AD-A053 020 GENERAL RESEARCH CORP SANTA BARBARA CALIF F/G 9/2
COST REPORTING ELEMENTS AND ACTIVITY COST TRADEOFFS FOR DEFENSE--ETC(U)
MAY 77 C A GRAVER, W M CARRIERE F19628-76-C-0180
UNCLASSIFIED CR-1-721-VOL-1 ESD-TR-77-262-VOL-1 NL

1 OF 4
AD
AD53020

ESD-TR-77-262, Vol. I

COST REPORTING ELEMENTS AND ACTIVITY COST TRADEOFFS FOR DEFENSE SYSTEM SOFTWARE (STUDY RESULTS)

General Research Corporation
P.O. Box 3587
Santa Barbara, CA 93105

May 1977

Approved for Public Release;
Distribution Unlimited.

Prepared for

DEPUTY FOR COMMAND AND MANAGEMENT SYSTEMS
ELECTRONIC SYSTEMS DIVISION
HANSCOM AIR FORCE BASE, MA 01731

LEGAL NOTICE

When U.S. Government drawings, specifications or other data are used for any purpose other than a definitely related government procurement operation, the government thereby incurs no responsibility nor any obligation whatsoever; and the fact that the government may have formulated, furnished, or in any way supplied the said drawings, specifications, or other data is not to be regarded by implication or otherwise as in any manner licensing the holder or any other person or conveying any rights or permission to manufacture, use, or sell any patented invention that may in any way be related thereto.

OTHER NOTICES

Do not return this copy. Retain or destroy.

This technical report has been reviewed and is approved for publication.

William J. White

WILLIAM J. WHITE, Captain, USAF
Project Engineer

John T. Holland

JOHN T. HOLLAND, Lt Col, USAF
Chief, Techniques Engineering Division

FOR THE COMMANDER

Toru Yamamoto

TORU YAMAMOTO, Colonel, USAF
Director, Computer Systems Engineering
Deputy for Command & Management Systems

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: ESD-TR-77-262-Vol-1	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle): Cost Reporting Elements and Activity Cost Trade-offs for Defense System Software. Volume I. Study Results.	5. TYPE OF REPORT & PERIOD COVERED: Final Report. 11 Mar 76 - 11 Mar 77
6. PERFORMING ORG. REPORT NUMBER: CR-1-721-Vol-1	
7. AUTHOR(s): C.A. Graver, W.M. Carriere, E.E. Balkovich, R. Thibodeau	8. CONTRACT OR GRANT NUMBER(s): F19628-76-C-0180 new
9. PERFORMING ORGANIZATION NAME AND ADDRESS: General Research Corporation, P.O. Box 3587, Santa Barbara, CA 93105	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: PE63101F, Project E234
11. CONTROLLING OFFICE NAME AND ADDRESS: Electronic Systems Division, Air Force Systems Command, Hanscom AFB, MA 01731	12. REPORT DATE: May 77
	13. NUMBER OF PAGES: 306 284 p.
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): Same	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)

18. SUPPLEMENTARY NOTES

ESD Project Engineer: Captain William J. White (MCI)

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

Software	Software Life Cycle	Software Size
Costs	Software Development	Computer Program
Reporting System	Software Maintenance	Components
Software Costs	Software Manhour Estimates	
Software Reporting System	Software Product Description	

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

This technical report examines the costs of developing and maintaining computer software for major defense systems. A process model is described which depicts the relations among activites and phases of the software life cycle, identifies the product and cost information that is normally available, and specifies the milestones. This process model is normally used as the basis for selecting the elements of a software cost reporting system. The suggested reporting system also includes descriptions of the final product, time phasing of product development, a standardized list of Computer Program Components, and a standardized

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

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

402 754

AD

(Block 20 continued)

list of labor categories.

During the study, data was collected from several sources including the following Air Force organizations:

Electronic Systems Division
Aeronautical Systems Division
Space and Missile Systems Organization
Data Systems Design Center

Cost estimating relationships for each phase of the software life cycle are explored, using the process model and the data. The importance of trade-offs in cost between phases is demonstrated. The report also contains estimating relationships for evaluating the cost effects of software size, computer capacity constraints, programming language, and changes in requirements. It also addresses the separation of two activities, error correction and product improvement, during the maintenance phase of the life cycle. Results are integrated with other software cost estimating techniques.

PREFACE

In April 1976, General Research Corporation (GRC) began a study of "Life-Cycle Costing of Major Defense System Software and Computer Resources," Contract F19628-76-C-0180. The purpose was to assist Air Force Program Offices and staff agencies in estimating, reporting and controlling the life-cycle costs of software. The study was performed under the direction of the Electronic Systems Division (AFSC), Computer Systems Engineering Office (TOI). Captain William White was the Project Officer and coordinator of the data-collection survey. The project team wishes to thank him for his many contributions.

The project team was also assisted by many other individuals, to whom we owe our thanks. Among them the following individuals made possible the detailed data collection: Captain J. M. Hall, Captain Baden, and Lt. Anita Cohen of SAMSO (SCF); Ray Erickson, Joe Thompson, Chuck Chiodini, and Everton Griffith of System Development Corporation; and Eugene Kelly, Major Dale Wooldridge, and Thomas Kennedy of the US Air Force Data Systems Design Center.

