentation

There is a definite need for quality documentation before, during and after the development of any given software development project. Documentation must provide technical guidance for many different people at all levels of the software project throughout its life cycle. It is an important and vital communication link; yet, it is beset with a myriad of problems, based for the most part on the reluctance of software systems personnel to record the functions and conditions specific to the give software project. Documentation is responsible for a large percentage of the costs incurred in software development, both as a deliverable requirement and as a vehicle for transmitting erroneous, vague and/or poorly defined information. Trauboth [60] has stated that "for certain real-time software systems, the documentation effort has been estimated to be up to 30 percent of the total development costs." For all the analysis directed at costs of development software, perhaps a basic element to those costs - documentation - has been overlooked.

The Sizing and Costing Workshop [27] indicated that documentation costs were approximately 10% of the total software costs, or $35 - $150 per non-automated page. Using the data presented by Nelson [38] in estimating resources needed for each task, the preparation of each user documentation requires:

a. Two man weeks per outline

b. Three-five pages per man-day for drafting

c. Twenty pages per man-day for technical review

d. Fifty pages per man-day for editing

e. Ten pages per man-day for revisions

f. Two pages per man-day for illustrations

g. Twenty pages per man-day for typing

With these estimations, it becomes obvious that documentation costs could easily run 10% of the total software budget.

Productivity has been, and continues to be, measured in lines of code (source or machine) per man-power unit. It seldom includes number of pages of documentation reflecting detailed study and analysis. Perhaps the emphasis in the software industry has been too much and too long on measuring productivity on the lines of deliverable code, which is only one component of the software development process.

There are documentation requirements in almost every software development project. These range from simplified user's manuals to complex descriptions of trade-off studies and algorithm specifications. There have been some advances in automating the documentation process, including storage and retrieval of documents, selective extraction of information, automatic editing of document information files, and automatic flow chart capabilities. However, these automatic aids do not provide for all the documentation needs of the software industry. An important factor in productivity measurement is the total amount of documentation required to be written for, or distributed to, the customer for the design, implementation, and the testing phase of the software under development.

An evaluation of the availability and quality of documentation needed by project personnel to be used in the development process is also an important piece of information to be evaluated in relation to the overall quality of the final software product.

3.6.6.1 Requirements Specification

The single most important document produced in the software development process, according to the attendees at the Monterey conference [58] and the ESD workshop on Sizing and Costing [27], is the requirements specification document. The design, implementation and test activities of the software development process are based on this set of agreements between customer and contractor. If the specifications are poorly defined, or if there is a redirection of effort and purpose during the course of development that is not reflected in this document, estimates of resource requirements, productivity and reliability will be invalidated. Much time will be lost in defining requirements and getting agreement on them; much code may be discarded through misinterpretation, misdirection and change; and numerous errors in logic and performance may be expected to result. Dr. Carl Davis, BMDATC, indicated at the Data Collection Workshop at SDC in December 1975 that the stability and quality of the requirements specification may be the key factor in project productivity.

The quality and stability of requirements specification should be evaluated both before and after the software system is developed. They could also be correlated to the quality of the system design and eventual ease of testing. The measurable characteristics of documents (size, mode of presentation, amounts of tabular and illustrative material, organization, clarity, and completeness of coverage) may be related to project productivity and software reliability. If these specifications are found to influence performance, data may be obtained to improve both presentation and interpretation of requirement specifications for future software development projects. Initially, a subjective evaluation of the requirements specification should be a definite data parameter to collect.

3.6.6.2 Design Specifications

Design specifications, whether expressed as narratives, flow diagrams, decision tables, or as special forms, are usually better documented than requirements specifications. Controversy exists over the degree of detail, mode of presentation, and, to a certain extent, the content of design specifications. Recently IBM has suggested HIPO (Hierarchy-Input Process, Output) charts for design documentation [28]. For large military systems several levels of documentation and many aspects of the system may be involved (see MIL-STD's 490 and 483; DOD Manual 4120.17M); so that design specifications may represent a huge volume of paper and a great deal of work. Obviously, these have high impact on project costs. Hence, design specification quality should be evaluated.

3.6.6.3 Program Documentation

In the commercial world, program documentation usually means a description of what has been produced, which becomes the basis for program maintenance and training. In military systems, where much more detailed design documents are produced and maintained, these documents become the maintenace manuals. Recent advocates of structured programming, higher level languages, and the elements of programming style have stated that top-down design, levels of abstraction, functionally simple modules, and adequate commentary have removed the need for much descriptive documentation. That is, the programs are self-documenting and all that is needed is descriptions of the levels of abstraction and the data structures of each. Further, it is alleged that the use of a Chief Programmer Team reduces the communication and documentation requirements to such a degree that formal design and descriptive documents are greatly reduced in importance. A single person, the Chief Programmer, is responsible for all designs by persons under his management (requiring at least a small team and a phenomenal memory), and needs no detailed documentation to communicate. The use of design walkthroughs may catch most incompatibilities among submodules. However, Wasserman [64] states that "the discipline involved in actually preparing documentation is often as valuable as the document itself." In short, the act of organizing the information and writing it down forces many minor design decisions and

clarifies many problems that could later cause difficulty in implementation or maintenance. Evaluating the impact of various methodologies of program documentation also appears to be an important investigation.

3.6.6.4 User Documentation

The requirement for user documentation varies greatly with the nature of the system. Certainly a realtime command and control system requires a great deal more in the way of positional handbooks and console guides than a simple, stand-alone function. Minimal operating and using instructions are usually provided with a program, but in large multi-purpose systems separate documentation is normally provided. There are many questions that can be asked about the effectiveness of various ways of presenting using procedures, but these may be beyond the scope of the repository.

3.6.7 Data Base Requirements

The construction and maintenance of the data base used by a software development project can range from a simple common declaration to a complex data management program and data base structure. It appears necessary to at least estimate the complexity of the software project data base in order to quantify this project attribute for comparative purposes. Some of the variables to be considered in estimating complexity of the data base structure include the type and number of relational representations, hierarchical structure, repeating substructures, and number of instances of each data type. The parameters to be considered for the complexity of the management system associated with the data base include volume of transactions, including updating and retrieval, response time requirements for transactions, retrieval identification logic, and number (or porportion) of elements in data base upon which retrieval can be made. It is recommended that, initially, the repository collect only data base complexity estimates.

3.6.8 Program Production Library (PPL)

One of the programming support aids currently being employed by many software development projects is the program production library. This support tool consists of a set of office and computer procedures designed to provide program and test case maintenance, enforce established programming standards, and provide information and visability for both project management and programing personnel. The use of the PPL helps to coordinate the status of program components under development, while also helping to automate configuration control procedures. This is accomplished by storing program modules in a data base and maintaining records of data on the contents of the data base. The procedure is analogous to the storage and cataloguing of books in a library.

The program production library system attempts to reduce error and increase project productivity by providing information relating to and storage of modules of code. Usually a program librarian is assigned to clerical tasks inherent in the handling of program decks and job submittals. In this manner, the program librarian serves as an interface between the project personnel and the computer operations, while collecting any necessary data specific to the requirements of the project.

There has been much discussion on the benefits derived from a program production library, although there are obviously costs incurred in the establishment and maintenance of a PPL. Small programming projects may not need a library, and/or may not be able to justify the costs associated with it. However, the PPL concept can handle many small programming projects, thereby distributing the costs.

It does appear that the PPL system facilitates the collection of project data by assigning the task to the program librarian. It may serve to remove bias, or other effects, inherent in a data collection procedure by removing it from the actual programming team. However, a PPL can be maintained by programming personnel when the costs of a librarian are prohibitive for the specific project.

Data reflecting the use of a PPL, or equivalent, is meaningful to the analysis of project productivity. Further, data should be gathered that represents what the costs of establishing and maintaining a PPL represents to a project. These costs include the program librarian and computer overhead charges. It would also be useful to obtain subjective evaluations of the PPL from project personnel as to its contribution to project effectivity.

4. PROJECT PERFORMANCE ATTRIBUTES

In order to ensure the success of a computer programming project, there must be a means by which a project manager can monitor the work being performed in the time frame allowed. Regardless of the type of management practices employed, the major objective of software management is to obtain a sufficient quantity of data by which to evaluate and control the data processing expenditures, coordinate and direct the systems plans and activities, allocated and direct the available personnel, and monitor the quality of the final product. Often, the project manager must ensure that configuration control procedures are followed. The plans and procedures established for the management of software projects range from being totally inadequate to being overwhelmingly detailed and complex.

The objective of this section is to delineate a minimum set of data necessary to adequately define and measure the work package, the products to be delivered and the quality assurance provisions pertaining to the entire development process. In some cases the data are estimated data, but these parameters are considered to be firm and measurable because they are generally mapped to the available resources of the project within a specific time frame, both of which are firm and measurable. The three areas of data parameters to be discussed are associated with the work and product data, the configuration items, and the quality control procedures.

4.1 RESOURCE ALLOCATION

The definition of the work plan is necessary inorder to establish a viable direction by which to proceed in the development of a product, and subsequently, obtain sufficient data from project personnel and cost centers to provide visibility into what is actually occurring in the software development. The manager has the responsibility of dividing the entire task into a number of sub-tasks with the proper allocation of available resources, and most likely, under the scrutiny of the procuring agency, as well as his own upper management. This problem requires the manager to accept an extremely large number of factors (including those mentioned in Section 3), mentally process these factors, and produce an optimum solution.

The division of the total development process into distinct phases has been discussed in a previous volume. The necessity of these phases is of interest here for the specific task of allocating and expending the project's resources adequately.

Aron [2] indicates that the "ideal" elapsed time of software development should be allocated such that 30% of the time is used for design, 40% used for implementation and 30% used for testing. Since there is a relationship between the resources required and the product produced, the trade-offs applied to resource allocation during project duration need careful analysis by project management. This is demonstrated by looking at the "crash" project syndrome. A crash project is one where the allocation of time to the development phases, as well as to the total elapsed time, is far from optimum. In other words, excessive resource expenditures are required to complete the project in the time period allowed with the result that individual productivity and organizational efficiency is decreased [44]. Generally, the results of the "crash" project reflect the improper allocation by management of resources to time by poor system design, excessive computer usage, poor quality software, and poor moral of project personnel.