ABSTRACT

This technical report examines the costs of developing and maintaining computer software for major defense systems. A process model is described which depicts the relations among activities and phases of the software life cycle, identifies the product and cost information that is normally available, and specifies the milestones. This process model is used as the basis for selecting the elements of a software cost reporting system. The suggested reporting system also includes descriptions of the final product, time phasing of product development, a standardized list of Computer Program Components, and a standardized list of labor categories.

During the study, data was collected from several sources including the following Air Force organizations:

Electronic Systems Division
Avionics Systems Division
Space and Missile Systems Organization
Data Systems Design Center

Cost estimating relationships for each phase of the software life cycle are explored, using the process model and the data. The importance of trade-offs in cost between phases is demonstrated. The report also contains estimating relationships for evaluating the cost effects of software size, computer capacity constraints, programming language, and changes in requirements. It also addresses the separation of two activities, error correction and product improvement, during the maintenance phase of the life cycle. Results are integrated with other software cost estimating techniques.

ABBREVIATIONS

ADP	Automated Data Processing
ADPREP	ADP Resource Estimating Procedure (Ref. 12)
AFDSDC	Air Force Data Systems Design Center (Gunter AFS, Alabama)
AOES	Advanced Orbital Ephemeris Subsystem (part of SCF)
CDR	Critical Design Review
CER	Cost Estimating Relationship
CPC	Computer Program Component
CPCI	Computer Program Configuration Item
DCR	Design Change Request (similar to ECP)
DRF	Discrepancy Report Form
DSP	Defense Support Program
ECP	Engineering Change Proposal
FCA	Functional Configuration Audit
FQT	Final Qualification Tests
GFE	Government Furnished Equipment
IOT&E	Initial Operational Test and Evaluation
PARMIS	Planning and Resources Management Information System
PBC	Program Breakdown Code
PCA	Physical Configuration Audit
PDR	Preliminary Design Review
PON	Project Originator Number
PQT	Preliminary Qualification Tests
SCF	Satellite Control Facility
SCN	Specification Change Notice
SDR	System Design Review
SPS	Simplified Processing System (part of DSP)
WBS	Work Breakdown Structure

CONTENTS

CONTENTS (Continued)

CONTENTS (Continued)

ILLUSTRATIONS

ILLUSTRATIONS (Contd.)

TABLES

TABLES (Continued)

1 INTRODUCTION

1.1 GOALS OF THE STUDY

The two main goals of the study were:

To define the elements of a system for reporting the costs of developing and maintaining computer software

To develop estimating relationships for the resources (man-hours, computer-hours, elapsed time) consumed in the different phases of the software life cycle.

These goals reflect a critical need for improving the Air Force's ability to estimate the cost of software.

The requirement for good software cost-estimating techniques has been increasing in the last decade as defense systems increased in sophistication and in their dependence upon computers. More and more of the defense dollar is going into the development and maintenance of software; as Captain Devenny cit es in his thesis,[1] "DoD program managers will buy an estimated three billion dollars worth of software in 1976."

As a result, what used to be a minor part of a system's acquisition cost, requiring no more than gross rule-of-thumb estimating, is now a major procurement item, requiring more accurate techniques. Thus, the goals of this study are of great importance.

It is truly unfortunate that the first goal is still a requirement. As early as 1966, reporting systems were being devised for collecting software cost data. Weinwurm[2] described the data elements of such a system in that year, and Nelson and Fleishman expanded Weinwurm's work into a reporting system, including forms, formats, etc.[3] These early efforts were companions to an extensive effort at System Development Corporation (SDC) to develop cost estimating relationships for software.[4,5,6]

More recently, Ronald A. Smith, in a study for Rome Air Development Center, reported on a suggested "Management Data Collection and Reporting System",[7] prepared for RADC in October 1974. The report is part of the structured programming series, and his system makes use of the library concept included in that technique. The library concept is a powerful data gathering and configuration management tool that makes it technically possible to gather very detailed data about the status of a software development.

So far, however, no uniform reporting system has been implemented by DoD. As a result, "There is no widely accessible collection of cost data which can be applied to cost estimation." (Ref. 8, page 11). Considering that it takes 10 to 15 years for some large defense systems to complete their life cycle, complete cost data would not be available for a long time, even if a reporting system were implemented today. However, if one had been implemented in 1966, useful data would be available today.

A more immediate need that a reporting system can fulfill is to provide the Air Force's Program Offices with better information for cost control. As Devenny[1] adequately demonstrates in his thesis, software developments rarely finish within their budgets. One major reason is the poor quality of information available during the development, which makes it nearly impossible to spot problems at early stages.

Why has no uniform system for data collection yet been implemented? For one thing, the small part of the budget devoted to software in the past probably discouraged any attempt to finance such a reporting system. However, we speculate that a significant reason is the unwillingness of the contractors to provide data in as much detail as the proposed systems have required. What a contractor is technically able and willing to report internally for management control, e.g., by using the Smith system[7] mentioned above, could be far more than what he is willing to share

cheerfully with the Government. Thus, care should be taken to avoid asking for excessive detail in any cost reporting system that one hopes to implement.

The second major goal of this study was to improve the general understanding of software cost estimating. This too has had a long history of attempts by competent groups with little success. As Morin[9] noted in her review of software cost estimating techniques, "While a number of efforts have been made to develop improved cost estimating techniques, no generally accurate nor reliable method for estimating software development costs has been found."