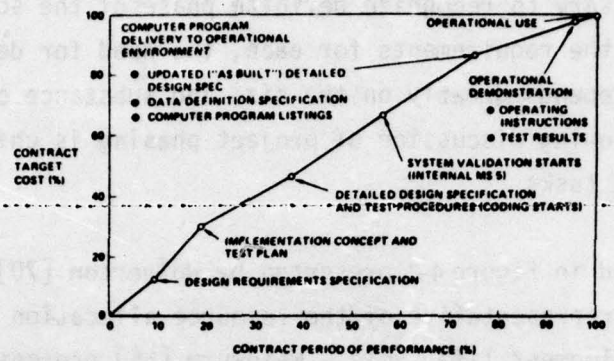
McHenry [34] states that the traditional design phase of the software development process expends 20% of the total costs, and the traditional development phase expends 40% of the total costs. After analyzing software errors generated in those two phases, he states that the design phase contributes to 40% of the errors, and the development phase contributes to 50% of the errors. (Of course, percentages are approximations.) The remaining costs expended and errors generated occur in other testing and administration activities. In practice, the software phases are not always distinct time phases, and the actions that properly terminate one phase are not always performed until the subsequent phase has been initiated. Unfortunately, reliability of the software and production costs are greatly impacted by this occurrence. (Design errors found in integration and system testing have the probability of having much greater impact on system components than if those design errors had been discovered during a design review that terminated the analysis and design phase and preceded the production phases.)

While it is necessary to recognize definite phases of the software development process and the requirements for each, the need for delineation of software phases depends greatly on the size and substance of the project itself. The following discussion of project phasing is chiefly aimed at large programming tasks.

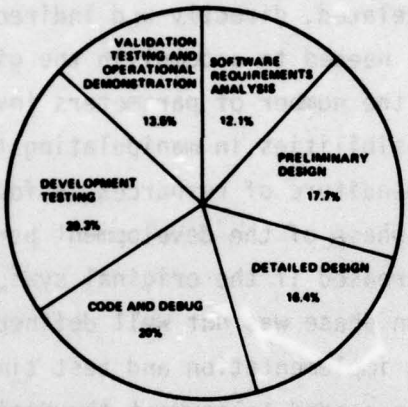
The data presented in Figure 4-1, presented by Wolverton [70] for a specific TRW project, are representative of the resource allocation necessary during the software development life cycle. Weinwurm [65] presents requirements for manpower, machine-time, and overall project cost in Figure 4-2 and 4-3. It is important to note that although these variables are directly related to time, they are also related, directly and indirectly, to each other. Estimating the resources needed to accomplish the given task is a difficult job not only because of the number of parameters involved, but also because of the wide range of possibilities in manipulating those parameters. For example, the maximum expenditure of resources is found to be during the implementation and test phase of the development period. However, even these costs can be greatly increased if the original system design was poorly constructed or the design phase was not well defined. The consequence of poor design is that more implementation and test time, in both computer and manpower requirements, is needed to correct the design errors. Weinwurm presents a sample effect of resource allocation trade-offs in Figure 4-4.

Cammack and Rodgers [15] discussed IBM's examination of maintenance costs[1] for Release 18, 19, and 20.0. Average costs for these releases for one year following FCS was 56% of the original costs. This was due, in part, to the fact that the cost of finding and correcting an error was increased as the programming cycle progressed. "The cost of finding and fixing a problem after the system was released was thirty times the cost of fixing it in unit test, the first set of tests our code goes through in development." In order to decrease costs and increase overall project productivity, the objective at IBM has been to spend more money early in the cycle so that they will have significantly fewer errors when the product is

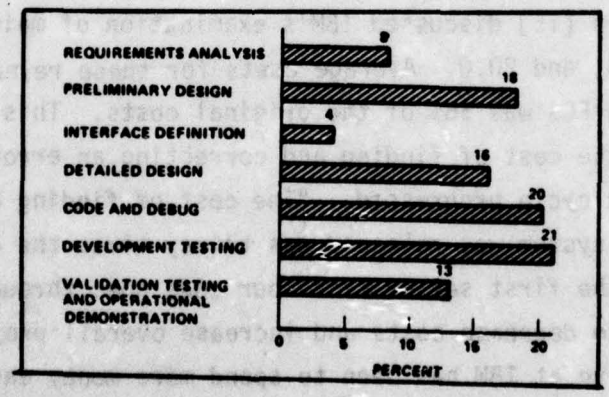
[1]Maintenance costs began with FCS-First Customer Shipment.



Typical Software Development Cost Experience

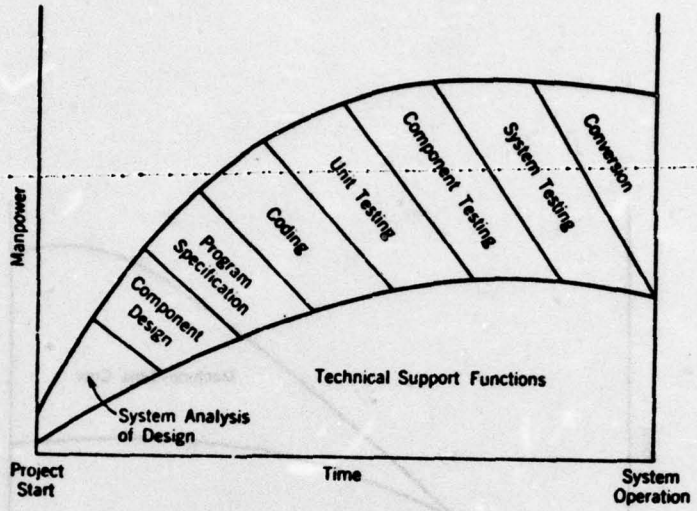


Cost Distribution by activity during full period of performance. (All activities include documentation and travel costs.)

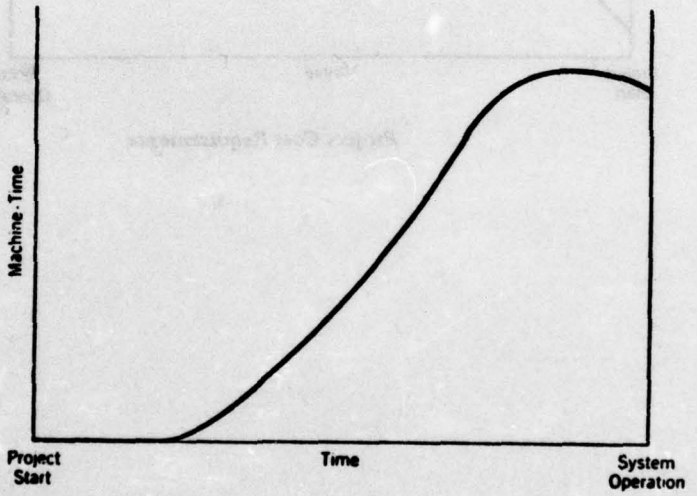


Typical allocation of resources in custom software development and test.

Figure 4-1. Specific Project Resource Allocation Experience [70].



Manpower Requirements



Machine-Time Requirements

Figure 4-2. Typical Resource Allocation [65]

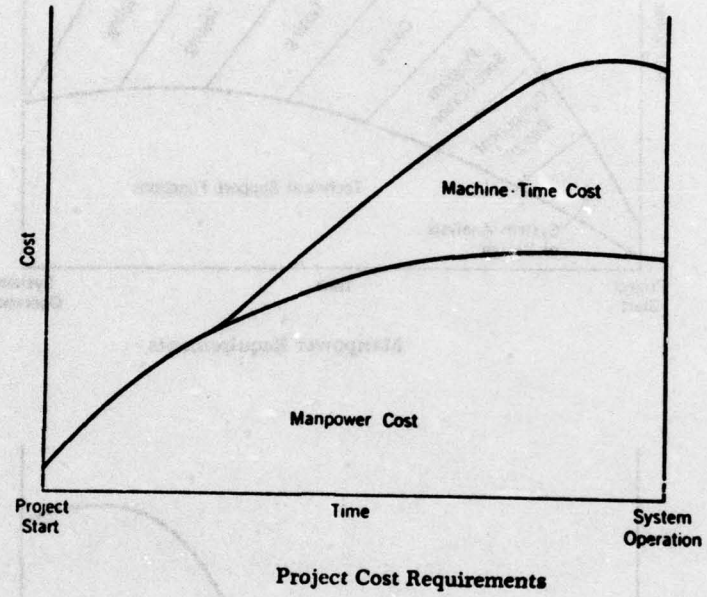
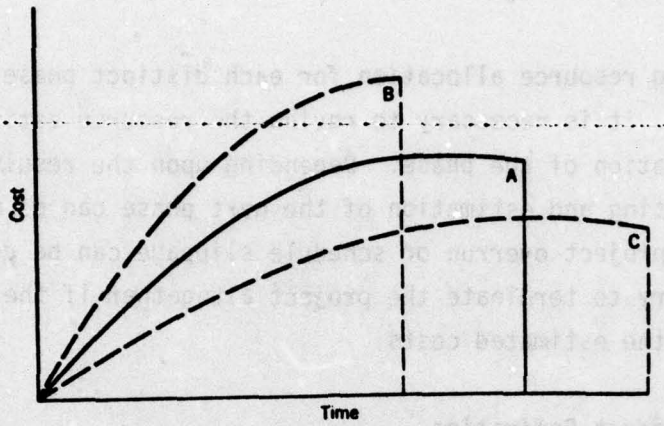
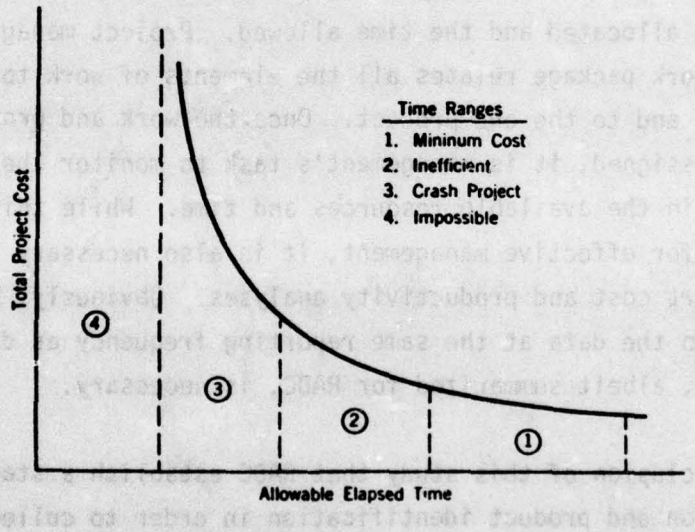


Figure 4-3. Typical Resource Allocation [65]



Time-resource Alternatives



Cost-time Trade-off Function

Figure 4-4. Resource Allocation Trade-offs [65]

shipped to the customer. The results of this change in resource expenditure are demonstrated in Figure 4-5.