In general, previous attempts (by SDC,[4,5,6] Tecolote,[10] and others) had concentrated on total cost. Their lack of success was not due to a lack of competence; many imaginative relationships were developed and tested. Unfortunately, variances remained too large to be useful for estimation, as witnessed by the fact that these groups did not recommend the use of their findings to estimate software cost.[1]

In an effort to plow new ground, we were directed to develop resource estimating relationships (for man-hours, computer-hours, and elapsed time) for each phase of the life cycle separately, rather than total man-hours or costs. The relationships were to be based upon currently available data and developed using parametric cost estimating techniques.

The following principles evolved during the course of the study and guided our progress towards achieving the study's goals.

First, both the recommended cost reporting system and the estimating relationships must be related to the Air Force's procurement process. Attempts now underway to standardize this process are reported in AFR 800-14,[11] which we have used as our guide in developing a process model, tempered by visits to Program Offices and by our own experience.

Second, the cost reporting system must meet several objectives. It must provide information which will help the Program Offices to control software costs. At the same time it must be used to compile cost data on many systems, using standard definitions, so that better cost estimating relationships can eventually be developed. Furthermore, the system must not be so detailed that it is impracticable, and it must be based on items that are directly measurable by contractors; for example, payroll and configuration-control data.

Third, resource estimating relationships for man-hours, computer-hours, and elapsed time should be developed separately for each phase of the software life cycle, using parametric cost estimating techniques.* The relationships should depend on inputs normally available to the Air Force. A relationship which depends on input data not available to the Program Office (for example, competence of the individual programmer) might be statistically valid, but would be useless in practice. Since the relationships must be validated by using currently available data, they will have to be based upon less detailed information than that proposed to be collected with the reporting system.

1.2 PROGRESS OF THE STUDY

Five tasks were identified in the Statement of Work. Task 1 was to conduct a literature survey to develop lists of the human and computer resources required to develop and maintain software.

---

*As will be demonstrated, addressing the life-cycle phases separately was not sufficient; the trade-offs between phases are of utmost importance in determining total (and phase) resource requirements. Future study of these relations between phases should also lead to information that will identify efficient allocations of resources among the phases. It is striking that the literature is full of references to the "40-20-40" rule: 40% analysis and design, 20% coding and checkout, and 40% integration and test. Having 40% of the work still ahead after all of the code has been written makes one wonder whether enough time is spent in analysis and design. Discovering the optimal level of analysis and design should go a long way to reduce the risk associated with software development.

Building upon this review and our own experience in software development. we were to define the cost elements of a recommended reporting system (Task 2) and relate them to a typical software work breakdown structure (WBS) (Task 3).

During the study, data would be collected for verifying the usability of the WBS for gathering and reporting cost performance data (Task 4), and the cost elements and WBS would be modified as necessary. The data were also to be used in developing estimating relationships.

Finally, we were to develop estimating relationships for man-hours, computer-hours, and elapsed time for each of the life-cycle phases (Task 5). Of particular concern were relationships of life-cycle costs to system characteristics and design parameters; differentiating between alternative designs; performing trade-offs between development and maintenance costs; evaluating the effect of Engineering Change Proposals (ECPs) on costs; and estimating the cost consequences of interface-equipment constraints.

By the first Technical Direction meeting (June 22) GRC had made the following progress. An initial human-resource list and computer-resource list had been completed (Task 1) as well as an initial cost category list (Task 2). Seven Air Force Program Offices had been visited (three at ESD, two at ASD, and two at SAMSO)* for the purpose of determining data availability (Task 4), management cost-control problems (Task 2), and typical WBSs (Task 3). Finally, a Process Model had been developed which

---

*The following Program Offices were visited:

ESD
- Cheyenne Mountain (427-M)
- Combat Grande
- Over the Horizon Backscatter Radar

ASD
- EF-111A
- F-16

SAMSO
- Satellite Central Facility (SCF)
- Defense Support Program (DSP)

identified the phases of Air Force software development and maintenance and the information available at each phase (Task 5).

The results of the visits to Program Offices were extremely important in shaping the study's direction. They were especially significant in defining the elements of the recommended reporting system. The Program Offices have a difficult problem in controlling software cost. A major reason is that software costs are reported at far too aggregated a level--in some cases as only a single cost. The problem is complicated by the inability to relate software cost to progress towards completion of software development. Progress is often reported by the percentage of estimated man-hours expended to date; a less than reliable estimator of the amount of software actually completed, in most cases. Also, since there is no standard list of software products (end items) for which resource requirements have been collected in previous projects, the Program Offices have few precedents upon which to base estimates of resource requirements.

The variability in the recorded costs of previous developments is due not only to their size and complexity, but also to the following:

1. Fixed-price contracts have gone to ceiling. Hence, the costs reported understate the costs incurred.
2. Cost-reimbursement contracts have been augmented by supplementary agreements that incorporate changes in requirements.* This has resulted in redoing work already completed, so that costs are abnormally high per line of code delivered.
3. Level-of-effort maintenance contracts include costs for product improvement as well as error correction. Costs are difficult to assign to these two different functions.
4. Software costs are often hidden because of the difficulty--in some instances--of allocating costs between hardware and software.

*Generally initiated by Engineering Change Proposals (ECPs).

We concluded that the Program Offices were not a good source of data on the resource requirements for individual phases of the life cycle, at least within the resources available in the current contract. The quality of the data was poor because of the lack of uniform reporting, as described above. Also, there was not sufficient detail relating product completion to cost. Finally, what data existed was difficult to collect and did not include all phases of the life cycle, especially maintenance. At some future time the data from the Program Offices should be compiled into a usable data base, but the effort required will be well beyond the resources of this contract.

Hence, at the first TD meeting we proposed to take the following study approach. Data would be collected from readily available sources in which data on software product development could be measured. Of special concern was visibility into both development and maintenance portions of the life cycle. SAMSO contractors appeared to be a good source, although PARMIS (see Sec. 5) turned out to be the best source.

These data would be used to test hypotheses about resource consumption developed from the process model. These relationships would then be applied to the aggregated cost and man-hour data available in the literature (SDC[4,5,6] and ADPREP[12]) to derive interim estimating relationships.

On a parallel path, the cost reporting system would be defined. In the long term, it will be used to establish an improved cost data base and improved estimating relationships.

Since that meeting, we have defined the cost elements and WBS (Tasks 2 and 3); hypothesized resource estimating relationships (Task 5); and collected data from PARMIS, SDC, and IBM (Task 4). Changes to the Process Model have been made (Task 5) and data have been analyzed with

respect to the hypothesized estimating relationships (Task 5). Work has concentrated on man-hour estimation.

Results were at first disappointing. Applying the data to the hypothesized man-hour relationships for the individual phases of the life cycle resulted in less precision than estimates made on the total. That is, data plots of different life-cycle phases showed such "buck-shot" patterns that an estimate made by adding up estimates of the individual phases showed much greater variability than aggregated estimates based upon relationships for total man-hours.

It was at this point that the importance of the relations between phases became clear. Much of the variability in the total, which has been seen previously by various researchers, was due to differences in man-hour allocation among phases in the development process. There are trade-offs between the phases. For example, allocating too few man-hours to analysis and design can cause higher error rates, resulting in increased man-hours for integration and test. It could also result in large maintenance costs, which are not even seen in the development-cost data. Conversely, efficient levels of analysis and design would yield lower total man-hours or costs. Trying to estimate man-hours for each individual phase only accentuates the variance, and the trade-offs implicit in the totals are not considered at all. Statistically speaking, the total variance was bound to be larger since the covariance terms (some of which are negative) were not considered.

If this theory could in the future be developed and tested, not only could we better explain software costs, but we could identify rules for the efficient distribution of resources among phases. Using these rules we would have hope of reducing the risk of software development, a problem stressed by Clapp.[8]

1.3 ORGANIZATION OF THIS REPORT

This final report gives a detailed account of our work towards the development of phase-specific estimating relationships and the cost reporting system.

Sec. 2 defines a process model of the software life cycle. This model serves as the basis for all the later sections.

In Sec. 3, the man-hour estimating relationships hypothesized are given. Although they were not substantiated with any reasonable degree of success, they serve to identify the key parameters and equation forms.

In Sec. 4, we develop hypotheses for the trade-offs between phases. Work towards testing these relationships, using the PARMIS data base, is reported in Sec. 5. Although results are not statistically conclusive, the approach offers great potential for resource allocation, risk reduction, and eventually software cost estimation.

Sec. 6 contains a number of estimating relationship results using other, more aggregated, data bases. The appropriateness of using data on business-software developments, such as PARMIS, to estimate resource requirements for defense-system software developments is addressed by examining differences in resource consumption between the two types. We believe that the form of the estimating relationships should be applicable to both types, although the coefficients will change. Also included are results that demonstrate (1) the importance of several other explanatory variables to software cost, (2) relationships between alternative measures of the software product, and (3) characteristics of the maintenance activity of software.

In Sec. 7 some "rule of thumb" estimating relationships and methods of estimation are given. It was felt that a presentation of some of the

estimating techniques in the literature, updated by the experience of the present study, would be a useful supplement to the report. The reader should be cautioned that this was not a major effort of the study, and does not represent an exhaustive attempt to pick the best relationships out of the literature.

In Sec. 8, the elements of the future cost reporting system are defined. Also included are data requirements to measure product completion, a standardized list of software end items, man-hour and computer-hour reporting elements, and the relationships of these reporting elements to an existing automated report system.

Conclusions of the study and recommendations for future work are contained in Sec. 9.

2 PROCESS MODEL OF THE SOFTWARE LIFE CYCLE

2.1 INTRODUCTION

This software cost study differs from earlier investigations in two significant ways. First, it has examined the entire life cycle of software, rather than concentrating on the development phase. There is evidence[13] that the post-development activities of the software life cycle account for a significant fraction of all software costs and that the fraction is growing. Thus, a life-cycle cost model that considers more than development costs is necessary. Second, the current investigation was directed to organize its resource estimation techniques in terms of separate estimators for the component activities of the software life cycle. This should be contrasted with aggregated estimation techniques such as those reported in Refs. 14, 15, 16, and 17.

A disaggregated estimating technique for software resource consumption has important advantages. First, it provides insight into the development process, possibly providing direction for modifying the process. More importantly, it can serve as both a predictor and a control or monitoring mechanism. Aggregated techniques cannot be used to detect cost problems during a life cycle, unless a means exists for allocating the aggregate costs. Finally, disaggregated techniques may serve to reduce error in the estimates as information about the project improves. For example, explanatory factors for later activities may be based upon estimates of the outputs of the early activities. The estimate of remaining costs can then be improved at the completion of the early activities.