Besides estimating resource allocation for each distinct phase of the development cycle, it is necessary to review the resource estimates with actuals at termination of the phase. Depending upon the result of this analysis, forecasting and estimation of the next phase can be more realistic, or problems like project overrun or schedule slippage can be determined. It may be necessary to terminate the project altogether if the actual cost data far exceeds the estimated costs.

4.1.1 Work/Product Definition

The project work plan, or combinations of plans, defines and distributes the total work package in such a way as to produce the deliverable items within the resources allocated and the time allowed. Project management's definition of the work package relates all the elements of work to be accomplished to each other and to the end product. Once the work and product have been defined and assigned, it is management's task to monitor the performance of the work within the available resources and time. While this information is necessary for effective management, it is also necessary for the repository to support cost and productivity analyses. Obviously, RADC does not need to obtain the data at the same reporting frequency as do projects, but the same data, albeit summarized for RADC, is necessary.

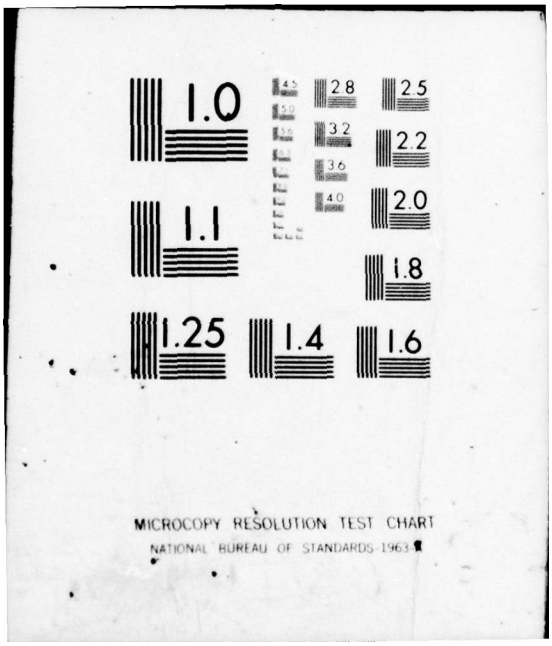
It is the conclusion of this study that RADC establish a standard method of work definition and product identification in order to collect performance data during software development. Unless there is a commonality in the collection of this type of data, measurements across projects will not be comparable, and perhaps measurements within projects may be meaningless to RADC research needs unless the project's system elements are defined.

AD-A036 064 SYSTEM DEVELOPMENT CORP SANTA MONICA CALIF F/G 9/2
SOFTWARE DATA COLLECTION STUDY. VOLUME III. DATA REQUIREMENTS F--ETC(U)
DEC 76 M C FINFER F30602-75-C-0248
UNCLASSIFIED SDC-TM-5542/003/01 RADC-TR-76-329-VOL-3 NL

2 OF 2
AD
A036064

END

DATE
FILMED
3-77


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

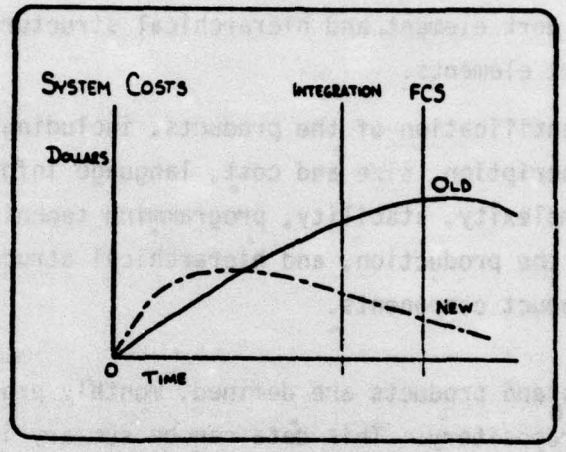


Figure 4-5. Old and New Curves for Resource Allocation in Software Life Cycle [15].

The definitions of the system elements should include:

a. Project Identification

b. Identification and description of the work elements, including stop/start dates, duration, resources allocated and expended to date, products to be produced, milestones, identification of the level of work element, and hierarchical structure of the work elements.

c. Identification of the products, including type, description, size and cost, language information, complexity, stability, programming techniques used in the production, and hierarchical structure of product components.

Once work elements and products are defined, monthly progress data can be submitted to the repository. This data can be summary in nature, but should include data from the product configuration reporting level and the work configuration reporting level as follows:

- Product Configuration Reports - including data reflecting progress on the module, subsystem, and system level for the current reporting period.
- Work Configuration Reports - including data on work performed for activity, task, phase, and total project.

4.1.2 Summary of Mangement Data

Regardless of the automatic or manual method for obtaining the management summary data, this information is derived by comparing and calculating the work plan, or estimated, data against the work actual data. It is necessary to have the planned data represented in a format so that actual progress can be measured against it. When the required project tasks have been identified with their required resources allocated, it is necessary to

monitor the actual events and activities in the time span allowed with the resource available. There are numerous management techniques for this evaluation, such as Critical Path Method (CPM), Project Evaluation and Review Technique (PERT), and Line of Balance (LOB).

The major objective of the management techniques, regardless of the type used, is to provide a picture of the actual time and costs of the individual activities and total project against the projected estimates. The management summary should provide at a minimum the following data:

Time-schedule slippages and cost overruns in actual time , and cost variance by a comparison of estimated costs with actual cost on a module, subsystem and system level, current to the reporting period.

In addition to the cost data, the management summary should provide an indication of productivity rates of individual programmers through the activity and/or milestone status reports, and, perhaps, computer utilization reports. The productivity rate is certainly more difficulty to assess, and depends heavily upon the parameters previously discussed, such as individual skill, program complexity, working environment, etc. The examination of the progress data and computer usage data can indicate to the manager areas of inefficiency or programming difficulties. If the management data reflects schedule slippages and overruns in resource expenditures, the problems inherent in the project may not be due to programmer productivity at all. They may be a result of unrealistic estimations in initial staffing and scheduling. However, it is important for productivity data to be included as an integral part of the actual work data for evaluation of this nature.

4.2 CONFIGURATION MANAGEMENT DATA

The purpose of configuration management is to define the software products to be produced, control changes to the products, maintain accounts on the current status of the products during development, and certify the quality and completeness of the products on and after delivery. Configuration management procedures became established as an adjunct to military contracts in order to ensure the technological content and integrity of the software as specified by the contract. While configuration control procedures are especially necessary for military contracts, configuration management procedures are a valid method of monitoring the status of the software configuration for any software development project.

The amount of work and the formality of the procedures involved in configuration management depend heavily upon project characteristics. Procedures should vary directly with the size, complexity and degree of innovation of the software package; the amount of coordination among project personnel; the number of clients and subcontractors; and the complexity and number of external interfaces. For small projects, the project manager's office may have total responsibility for configuration management; for large software systems, an entire organizational structure may be required to perform this function. After delivery, unless an office or agency is specifically charged with configuration management duties, modification and error corrections quickly obsolete the delivery system and knowledge about it.

Configuration management is normally divided into procedures for configuration identification, configuration control, and configuration accounting and reporting. The procedures mesh with project performance through the work breakdown structure, allied tasking procedures, and with quality assurance through the auditing, reviewing and testing schedules that establish configuration identification and processing status at any of the development and operational life-cycle phases.

4.2.1 Configuration Identification

The intent of configuration identification is to define the technological content and structure of the system as it is being developed. This is done both through the issuance and mutual agreement upon sets of formal specifications and through the recording of formal lists of configuration elements and relationships.

Each phase in the development process has its formal specification, its inventory of configurational elements, and its formal authentication procedures to "baseline" the configuration at the end of each phase. These are shown in Figure 4-6.

In the Operation System Requirements Phase, the system concept of operation is developed and major functions allocated to the software subsystem. If the requirement are not entirely firm, importance and priority ratings may be assigned. When a contract is let, a Work Breakdown Structure showing the major functional subsystems (to a four level hierarchy) and the major developmental tasks will depict the system. If implementation priorities are involved, incremental builds of the system may be shown. Requirements, priorities, WBS and builds are subject to negotiation between client and developers. Configuration data for this phase include: Function Lists, Priority Ratings, Operational Flows, Data Characteristics.

In the Software Requirements Analysis Phase the functions allocated to the software system are decomposed in detail. The structural requirements for the system (language, modularity, reliability, security, etc.,) are established, as are the decompositions to achieve generalizations and simplications, remove redundancies, and evaluate the information flow and temporal dependencies of the functions. Software acceptance criteria and testing requirements are determined and external interfaces defined. If protocols, formats, and traffic across any interface have not yet been determined, Interface Working Groups resolve these. Configuration data for this phase include: Functional Descriptions, Functional Flows, Interface Characteristics, Data Definitions, Acceptance Criteria.