The investigation started with an oversimplified view of the estimators for the component man-hours (costs) of the software life cycle. An aggregated approach to making an estimate $\hat{Y}$ of man-hours Y can be represented by:

$$\hat{Y}_{aggregate} = f(\text{explanatory factors})$$

When estimating each activity separately, the representation is modified as follows:

$$\hat{Y}_{\text{disaggregate}} = \sum \hat{y}_j$$

where

$$\hat{y}_j = f(\text{explanatory factors for activity } j)$$

If such estimators are based on regression analyses, then the accuracy of an estimator can be quantified by the variance of the estimate. For either approach, a variance can be associated with each estimator: i.e., $\hat{Y}_{\text{aggregate}}$ will have a variance $\hat{V}_{agg}$, and $\hat{y}_j$ will have variance $\hat{v}_j$. If the total man-hours are estimated by adding up the component man-hours, the variance of the sum is the sum of the variances.

$$\hat{V}_{\text{disagg}} = \sum \hat{v}_j$$

assuming that there are no interactions or tradeoffs among the activities (i.e., they are independent).

Late in this study it became apparent that the variance of the sum of component estimators was generally larger than that of known aggregate estimators. This would result in a composite estimate of total man-hours that was poorer than an estimate made by aggregate techniques.

It thus became apparent that a model based upon independence of the activities is far too simple. In statistical terms, the assumption of independence between activities must be dropped and covariance terms must be added to the estimated variance. That is,

$$\hat{V}_{\text{disagg}} = \sum \hat{v}_j + \sum \hat{\text{covar}}_{ij}$$

Covariance, in this case, captures the interactions and trade-offs between the man-hours required for individual activities that are not apparent in a composite estimation technique.

Thus, a second requirement has been levied on the approach this investigation was taking. That is, any estimation technique based upon individual activities will have to consider the interactions of the activities if the variance of the life-cycle estimates is to be small enough to make the technique viable. Preliminary attempts have been made to change the model to incorporate these interactions, and some estimates of these interactions have been achieved. Sample size is too small, however, for results to be conclusive.

We have used the parametric approach to estimation throughout the study. The approach is to identify a number of physical, performance, or design characteristics which are directly related to costs (or resource consumption), for the system or equipment under study. Then, historical data from related, or similar, systems are used to calibrate and verify a postulated mathematical relationship between cost and these characteristics (or parameters); the resulting equation is a cost estimating relationship (CER).

The parametric approach uses principles of statistical inference; we particularly emphasize the principle that a specific selection of characteristics (independent variables) and a specific mathematical form should be hypothesized on the basis of a technical understanding of the process under study. Then, statistical procedures may be employed to <u>verify</u> the postulated CER.

In practice, however, statistical procedures are often used to <u>identify</u> the independent variables and to <u>specify</u> the functional form of the estimating equation. We believe this procedure is incorrect,

although it is difficult to be rigorous on this point. Parametric cost analysis is largely exploratory and there is invariably some trial-and-error in finding a defensible CER. Thus, a search (using statistical selection criteria) among a limited number of hypotheses is not necessarily to be faulted. An understanding of the underlying relationships between dependent and independent variables may be achieved only after statistical indicators have pointed the way. But there must be an eventual logic and rationale for the engineering and economic relationships which underlie the CER.

The materials presented in this and the next two sections develop the hypotheses for a software life-cycle cost estimating technique subject to the considerations previously mentioned.* This section introduces a process model of the software life cycle, upon which all subsequent arguments are based. Sec. 3 develops hypotheses about the factors that might be used to explain the man-hours required for each component of the software life cycle, assuming independence between the components. Sec. 4 proposes hypotheses regarding the interactions of the life-cycle components and how such interactions might be combined with the other explanatory factors to form a man-hour estimation technique. Validation of the hypotheses is the subject of Secs. 5 and 6.

2.2 OVERVIEW OF THE PROCESS MODEL

The initial version of the process model of software development was derived from interviews at Air Force Program Offices and from current Air Force regulations; in particular, AFR 800-14.[11] During the contract, the model was revised in several respects. This section is an overview of the original model, the revisions, and the reasons for the revisions. The revised model is then defined in detail in Secs. 2.3, 2.4, and 2.5.

---

*For the most part man-hours, the most significant resource, are evaluated instead of cost.

The development of software for use in a defense system can occur anywhere in the "major defense system acquisition life cycle" shown in Fig. 2.1. Interviews at Program Offices that are developing software for communications, command, control, and intelligence (C^3I), and for avionic systems, indicated that software development usually occurs during the validation, full-scale development, or deployment phases of the major defense system acquisition life cycle. However, software may be developed in any phase.

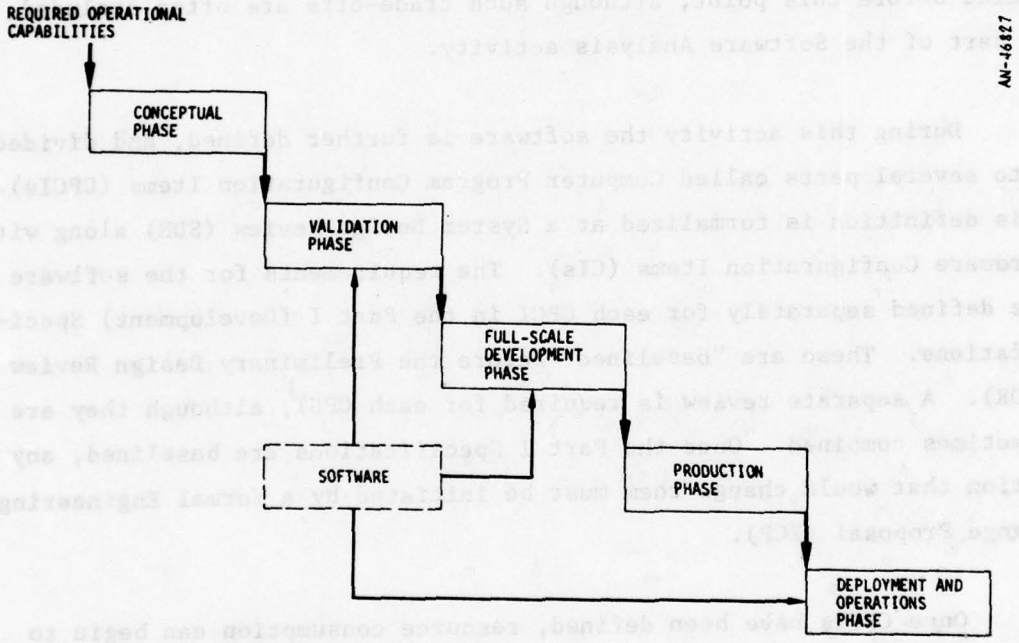


Figure 2.1. Major Defense System Acquisition Life Cycle

Our initial formulation of the Process Model for software development (Fig. 2.2) divides the software life cycle into three major phases: (1) Development, (2) Installation* (or Production), and (3) Operation and Support, each phase consisting of several activities. This flow-diagram representation of the process depicts the relationships between the activities and the information flows that define the software product and the changes to it.

The Development Phase begins with the definition of the software to be developed, generally contained in the System Specification. Trade-offs between hardware and software are assumed to have been completed before this point, although such trade-offs are often included as part of the Software Analysis activity.

During this activity the software is further defined, and divided into several parts called Computer Program Configuration Items (CPCIs). This definition is formalized at a System Design Review (SDR) along with hardware Configuration Items (CIs). The requirements for the software are defined separately for each CPCI in the Part I (Development) Specifications. These are "baselined" before the Preliminary Design Review (PDR). A separate review is required for each CPCI, although they are sometimes combined. Once the Part I Specifications are baselined, any action that would change them must be initiated by a formal Engineering Change Proposal (ECP).

Once CPCIs have been defined, resource consumption can begin to be separately recorded for each CPCI. Unfortunately, there are no standard definitions for CPCIs, so that comparisons among software developments are not easy. Furthermore, the CPCIs as defined often

---

*For software the Production phase is primarily a process of installation.