INVENTORY/DISPLAY	SPECIFICATION	BASELINE REVIEW
	OPERATIONAL SYSTEM REQUIREMENTS PHASE	
Required Functions List (Work Breakdown Structure)	Operational System Description	System Requirements Review (SRR)
	SOFTWARE REQUIREMENTS ANALYSIS PHASE	
Functional Allocations List (Logical System Design Block Diagram)	Computer Program Performance Specification	System Design Review (SDR)
	PRELIMINARY (SYSTEM) DESIGN PHASE	
Configuration Index/Status (System Level Block Diagram) (Module/Function Matrix)	System (Preliminary) Design Specification (Plus SCN/ECP's)	Preliminary Design Review (PDR)
	DETAILED (COMPONENT) DESIGN PHASE	
Configuration Index/Status (Computer Program Components) (Detailed Block Diagram) (Flow Charts)	Detailed Design Specification (Plus SCN/ECP's)	Critical Design Review (CDR)
	PROGRAM IMPLEMENTATION PHASE	
Configuration Index/Status Program Master File Directory (Flow Diagrams) (System Cross-Reference Table)	Program Package: Source Code, Listings, Maintained Specs, Using Instructions, Flows	(Acceptance Inspection) Physical Configuration Audit (PCA) Functional Configuration Audit (FCA)
	SOFTWARE SYSTEM TESTING PHASE	
Inventory of Delivered Items (Flow and Block Diagrams) (System Cross-Reference) (Data Description Lists)	Version Description Document	Formal Qualification Review (PQR)
	OPERATIONS AND MAINTENANCE PHASE	
Configuration Index/Status (Flow and Block Diagrams) (System Cross-Reference) (Data Description List)	Version Description Document + Change Packages + ECP's + Discrepancy Corrections	Benchmark Testing

Figure 4-6. Software Development Phase Representation

In the System or Preliminary Design Phase, the major modules of the software are identified and functions allocated to them. This includes sorting functions into successive versions of the system if a build approach is taken. The structure and operating dependencies of each version are established. A cross-reference matrix should show in which modules every identified function will be performed. Test plans should show how the proper performance of the function will be verified. Configuration data for this phase include: Major Modules, Module Dependencies, Function/Module Cross-Reference, Module Classifications, Module Sizes, Difficulty Levels, Complexity Level, Data Modules, Classification, and Sizes.

In the Detailed or Componenet Design Phase, the internal structure and operation of the major modules are defined, design optimizations performed, and data formats and interface formats defined in detail. Procedures for accomplishing each of the planned tests are also defined. In this stage, decisions concerning implementation strategy may be made. Implementation priorities may be assigned to modules and partial builds defined. Blocks of work (modules) may be independently scheduled for design, code and test with appropriate reviews and audits governing quality. Configuration data for this phase remain the same as for the Preliminary Design Phase, but with further differentiation of structural elements and interactions. Implementation priorities may also be assigned.

In the Program Implementation Phase, modules are coded, compiled, debugged and verified. The system inventory consists of those mdoules actually included in the programming support library (if one is used). The initial representation may only be program stubs, and modules may be moved from a "development" to a "tested"file in the library.

Configuration data for this phase include all the system elements defined to the programming support library: source and object program modules, data modules, test modules, load modules, builds, versions, and mods. Actual sizes may be added to estimated sizes. Actual module cross-reference matrices may be compared to those designed.

In the Software System Test Phase, the delivered software system is submitted to inspections, audit procedures, and tests to verify that all required items have been delivered, all functions are encompassed, programming (and other) standards have been observed, and all functions operate at the required performance levels. Configuration data of this phase include the status updates as the system is certified.

In the Operations and Maintenance Phase, the system is installed in its operational environment (sometimes considered a separate phase). It is maintained to remove discovered discrepancies in programs, using and operating instructions, changes in specifications, and improvements and modifications. Configuration data of this phase include the tracking of system versions and modifications, change packages, the addition of new functions and modules, and/or the revisions of existing program and data modules.

In most military software systems configuration identification is exercised through the specification tree. Documents are produced for each phase of the development and for each configuration item and computer program component defined, but more detailed statistical information is buried with the specification and must be extracted if it is to be treated as manipulatable data. The Configuration Index lists each baselined document produced (for requirements, performance, modules, interfaces, operating instructions, tests, data base, etc.,) and the state and date of authentication. Alternatively, structural descriptions may be made of the various representations and their interrelationships, often in much finer detail than if controlled through specifications. It would appear for research purposes that the finer information is highly desirable, revealing much about the implementation strategy and the exactitute of record keeping and control.

4.2.2 Configuration Control

The second function of configuration management is to protect the integrity of the approved configurations against unauthorized change and to ensure that all authorized changes are in fact incorporated into the system. Changes to the system may arise as a result of a change in requirements, a problem arising in design or implementation, or an error or discrepancy

discovered in a design, program, data module, or using instruction following the baselining of a specific system definition (identification). Normally, a configuration control board (CCB) is created to receive, coordinate, obtain evaluation, and determine the disposition of all significant changes to the system. (Non-cost, non-impact changes are permitted without approval, but must be announced to the configuration control system.)

Requirements changes are submitted to the CCB as Engineering Change Proposals (ECP). The CCB may approve, reject, or direct further study of the proposed change. If study is directed, an analysis team is formed of one or more persons, and the technical and project (schedule and cost) impacts of the proposal evaluated. The CCB again decides the disposition of the ECP, and contracts personnel may be brought in to negotiate changes in funding and scheduling. When implemented, change pages to specifications are issued under the cover of a Specification Change Notice (SCN). Changes to modules may be issued as new mods of the module, or as a change packet to the system under cover of a Change Notice (CN) or Modification Transmittal Memorandum (MTM).

The developer may decide to write up a problem arising in design or implementation that can be implemented without impact costs, or resulting in degradation of performance (i.e., Class II Changes), on a Change Report (CR). It is then transmitted to specifications on a SCN or to modules on a CN. Problems that are serious (i.e., that could result in a Class I change - impacts on costs or functions, or require renegotiation of contract terms) may be submitted as ECP's or as Problem Reports (PR). If technical, the CCB again disposes and may request an ECP to be filed and processed. Or, the CCB may request the problem to be studied prior to making a decision to consider a change. ECP processing is completed as above. If the PR involves revision and renegotiation of the project, contract personnel are also brought in. (NOTE: PR's are much more frequently used on internal projects than on client-oriented contracts.) Change Notices are issued as before.

Error or Discrepancy Reports (DRF) are also submitted to the CCB, which forwards them to the responsible agency for response. (Errors are frequencly passed through a screening committee for preliminary review before being officially submitted.) After verifying that the alleged discrepancy exists and after determining the actions necessary fix it, the CCB disposes of it by rejecting, deferring, or directing its correction. Minor corrections are frequently deferred to be cleared up on the next version of the system rather than incorporated in the current version as new mods or change packages. Transmittal of changes is via SCN's and CN's as above. A Version Description Document (VDD) also lists ECP's and DRF's cleared by the review.

When Configuration Status Reports are issued, the exact content of the system representations is posted by listing all the SCN's or CN's issued against a specification or module, and the ECPs, CRs, and DRFs that are disposed of by the notice. Change Status Reports, usually listed by type, are issued to report the current state of all changes, including those rejected.

Configuration data for changes includes the change identify and type, the configuration items impacted, including all affected specifications, the classification (within change type); the project cost (estimated and actual); and the current status.

4.2.3 Configuration Accounting

Configuration Accounting is the record keeping and report generation functions of configuration management. Records are kept of all the various system representations that exist; either as specifications and documents, or as system elements; all versions and mods of the system elements; and of all the changes to the representations. Status records of configuration definition are kept vis a vis milestone events, i.e., the reviews or tests that baseline the definition. Status records are kept on the processing state of all proposed modifications (as rejected, pending, deferred, approved, or implemented).

In addition, inventory and status records may be kept on other project products, such as document libraries, tape libraries, subroutine libraries, master program files, disc files, data files, equipment, etc., Records are kept on the distribution lists for sensitive documents, as a security procedure, and of specification documents to be sure that changes are properly distributed. Records may be kept on the system version, or versions, that are active at particular locations to ensure that changes are properly distributed.

At a minimum, the RADC data collection system must be compatible with the established military configuration control procedures. It can be seen that it is possible to obtain meaningful system development data from current reporting forms. However, since configuration management procedures were not designed to establish an historical data base to support research needs, this study recommends that the RADC data collection effort be compatible with configuration management initially, while being cognizant to more direct support at a later time.

4.3 QUALITY CONTROL PROCEDURES

Generally speaking, there is little commonality in either the quality control procedures or methods used by project managers in the development of software. Most large companies that produce software have established and/or published a set of standards and procedures to be followed during the life cycle of software development. The objective of these documents is to establish guidelines and steps during the entire process which contribute to the overall quality of the product. Formal reviews and audits of specified software products establish a set of "baselines," which are not only important for configuration management, but are essential for determining performance of the software to the contract as specified. These periodic reviews are an important step in assuring quality of the end product. Failure to establish either quality project standards or conduct the reviews contribute to poorly defined, or absent, programming standards and conventions; poorly defined, or absent, software acceptance criteria; poor, or inadequate, system design; and/or insufficient, or poor quality system documentation.

4.3.1 Quality Assurance Reviews

On large software development projects and/or military contracts there is a necessity to terminate each phase with a benchmark of the products and a formal quality assurance review of those products. (Although software projects are initiated at the marketing and proposal stage, the reviews discussed will begin with the Requirements Analysis Phase and Performance Requirements Review). The organization of phases and reviews vary with completions of specific tasks throughout the industry, but the purpose of the phase reviews is essentially the same. That is, periodic and consistent reviews allow an examination of the work performed to date in order to checkpoint its quality and proceed with the subsequent work from that checkpoint. It allows the control authority, management and programmer to evaluate the work in relation to the entire system, and it has the effect of reducing software costs by discovering discrepancies earlier in the development cycle than they otherwise would be. Figure 4-7 depicts the software system life cycle phases with the associated reviews according to System Development Corporation.

Reviewing the products developed in order to improve the quality of the project and examine associated costs are the principal objectives of any review. Cammack and Rodgers [15] have analyzed the effect of the "walk-through" on development costs. (Walkthrough is IBM's term for a "structured review attended by the persons most closely affected by the specification." This corresponds to SDC's iterative DDR's in Figure 4-7). They found that the cost of finding errors in a walkthrough were 14 to 15 times less than to find a problem through unit test; unit test being the cheapest point in the test cycle to find errors.

Analysis of the error rate and cost effect by IBM was possible by the collection of strategic data points throughout the life cycle of software development projects. Obviously, the benefits realized by IBM in reducing overall costs were accomplished after spending resources on data collection and analysis.