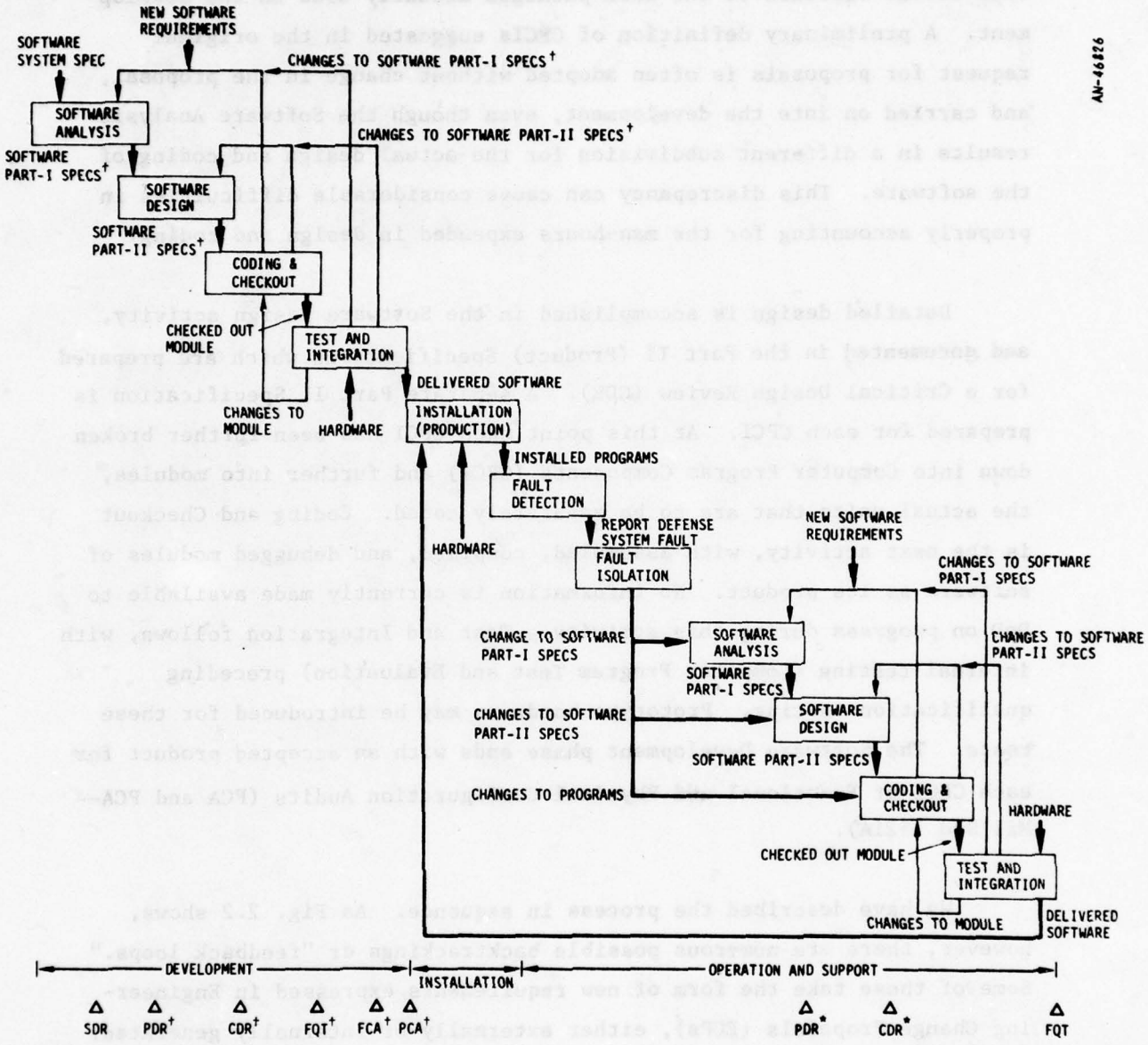


Figure 2.2. Initial Process Model

bear little relation to the work packages actually used in the development. A preliminary definition of CPCIs suggested in the original request for proposals is often adopted without change in the proposal, and carried on into the development, even though the Software Analysis results in a different subdivision for the actual design and coding of the software. This discrepancy can cause considerable difficulties in properly accounting for the man-hours expended in design and coding.

Detailed design is accomplished in the Software Design activity, and documented in the Part II (Product) Specifications which are prepared for a Critical Design Review (CDR). A separate Part II Specification is prepared for each CPCI. At this point each CPCI has been further broken down into Computer Program Components (CPCs) and further into modules, the actual units that are to be separately coded. Coding and Checkout is the next activity, with assembled, compiled, and debugged modules of software as its product. No information is currently made available to DoD on progress during this activity. Test and Integration follows, with internal testing (Computer Program Test and Evaluation) preceding qualification testing. Prototype hardware may be introduced for these tests. The Software Development phase ends with an accepted product for each CPCI at Functional and Physical Configuration Audits (FCA and PCA--Mil Std 1521A).

We have described the process in sequence. As Fig. 2.2 shows, however, there are numerous possible backtrackings or "feedback loops." Some of these take the form of new requirements expressed in Engineering Change Proposals (ECPs), either externally or internally generated. When these cause changes in the Part I Specifications, they require a repetition of analysis, design, coding, etc. Similarly, redesign can occur that changes the Part II Specifications, requiring recoding and retesting. However, there is no documentary record of this loop, since Part II Specifications are rarely baselined until after qualification

testing. Thus, even though there is considerable looping through the activities, only the loops that involve changes in Part I Specifications are recorded and visible outside the development group. Furthermore, there is no way to track the progress of development between CDR and the beginning of formal qualification testing.

Software is then installed (a phase that corresponds to Production in a hardware procurement), and enters the Operation and Support phase. During this phase of the life cycle, it is often impossible to separate software from hardware activities. For example, the Fault Detection activity shown in Fig. 2.2 may result from either a hardware malfunction or a bug in the software.

Fault Isolation may result in the detection of a software error. In that case, software maintenance responds to the error, possibly requiring new analysis, design, coding, etc. The development cycle is essentially repeated (although a change in specifications could hardly be termed "maintenance").

The process model has several features that distinguish it from earlier models of software development:

1. It addresses the entire life cycle of software, not only development.
2. It identifies parameters that could be used to measure the changes of state in the software during development.
3. It is oriented to USAF standards and regulations.

The initial formulation of the process model based on AFR 800-14[11] also has several shortcomings, which have led to the revised process model shown in Fig. 2.3. (We should emphasize that acceptance of these revisions will require modification of AFR 800-14.)