PHASE	PRODUCTS	REVIEW
Requirements Analysis	1) Performance Requirements Specification 2) Acceptance Criteria 3) Project Studies 4) Estimate/Actuals Cost Data	Performance Requirement Review (PRR)
System (or Preliminary) Design	1) System Design Specification 2) Test Plan 3) Project Studies 4) Development Plan 5) Estimates/Actuals Cost Data	System Design Review (SDR)
Program (or Detailed) Design and Implementation	1) Detailed Design Specifications 2) Program Documentation User Interfaces, System Interfaces, Cause-effect Graphs 3) Test Procedures 4) Integration Test Results 5) Project Notebook 6) Estimates/Actuals Cost Data	Iterative Detailed Design Reviews (DDR's)
Test (System and/or acceptance)	1) Certified Programs 2) User Documentation 3) System/Acceptance Test Results	Formal Qualification Review

FIGURE 4-7. Software System Life Cycle Phases at SDC

In summary, there is a definite need for a periodic set of reviews of the work being performed. Whether these reviews are formal audits, resulting in the customer "signing off" on the work performed to date in order to establish a baseline for configuration control, the objective of the reviews are to establish a mechanism by which to judge the quality of the end product. It is recommended that the repository collect estimated data on the type, number, purpose, and timing of the project reviews. Subsequent to the review meeting, the review results should be examined, as well as actual updates to the original estimated data as demonstrated by the review.

4.3.2 Program Error Reporting Procedures

Most software development projects initiate a procedure for modification/discrepancy/error reporting during some phase of the life cycle of the project. The time at which the procedure is initiated depends upon the type of contract and customer for which the software is being developed, as well as individual management procedures. Generally, errors are reported following the unit or module testing level, performed by the originator of the module. Rubey [47] states that approximately 2% of the errors are discovered during the validation phase of the development process; therefore, 98% of the errors are discovered and corrected during the module through integration testing periods, or equivalent test phases. While it would be most infeasible to monitor the error data during module checkout by manual methods, an error reporting mechanism is most valuable for monitoring product quality during integration and system test phases. Accumulation of error data of this nature is extremely important for reliability analysis.

The problem report/correction procedures should be established early in the life cycle, such as the design phase. Education in the use of the forms is also essential if the procedures are to be effective. A recent error procedure was proposed for a project currently being performed at SDC. The procedure was designed to collect productivity data as well as reliability statistics. Although the procedure was examined and found to be comprehensive enough in detail, the time in which the data collection procedure was proposed was subsequent to the design phase, but prior to integration

testing. Consequently, the proposed data collection effort was rejected because of the timing of the proposal, besides the additional costs to be incurred when making a change of this nature.

The analysis of error data that the individual project manager makes can provide a clear indication of the overall and continual quality of the emerging product. However, it appears that the data obtained from many of the error reporting procedures initiated on software development projects are not effectively used. Either the goals of error reporting are vague, or the knowledge obtained from the evaluation of the data is not disseminated to project personnel. Many papers addressing the problem of error collection and quantitization state that greater understanding of software errors will lead to the improvement in the design and application of software development tools and techniques [47], but the reality of the situation does not support this conclusion. Project managers who have the ability to both initiate error reporting procedures, and analyze the incoming data, do not consistantly take action resulting from the analysis of the error reports. Project cost is one area that is greatly impacted by software errors. Data obtained from an error reporting procedure can indicate where costs are being expended if the manager has records detailing the effort and time expended on program bugs. Error analysis can indicate the necessity to apply additional personnel to a particularly error prone program or subsystem. A cluster of errors in a related group of programs may indicate poor design of that particular software. It may also be possible to determine the competency of the originator of programs by examing the frequency and severity of errors at the completion of the software projects. Little post-mortem analysis is made by way of the established error reporting procedures, either because of lack of time and funding, or because the error reporting system contributes too little to the total success or failure of the entire project.

One important contribution made by an established error reporting procedure is that it helps control the number of changes and modifications arbitrarily made to the software. Programmers are famous for altering their code for

any number of reasons. When a modification report must be filed for every software change, with specificity to the reasons for modification, changes are likely to be made only when the justification warrants the expense of the modification and retest.

The volume of data obtained from an error reporting system for even a single, moderately sized programming project is extremely high. Collecting error data from numerous projects for analysis and study must support the established and specific objective(s) of the study; obviously, those objectives must be determined first. It appears to be almost impossible to collect error data sufficient in detail and quantity to support a variety of non-specific reliability analyses. There have been many reliability studies, such as examining error type and density, reliability modeling and projection, etc. Besides these studies, configuration management procedures have made it a requirement to document the type of change or modification made to baselined configuration items. Perhaps one of the best and most recent error studies was performed by TRW for RADC. This study [61] categorized error types for several software development projects. Besides the categorization of error types, several valid suggestions were made to improve overall software reliability. However, there appears to be a general reluctance in the software industry to examine the results of exhaustive reliability studies, such as The Reliability Study done by TRW, and benefit from the experience. Perhaps it is the competitive nature of the industry that initiates the many duplicate efforts, but the TRW work specifically states that the results could aid test teams in testing by delineating a list of symptoms produced by errors, while the list of error causes could contribute to the generation of new testing aids.

A systematic error and correction procedure is a necessity in providing reliable software. However, the error reports examined for many projects were found to be lacking in detail, and correction or closure reports were not always filed in response to the errors found. The procedure for error reporting and correction reporting is often initiated late in the testing

process so that the project manager is unable to monitor software quality until it is almost too late to take effective action. This study recommends that the report vehicles used in the project's error reporting procedure be the same vehicles used by RADC to obtain error data. If the data collection forms for error reporting and error correction are supplied by RADC to contributing software development projects, the repository will benefit by:

a. Acquisition of real error data, not after-the-fact error data that may be contaminated;

b. Data to support analysis on the adequacy of collection forms;

c. The ability to request new parameters as the need for them arises.

The software development project also has benfits, including:

a. RADC's experience in establishing effective procedures and forms.

b. Effort saved by submission of the same error report form as was submitted internally by project members.

The Software Reliability Study performed by TRW [61] strongly recommended the establishment of a procedure for both the reporting and closure of errors, with an additional separation in procedure for the explanation of the fix and delivering the modified code. Also, the procedure and forms for accomplishing the error tracking job should be established before program implementation. Other recommendations for establishing an effective system include:

a. Assignment of specific personnel for monitoring the reporting procedure, and tracking the error(s) status.

b. Allocation of resources, either manpower costs for manual accounting or computer costs for automatic collection.

c. Development of problem report forms, including the following suggested data items:

1. Problem Reports (PR)

a. Problem Report Number - A unique PR number assigned by Configuration Management (CM); or the Program management Office (PMO).

b. Date - The date the PR is logged by CM, or PMO.

c. Time - Time of day the problem was discovered.

d. Originator - Name of employee submitting the PR.

e. Status - Status may be used to denote action taken, such as open, closed, deferred.

f. Test Phase - Test phase during which the problem was discovered, including: integration testing, system testing, validation testing, acceptance testing.

g. Problem ID - Identification or location of the problem, including module, data base, document, or other.

h. Priority - Indicates priority for fix, including low, medium, or high.

i. Element in Error - Name of the element exhibiting the problem. (If the specific module, data base, document is not known, identify subsystem.)

j. Mod - Modification of module, data base, etc., exhibiting the problem, if known.

k. Test Case - ID of the test case(s) which demonstrated the error, if available.

l. Problem Description - Description of the symptoms and, if possible, a description of the actual problem.

m. Means of Detection - Description of how error was discovered, including desk checking, personal communication, O/S error code, incorrect output, missing output, other.

n. Remarks - Additional information relevant to the problem.

2. Modification Reports (MR)

a. Modification Report Number - A unique MR number assigned by CM, or PMO, upon receipt of the modification code.

b. Date - Date the MR is logged by CM, or PMO.

c. Time - Time the MR and the delivered modification code are received by CM, or PMO.

d. Originator - Author of the modification (generally the individual closing the PR's).

e. PR Reference Number (s) - PR number(s) being totally or partially closed by this MR.

f. Response - Description of the correction being made to the software item. (In the case of a document or data base change, the document number or name and data base change are referenced with the description).

g. Source Code Type - Type(s) of source code involved in the modification, including input/output, computational or mathematical code, logical, control and data management.

h. Element - Name of module being modified, document being changed, or data base being altered. (In the case of a change to a document the title of the document to be changed is given. In the case of a data base change, the data base identifier is to be given.)

i. Old Mod/New Mod - Identification of old module modification to be altered to produce the new module modification.

j. PR Reference Evaluation - Identification of accuracy of the problem statement in the reference PR.

k. Reason for Modification - As near as possible, indicate type of error, including: Design Error, Documentation Error, Logic Error, I/O Error, Routine/Routine Interface Error, Computational Error, Data Handling Error, O/S System Support Interface Error, User Interface Error, Data Base Error, Global Data Definition Error, Routine/System Interface Error, Other.

l. Reason for No Modification - As near as possible indicate the reasons a modification was unnecessary, including: Operator Error, Documentation Error, Error Not Repeatable, Correction Deferred, Data Error, O/S Error, Support Software Error, Hardware Error, Other.

m. Estimated Resource for Diagnosis and Correction - Indicate effort required in Number of Computer Runs, Elapsed CPU Time, Man Hours Spent, Approximate Lines of Code (Added/Changed/Deleted).

n. Complexity of Module Corrected - Indicated the comlexity as easy, medium, hard.

o. Approval - Signature of PMO, if applicable.

4.3.3 Summary

At a minimum, the repository should manually collect quality control data resulting from formal or informal project reviews and data generated through an error reporting procedure. These data are necessary to support current RADC research requirements. In order to obtain meaningful data, the following recommendations are made:

- Collection and categorization of detailed data is an expensive and time consuming activity. Because of this, definitive goals should be made known to project personnel, as well as allocation of time and money for the activity. Detailed error categorization should be done only in support of specific requirements.
- The error reporting procedure established should directly support the stated and known objective of both the development project and the repository, as well as making a significant contribution to the project's configuration management requirements, if they exist.
- The error reporting procedure must be established prior to the project's program production, and must have proper resources allocated to it from initiation to completion of the activity.
- A research effort directed at analyzing the best method for reporting errors in order to establish a standard reporting system applicable to a wide variety of programming projects should be initiated; the result of which should be shared with participating development projects.
- Research in reliability analysis must establish a priori the specific data necessary to support the analysis.
- Continued examination of both the error reporting procedure and the error data must be done by the Software Data Repository in order to assure relevance of collected data to data requirements.

5. PRODUCT QUALITY MEASUREMENTS

In order to produce reliable software, money and time are being spent in large quantities to produce tools and improve techniques that will aid project personnel in the verification of the software. Papers presented at numerous conferences address the problem of rising software development costs without an equal rise in software reliability. To complicate the problems, many contracts are specifying that a certain percentage of the statements contained in the procured software system must have been successfully executed before final acceptance of the product by the customer. Still other contracts specify rigorous acceptance criteria before final delivery of the product. The customer has become cognizant of the pervasive problem of non-reliable software, resulting in the requirement or proof that the software system perform as specified. Unless there is a means by which to obtain that end and the project manager ensures product quality measurement, determination of the software reliability may be just an educated guess.

There are currently many tools and techniques available in the software industry to be used in the process of software development and validation. RADC is supporting the development of still other tools, designed to improve software quality while lowering the costs of program production. The objectives set forth by RADC include improving programmer productivity through the use of better programming languages, design, code and test techniques; improving management controls; and setting criteria for software quality by improvement of program readability, maintainability, portability and reliability. The Software Data Repository will provide a data base by which the data from the tools and techniques can be evaluated, software characteristics can be analyzed, and standards for quality can evolve.

Many of the tools used in the development of software generate output that could be used in a data repository, while the output from other tools can not currently be considered for that purpose. It appears that the data generated by some types of development tools is so finely detailed that specific research requirements for it need to be established before it can be systematically collected, summarized, filtered and stored in the

repository. This type of data may be essential for the repository to meet its goals, but the sheer volume and detail of the data may invalidate its utility to research unless there is a methodology for its acquisition, storage, retrieval, and purging. At the same time, it is important to relate the fine, product quality measurement data with both the project environment data (in order to define its development environment), and to the project performance data (in order to compare the performance to resources expended in the time allowed).

While it is possible to collect project environment data and project performance data by manual collection, it is nearly impossible to manually collect large volumes of data generated by development tools. This category of data, then, must be viewed as an evolving set of parameters, acquired as the tools are developed and integrated into the repository. Therefore, the exact structure, definition and content of these parameters must be obtained from the specifications and use of the tools.

For purposes of discussion only, the product quality measurements will be presented in relation to how this type of data is produced. There are two major methods for producing these data, including static analysis tools and the dynamic execution of the software under test conditions.

5.1 STATIC ANALYSIS TOOLS/AIDS

There are a number of tools available that collect data on the behavior and structure of modules, subsystems and systems without the execution of that software. Many of these tools are available as a part of existing compilers or operating systems. Unfortunately, there is little historical data supporting a conclusive recommendation that specific automatic aids contribute to quality software. There is growing evidence, however, that having available a set of automated aids, programming standards and a means to ensure the use of the standards have improved the overall performance of projects, as well as contributing to more reliable software [67].

5.1.1 Structural Characteristics of Modules

Recommendations by TRW [61] and others state that the structural characteristics of program code that can be measured are important and necessary for understanding the nature of software errors, and for providing a context for comparison of software projects. The analysis and accumulation of structural characteristics can be obtained by a syntax analysis program. The input to this type of program consists of the source code; the processing consists of examining the input stream according to the syntax rules of the specific source language in order to perform the desired function; the output consists of the structural analysis of the module of code, output as a subset of parameters to the primary processor capability. Programs that either currently provide this data, or that could be modified to provide it, include compilers, optimizers, instrumentation programs, code auditors, and complexity analyzers.

A display of structural characteristics can be done for individual modules of code or for entire systems of programs. These displays support analysis of both the set/use of data by system components and interface/communication links between system components. (A set/use listing of module data and a listing of internal/external data references are almost always an optional feature of compilers, while PPL and other library monitors provide listings of the programs contained in the system library, and other specialized tools for displaying system design, structure and communication links are becoming more widely used.) All of these tools provide project management and programming personnel with a mechanism for obtaining characteristics of the software before the execution of that software. Analysis of this data before program test execution can minimize program failures during testing, and also provide insight into the areas of poor design.

The structural characteristics of program modules that should be displayed and analyzed by programmers are more numerous than those required for analysis by project management and for the repository. However, summary reports of structural characteristics need to be generated for quality analysis and research-related work. These characteristics include the following:

a. Module name

b. COMPOOL dependency indicator
c. Subsystem name
d. Data entered into data base
e. Source language used
f. Last compilation date
g. Version and modification number
h. Routine size, including:
    1. Total source code statement
    2. Executable statements
    3. Non-executable statements
    4. Machine instructions
i. Number of branches
j. Number of external module interfaces
k. For each module called:
    1. Name of module
    2. Number of arguments in interface calls
l. Data interfaces, including:
    1. Number of global data blocks
    2. Number of internal data variables
m. Number of procedures
n. Number of entry points
o. Number of exit points
p. Routine code type, including:
    1. % computational
    2. % logical
    3. % I/O
q. Loop and nesting levels
r. Branch statement (IF)/nesting levels

There is another set of tools that has been used to demonstrate product quality through static analysis of the program code. These tools, code auditors of some type, are intended to ensure that programming standards established by project management or military organizations are observed. TRW has had a

successful application of this type of tool on the Site Defense Software [67]. Originally, 18 programming standards were established that were based on structured programming techniques, good programming practices, and the avoidance of error-prone code constructions. Every Site Defense program had to pass the analysis of the code auditor. It was found that this type of analysis increased the overall awareness of TRW personnel to what was needed in the growing field of software methodology. While individual program statistics generated by such programs as the code auditor can be unwieldy in the volume of data produced, the automatic enforcement of quality standards is an important data point to consider; and the contribution it makes to program quality measurement is obvious.

In the long run, the analysis of structural characteristics and program reliability may produce insight into the types of development tools necessary to improve software reliability. However, because of the large volume of data generated by these tools, the method for collection must be on an automatic, selective, and summary basis.

5.1.2 Data Definition and Communication

The data that programs use and set in order to perform their functions has always been a source of errors. Data definitions are not always complete; the wrong type of definition is used for the process; a supposedly common data definition differs between the routines that use them; access to data is not controlled, resulting in unexpected or erroneous value manipulation; and many other perturbations of data occur. The communication links between system components through the set and use of data is the basis for the development of multi-program software systems. If the data definition and communication is continually changing or in error, there is no basis for system reliability during the development stages of the software.

When the data structure or interface structure changes during program development, corresponding changes must be made to the modules which reference those

structures. The development of COMPOOL[1] facilities and COMMON[2] data spaces evolved in an attempt to minimize the problems inherent in data definition and communication. The use of such tools as a COMPOOL greatly enhances the visibility of data structures, and subsequently, of program reliability. It provides a means to establish overt control over the formation of the data base used by the system components, while providing public listings of the structure and content of the data base. Also, it may increase the project productivity by decreasing time required to alter program definitions.

Structured programming techniques address the problems of data relationships by limiting the access rights of modules to data structure and establishing rules for communication between modules. The concept of levels of abstraction restricts the data flow between levels [20,23]. The rules supporting the data communication in structured programming according to Bratman [12] include:

- Higher level of programs within the hierarchical system structure use lower levels of programs to obtain data.
- Explicit program arguments are passed down to a module from a higher level module while only values are returned to a higher level module.
- Each hierarchical level owns its data resources exclusively, so that changes in data structure are isolated from modules at other levels.
- Implicit interactions on common data structures may occur only within the modules belonging to the same hierarchical level.

[1]A COMPOOL provides for automatic definition and linkage of data variables used in program modules to the data base by a compiler.

[2]A COMMON declaration in programs provides for identification of data variables for information communication by automatic allocation of a region in memory to be shared by all programs containing the common declaration.

The use of tools and aids which attempt to illustrate the data structure in order to ensure a common use of the data, or limit the access to data, is an important aspect of ensuring product quality. Provided that these tools are used, they aid both programmers and management with problems associated with data definition and communication. However, the volume of data generated by these tools may make use of it infeasible for the repository. As an alternative, it can be stored as hard copy in the research library. Summarization of this type of data may invalidate its utility for research, although this possibility should be more thoroughly examined.

5.1.3 Module Logic Evaluation Tools

Although the tools available for evaluation of the program's logical, or computational, structure are not widely used in the software industry because of their lack of availability and unreliability, there is a growing need for them. Until recently, determining the complexity of a program's logic was a subjective estimation, based partially on the estimator's experience and partially on the intended function of the program. Currently, RADC is supporting work to develop a tool that analyzes program structure in order to evaluate complexity [7]. The basis for this work is the transformation of a program to a directed graph; the same principles being applied here as in code optimization programs [50], and automatic test case generation programs [25]. At this point, the use of this type of complexity measure program cannot be used by project management, programmers, or the repository to determine the quality of computational and logical code because of the experimental nature of the tool. This information could become an important asset to the repository and/or software projects. Because of this, the structure of the data base should be flexible enough to allow acquisition of this data, and other state-of-the-art tool data when it becomes available.

5.2 EXECUTION ANALYSIS TOOLS

Other existing tools provide information about the actual execution of the software in either a live or simulated environment. This classification of data allows the user to more effectively evaluate whether the program has performed according to a predetermined set of performance criteria. The use of

these tools in the development of software generally follows module level testing, although the time of their employment depends upon the management decisions or the construction of the individual program system. Many of the execution tools are geared to a specific language for a specific computer, and are, therefore, not reusable. (Reusable in this context is defined as having the capability of being extracted from their original environment and incorporated into a new environment without reprogramming.)

Many of the execution analysis tools generate an extremely large amount of execution data, such as the common trace programs used in debugging. The availability of the tools to project personnel is an important project environment factor to consider in relation to productivity, resource expenditure, and eventual software reliability; the use of the data generated by the tools is an important aid to personnel. However, the amount of data generated by such tools is so voluminous as to be almost nonusable for the data repository. If the data produced by these tools can be automatically acquired and summarized, the summary information would be an important asset to research studies.

5.2.1 Automatic Execution Analysis Systems

Most of the automatic execution analysis tools in the software industry, are designed to instrument[1] a program written in a specific source language, execute the program with user supplied input test data, and record dynamic program operation by the way of instrumentation and recording. Depending upon the size of the object module and the amount of test data provided by the user, these programs can provide a staggering amount of output, but even so, just sufficient to permit the programmer with enough data to determine exactly which program statements have been executed. This data can be used to generate a more exhaustive set of test cases, or determine that the code itself is superfluous or in error. It should be noted that by using this type of tool the project manager is able to determine the "testedness" of the pro-

[1] Instrumentation is the process of generating and inserting recording instructions at strategic program locations during a statis syntax analysis of the program. The modified program is subsequently compiled and linked with recording programs. The instrumentation of the source program is, in effect, transparent to the user.

gram, which indirectly impacts the overall program quality. It does not prove the program correct in any way.

While execution analysis tools are an effective contribution for aiding the programmer and project manager in determining qualities of program testing, the everyday use of these tools is not wide spread in the industry. Software companies which have these tools available usually must require programmers to use them. There are a number of reasons for the general nonacceptance of the tool. It does require time and effort from the programmer to become familiar with the automatic execution system operation. The output generated from a moderately sized program with a minimum set of test cases takes a great deal of time to analyze and understand. The tools themselves are not altogether reliable. An instrumented program may take as much as 75-85% longer to operate on a single test case than the non-instrumented version of the program on the same test case, so that the use of the tool appears to be quite costly. (It may save a great deal of later expense in resolving undiscovered errors, however). Even so, it does provide an effective contribution to product quality measurement.

5.2.2 Test Drivers/Test Data Generators

Many software systems, especially real-time programs, must be checked out by the use of test drivers and simulated test data. Some, as has been done by NASA for space born systems and by the Satellite Control Facility in its STAGES (Station Ground Environment System), are even checked out using complex hardware and software simulating the operational environment of the application software. These types of tools provide dynamic program operation within a controlled environment, allowing the user to evaluate the actual operation of the program against expected results. Generally, the individual test drivers and test data generation tools[1] are designed for specific

[1]In this context, test data generation tools do not include automatic test case generators that analyze a source program in order to provide input data information sufficient enough to exercise all statements of the program. Rather, these test data generators work on the principle of supplying data for files, or input buffer streams, such as simulated telemetry data input.

applications. The results aid project personnel in determining the performance of the program, and in turn, the overall quality of the program. It must be emphasized, however, that the test drivers and test generators are software systems themselves, and do not guarantee reliability in their own performance.

Several factors must be considered in both the production and use of test drivers and/or test data generators. The test drivers must be constructed so that they operate the application software modules in as nearly the same manner as the live environment would. The test data generated for data files must contain a variety of values, especially in minimum/maximum ranges. The distribution of values generated should be equal, as nearly as possible, to the expected distribution of actual values. A random sample of other values is also important for test purposes. Also, values should be generated that exceed the boundary range of values in order to test error conditions.

It is recommended that the data output by test drivers/test data generators should not be collected for the data repository because of the large and widely diversified volume of data generated by the tools. The quality and availability of these tools, however, may be an important consideration for the data repository as a project environment parameter.

5.2.3 Dynamic Analyses of System Structure Tools

There are many tools which output listings of the system or subsystem structure when it is prepared for execution, such as link editors and loaders. These listings provide information on the contents of each load module. Often, they specify the external references made by each module within the system. Other listings and catalogs of this nature are output automatically by program production library monitors. An important program of the SAGE System was the Automatic Core Allocator which output a core map of the subsystem program each time there was a new core allocation. These types of listings provide important visibility into the software system components, and often, they include information on the interrelationships of the program modules. This data appears to be essential information for the Software Data Repository,

and would have wide applicability for all projects participating in the data collection effort.

5.2.4 Operating and Performance Measurement Tools

Performance and operating measurements became important with the development of second generation solid state computers, which provided faster computational capability, as well as more efficient input-output peripheral equipment. The overall efficiency and productivity of software systems increased because of the ability to perform input/output functions simultaneously with the execution of program instructions through the use of operating systems. The third generation computers required even more complex operations and control of operating systems since the computer equipment was modular in design. Besides the computer configuration, application programming became more sophisticated, communicating with the computer system from a distance, i.e., remote access; communicating interactively with the computer, i.e., online processing; and communicating with the computer during the time frame in which an actual event occurs, i.e., real time processing. The problems of managing this wide scope of computational abilities made it imperative to develop a set of methods, techniques and tools which measured and evaluated the overall quality of the software systems, both being developed and operational.

While software and hardware measurement techniques have become quite sophisticated and have greatly affected the productivity of software system, the use of these techniques has been most widespread in evaluating the acquisition of hardware and/or software. The efficiency of a set of coded instructions is not as important today as it has been in the past, due, in part, to the speed of the existing computers. The demands for reliability of the software are far greater than the demands for efficiency from the individual programmer, as evidenced by the growing acceptance of structured programming techniques.

The development and use of tools for performance evaluation requires that the parameters impacting individual system performance be identified, as well as the interaction and dependence of those parameters upon each other. Once these critical performance parameters are established, controlled measurement and study contributes to the performance evaluation of the system. The

measurement and evaluation techniques that are currently in existance today, such as utilizing kernel programs, kiviat figures, instruction mix analyses, etc., contribute to improving the system cost effectiveness. However, the individual project manager rarely has the knowledge or responsibility to perform evaluations of this type, unless there is a specific task defined for this purpose.

A data gathering facility designed to collect information about the performance of individual jobs, rather than the performance of the software itself, is an important aspect of monitoring software projects. This facility is accessable to project management in many forms. There are both automatic and manual methods of obtaining and logging the job performance information. The PPL concept is one such method that attempts to combine the automatically collected data available to the individual computer installation with manually collected data, as necessary. In order to perform analysis on the problems in software development, either from the point of view of management to control costs and performance, or by the researcher to analyze data for any number of objectives, a monitor appended to the operating system to collect data about critical activities appears to be a most effective means of obtaining reliable and consistent data. The monitor can collect data as events occur, and store the information for subsequent processing. The data, when tabulated later, represent an accurate log of system operations. The amount and detail of the data collected should support only the specific objective for which it will be used.

A software monitor that collects data on system operations will cost money, not only in the development of the software, but in terms of the day-to-day computer run costs, storage costs, processing of the collected data, etc. Schwetman [51] presents a summary of the overhead of a probe used at Purdue University. This probe was designed to provide detailed data on the behavior of system operation, in addition to common computer usage statistics. The overhead statistics of the Purdue probe for the CDC 6500 computer are as follows:

Summary of Probe Overhead

Additional CM	456 words
Additional Time, probe inactive	
per PPU Function	1.3 μsec
per CPU Assignment	1.5 μsec
per MSA Function	4.6-6.8 μsec
Additional Time, probe active	
per Recorded Event	50-55 sec

Summary of Usage Statistics for Data Collection Program

Date	02/06/76
Starting Time	10:32:31
Elapsed Time	1008.531 seconds
Central Memory	5952 seconds
CPU Time	130.199 seconds
Events Received	1027409 events
Events Transmitted	767832 events

While Schwetman concludes that the data collection monitor used does impose an additional load on the system, the "benefits outweigh the costs which are incurred". These costs, however, are directly related to the type and amount of data collected. The benefits that Purdue has realized from the installation of the probe has been greatest for system designers and programmers by providing a detailed look at the system operation, and its impact on system design and implementation. It has also provided data on system inefficiency, which subsequently was improved; and on parameter values to be used by researchers in simulation modeling of timesharing behavior.

The data collected automatically on system operations and errors, computer usage, efficiency of processing, etc., is valuable and important data to measure program quality for both project management and the repository. Some of this data can be collected manually, but manual collection should be used only as a temporary method until an automatic method can be developed.

5.3 SUMMARY

The product quality measurements are that data demonstrating the product structure and behavior through the application of analysis tools and testing procedures. This category of data deals with very fine measures of the soft-

ware product and/or summations of those fine measures. This category of data should be collected automatically by support tools, software probes, or other instrumentation devices, as they are developed. The exact structure, definition, and content of data to be collected depend upon the specifications of the tools and the computer devices on which they are developed. It is recommended that RADC incorporate this type of data in the data base on a selective basis as the requirement for that data is realized. Therefore, the structure of the data base and data collection system should be flexible to support the acquisition of these data.

(The reverse of this page is blank)

BIBLIOGRAPHY

(1) AFSCP/AFLCP 173-5, AMCP 37-5, NAVMAT P5240, "Cost/Schedule Control Systems Criteria Joint Implementation Guide", March 31, 1972.

(2) Aron, J. D., "Estimating Resources for Large Programming Systems", IBM-FSD, Gaithersburg, Maryland.

(3) Aron, Joel D., The Program Development Process. Phillipines: Addison-Wesley, 1974.

(4) Baker, F. T., "Chief Programmer Team Management of Production Programming", IBM September Journal, No. 1, 1972.

(5) Baker, F. T., "System Quality Through Structured Programming", Proceedings FJCC 1972.

(6) Bakkegard, I. "Quantitative Data Base" TM-HU-195/000/00, System Development Corp., Santa Monica, Calif., April 1975.

(7) Bell, D.E., Sullivan, J.E., "Further Investigations into the Complexity of Software", MTR-2874, Volume II, MITRE Corp., Bedford Massachusetts, June 1974.

(8) Boehm, B. M. "Software and its Impact: A Quantitative Estimate," Datamation, May 1973.

(9) Book, E., Schaefer, M., Reemsnyder, M., and Willmorth, N. E. "A Software Systems Engineering Study." TM-5157/500/01, SDC July 1973.

(10) Book, E., "VMM Software Production Method Analyses", TM-5443/007/00, System Development Corp., Santa Monica, Calif., June 1975.

(11) Brandon, D. H., "The Economics of Computer Programming", in G.F. Weinwurm (ed.), On the Management of Computer Programming. Philadelphia: Auerbach, 1971.

(12) Bratman, H., "Structured Programming: Techniques for Developing Reliable Software Systems", SP-3693, System Development Corp., Santa Monica, Calif., December 1972.

(13) Bratman, H., "The SDC Software Factory", TM-5175/100/00, July 1973.

(14) Brooks, Frederick P., The Mythical Man-Month. Philippines: Addison-Wesley, 1975.

(15) Cammack, W. B., Rodgers, Jr., H.J., "Improving the Programming Process", TR00.2483, IBM, Poughkeepsie, New York, 1973.

(16) Cheng, R., Sullivan, J. E., "Case Studies in Software Design", MTR-2874, MITRE Corp., Bedford, Massachusetts, June 1974.

(17) Clewlow, Carl W., "Managing Camputer Programming in the Federal Government", in G. F. Weinwurm (ed.), On the Management of Computer Programming. Philadelphia: Auerbach, 1971.

(18) Corrigan, A. E., et al, "Specifications for SIMON, a Software Implementation Monitor", MTR-3056, MITRE Corp., Bedford, Massachusetts, July 1975.

(19) Diesen, C. F., "People Problems in Government EDP Work", in F. Gruenberger (ed.), The EDP Proper Problem. Los Angeles: Data Processing Digest, 1971

(20) Dijkstra, F. W., "Notes on Structured Programming", Technische Hogeschoal, Eindhoven, The Netherlands, August 1969.

(21) Ellingson, O. E., "A Predictor Tool for Estimating the Confidence Level of a Computer Program Subsystem in the Space Programs Department", TM 3335, System Development Corp., Santa Monica, Calif., January 1965, May 1967.

(22) Erickson, Warren Jr., "The Effect of Operating Systems on the Cost-Effectiveness of Computer Programming", in F. Gruenberger (ed.), The EDP People Problem. Los Angeles: Data Processing Digest, 1971.

(23) Ershov, A. P., "Aesthetics and the Human Factor in Programming", in Communications of the ACM, Volume 15, Number 7, July 1972.

(24) Ewart, R. F. and Nanney, D. M. An Analysis of Program Evaluation and Review Technique (PERT) in Weapons System Acquisition. Master's Thesis, Air Univ., Wright-Patterson AFB, Sept., 1974.

(25) Finfer, M. C., Automatic Test Case Generation Program, Research Work Performed at System Development Corp., Santa Monica, Calif., 1974.

(26) Fleishman, T., "Current Results from the Analysis of Cost Data for Computer Programming", AD 637801, System Development Corp., Santa Monica, Calif., August 1966.

(27) Geran, Daniel B., "Summary Notes of a Government/Industry Software Sizing and Costing Workshop", USAFESD, Bedford, Mass., October 1974.

(28) HIPO - A Design Aid and Documentation Technique, GC 20-1851-0, IBM, October 1974.

(29) Hosier, W. A. "Pitfalls and Safeguards in Real-Time Digital Systems with Emphasis on Programming," IRE Trans on Engineering Management, June 1961.

(30) Hudson, G. R., "Program Errors as a Birth-and-Death Process", SP-3011, System Development Corp., Santa Monica, Calif., December 1967.

(31) James, T. A., "Programming Language Characteristics and Comparison Reference", Advanced Software Techniques for Data Management Systems, Volume III, February 1972.

(32) Liskof, B.H., "Guidelines for the Design and Implementation of Reliable Software Systems", MTR-2345, MITRE Corp., Bedford, Massachusetts, April 1972.

(33) Littlewod, B., Verrall, J. L., "A Bayesian Reliability Growth Model for Computer Software", in Proc. Symposium on Computer Software Reliability, IEEE, 1973.

(34) McHenry, Robert C., "Software Development Process Revisions", Digest of Papers, COMPCON 75, TEEE, September, 1975.

(35) Metzger, Philip W., FSD Programming Project Management Guide, IBM-FSD, Gaithersburg, Maryland, 1970.

(36) Meyers, G. J., Reliable Software through Composite Design, Mason/Charter, London, 1975.

(37) Miller, E.F., "A Survey of Major Techniques of Program Validation", General Research Corp., Santa Barbara, Calif., October 1972.

(38) Nelson, E.A., "Management Handbook for the Estimation of Computer Programming Costs", AD 648 750, System Development Corp., Santa Monica, Calif., March 1967.

(39) Ogdin, Jerry L., "The Mongolian Hordes versus Superprogrammers", Infosystems, December 1972.

(40) PACE, Product Assurance Confidence Evaluator Tool, developed by TRW, Redondo Beach, Calif.

(41) PATH, Program Analysis and a Test Host Tool, developed by System Development Corp., Santa Monica, Calif.

(42) Patterson, J. W., IBM Corp. Gaithersburg, Md., private communication.

(43) Peck, M. J. and Scherer, F. M.," The Weapons Acquisition Process: An Economic Analysis," School of Business Administration, Harvard Univ., 1962.

(44) Pietrasanta, Alfred M., "Resource Analysis of Computer Program System Development", in G. F. Weinwurm (ed.), On the Management of Computer Programming. Philadelphia: Auerbach, 1971.

(45) Pietrasanta, A. M., "Two Empirical Studies of Program Production", Proc. IFIP Congress 1968, North-Holland Pub. Co., Amsterdam, 1968.

(46) Qualifier, Automatic Execution Analysis Tool developed by Computer Software Analysts, Inc., Los Angeles, Calif.

(47) Rubey, R. J., "Quantitative Aspects of Software Validation", in Proc. International Conference on Reliable Software, IEEE, April 1975.

(48) RXVP, FORTRAN Automatic Verification System, developed by General Research Corp., Santa Barbara, Calif.

(49) Sackman, H., Erikson, W. J., Grant, E. E., "Exploratory Experimental Studies Comparing Online and Offline Programming Performance", SP-2687, System Development Corp., Santa Monica, Calif., December 1966.

(50) Schaefer, Marvin, A Mathematical Theory of Global Program Optimization. Printice-Hall, Englewood Cliffs, New Jersey, 1973.

(51) Schwetman, H. D., "Gathering and Analyzing Data from a Computer System: A Case Study", in Proc. of the Annual Conference, ACM, October 1975.

(52) Scott, R., "A Computer Programmer Productivity Prediction Model", University Microfilm, Ann Arbor, Mich., December 1973.

(53) Scott, Randall F., Simmons, D.B., "Programmer Productivity and the Delphi Technique", Datamation, May 1974.

(54) Searle, L. V., Neil, G., and Benson, S. G., "Computer Program Acquisition: Data Requirements", TM-2547/000/00, SDC, July 1965.

(55) Searle, L. V., Neil, G., and Benson, S. G., "Configuration Management of Computer Programs for Information Systems", TM-1918/000/01, July 1965, (Org. pub. June, '64).

(56) Shelley, Marlin, "Computer Software Reliability; Fact or Myth?", Ogden Air Logistics Center, Hill, Utah, November 1973.

(57) Shooman, M. L., "Quantitative Analysis of Software Reliability", Proc. IEEE Reliability Symposium, January 1972.

(58) Slaughter, J. B., Small, A., Speirman, K., Steele, S. A., "Proc. of a Symposium on the High Cost of Software", Air Force Office of Scientific Research, Monterey, Calif., Sept., 1973.

(59) Smith, Ronald L., "Management Data Collection and Reporting", Volume IX in Structured Programming Series, IBM, Gaithersburg, Marlyland, Oct., 1974.

(60) Trauboth, H., "Guidelines for Documentation of Scientific Software Systems", in Proc. Symposium on Computer Software Reliability, IEEE, May 1973.

(61) TRW Systems Group, "Software Reliability Study", Interim Report, RADC-TR-74-250, October 1974.

(62) Very High Level Languages Symposium, Santa Monica, Ca., March 1974.

(63) Wagoner, W. L., "The Final Report on a Software Reliability Measurement Study", Report No. TOR-0074(4112)-1, Aerospace Corp., Redondo Beach, Calif., August 1973.

(64) Wasserman, A. I., "A Top-Down View of Software Engineering", in Proc. of the 1st Nat'l Conference on Software Engineering, IEEE, September 1975.

(65) Weinwurm, George F., "On the Economic Analysis of Computer Programming", in Weinwurm, G. F., (ed.), On the Management of Computer Programming. Philadelphia: Auerbach, 1971.

(66) Willmorth, N. E., "Integrated Management, Project Accounting and Control Techniques", System Development Corp., August 1975.

(67) Williams, R. D., "Managing the Development of Reliable Software", in Proc. Int'l Conference on Reliable Software, IEEE, April 1975.

(68) Willmorth, N. E., "Managing an Acre of Programmers", in Gruenberger F., (ed.), The EDP People Problem. Los Angeles: Data Processing Digest, 1971.

(69) Willmorth, N. E.,"System Programming Management, The Organization of Work", TM-2222/003/00 SDC, May 1965.

(70) Wolverton, Ray W., "The Cost of Developing Large-Scale Software", IEEE, Trans. on Computers, June 1974.

(71) Anderson, R. M., "Anguish in the Defense Industry," Harvard Business Review, December, 1969.

(72) EST-TR-66-673, Air Force ADP Experience Handbook (Pilot Version), A.J. Gradwohl, et al, PRC, December 1966.

ESD-TR-66-672, Primer for Air Force ADP Experience Handbook (Pilot Version), A.J. Gradwohl, et al, PRC, December 1966.

ESD-TR-66-671, Phase II Final Report on the Use of Air Force ADP Experience to Assist Air Force ADP Management, A.J. Gradwohl, G.S. Bechwith, S.H. Wong, W.O. Wootan Jr, PRC, 1966
Volume 1 - Summary, Conclusions and Recommendations
Volume 2 - Phase II Activities
Volume 3 - Phase III Concepts and Plan

(The reverse of this page is blank)

☆U.S. GOVERNMENT PRINTING OFFICE: 1977-714-038/88

MISSION
of
Rome Air Development Center

RADC plans and conducts research, exploratory and advanced development programs in command, control, and communications (C^3) activities, and in the C^3 areas of information sciences and intelligence. The principal technical mission areas are communications, electromagnetic guidance and control, surveillance of ground and aerospace objects, intelligence data collection and handling, information system technology, ionospheric propagation, solid state sciences, microwave physics and electronic reliability, maintainability and compatibility.