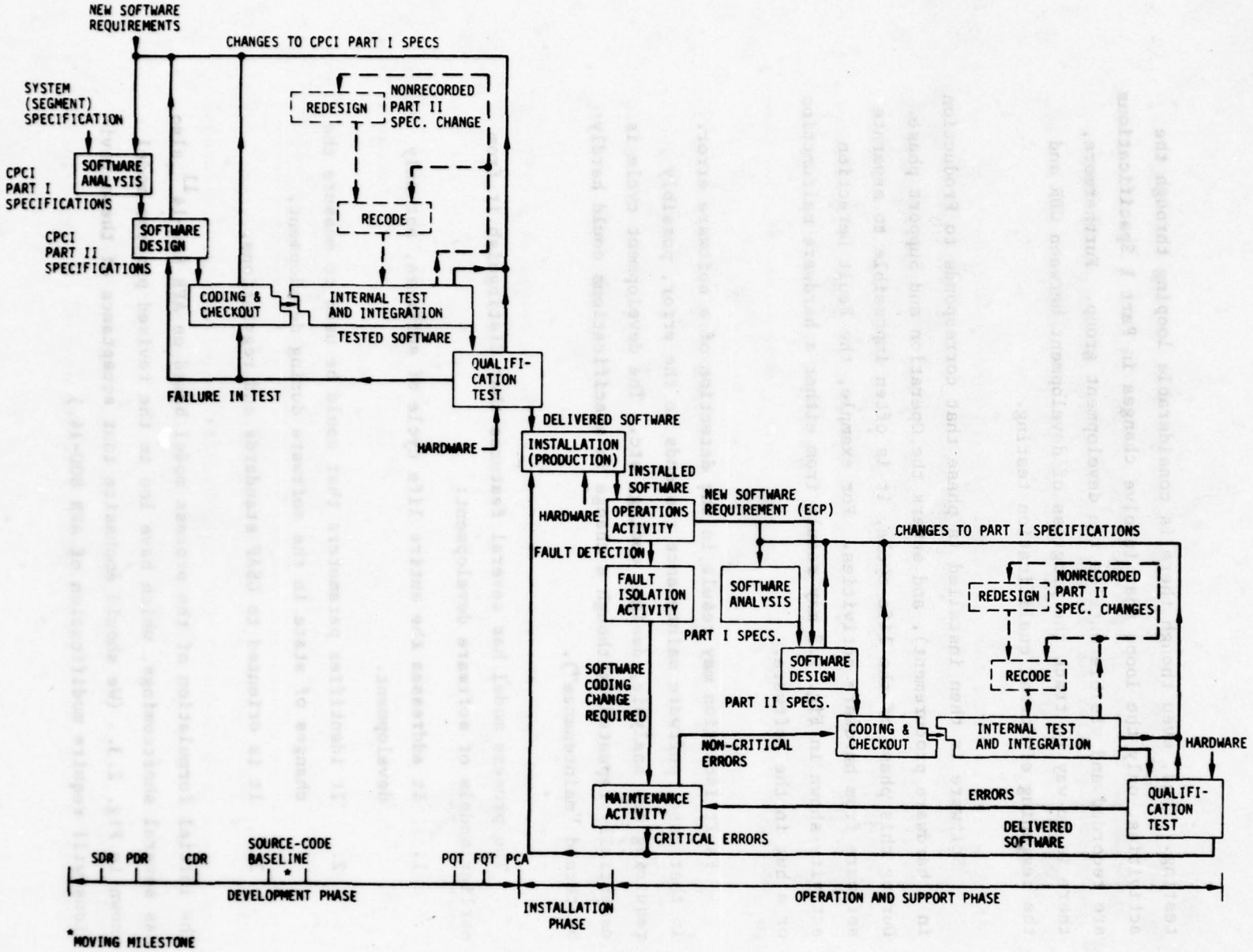


Figure 2.3. Revised Process Model

First, the fact that Part II Specifications are rarely baselined before qualification testing means that man-hours spent in design changes and resulting recoding before formal qualification testing cannot be separately measured. This is shown in Fig. 2.3 by the dashed lines and boxes connected to the Internal Test and Integration activity.*

In effect, these redesign and recoding activities are included in Internal Test and Integration. We believe that these activities are a primary cause of variability in software cost. It is easy to see how a poorly conceived Software Design activity could lead to a large number of errors in the Coding and Checkout activity and thus to a great deal of redesign. Since this consequence is not separately monitored and occurs late in the development phase, it can be the most important cause of cost and schedule problems. Reducing the amount of redesign and recoding is a primary way to reduce the risk in software development; but it can only be accomplished by understanding the relationships between design, coding errors, and redesign. That is, the trade-offs of resources between activities are of prime importance.

Desirable as it would be to account separately for redesign and recoding, we do not recommend an attempt to separate them in Internal Test and Integration. The activities are so intertwined that separation would be nearly impossible. Hence, the activities indicated by dashed lines in Fig. 2.3 are here treated as part of the Internal Test and Integration activity.

Note that there are two testing activities. Internal Test and Integration includes all the testing and integration that occur before formal qualification testing. Little or no visibility is available during this activity. During Qualification Test, on the other hand,

*Also known as Computer Program Test and Evaluation.

the contractor is carrying out formal tests, which are viewed by Air Force personnel. Tests include Preliminary Qualification Tests (PQT) and Final Qualification Tests (FQTs). Results are formally documented, and corrections of any errors that are discovered can be recorded.

A second shortcoming of the initial process model is that activities do not have clearly defined beginnings and ends. They may overlap in time, as Figs. 2.2 and 2.3 suggest, rather than being clearly separated by "milestones." Milestone definitions are very convenient, however. For example, Software Design can be conveniently defined as starting with PDR and ending with CDR. A further complication is that many software developments have reported man-hours by such milestone divisions, but others have reported them using "activity definitions" of terms such as analysis, design, and coding. This discrepancy has caused confusion as well as variability in the data. AFR 800-14 perpetuates this practice by using attributes of both definitions when it refers to "phases".

In this report we will try to avoid confusion in terms by adopting the following convention. When the terms analysis, design, and so on are used without qualifiers, an activity definition is intended. When the terms are followed by phase or milestone, a milestone definition is intended.

In general, we use "milestone" definitions of the activities because that appears to be the way the data has most often been reported. When analysis, design, etc. are defined in this manner, they form clearly separable phases, with no feedback other than ECPs (which we argue should have their own development cycle).

However, in several sections of this report we make use of activity definitions. In Secs. 4 and 5, the PARMIS data allows such a definition,

which is important to measuring overlap among activities, a key to the study of trade-offs between activities (as well as phases). In Sec. 8 (the reporting elements) both definitions are used. A transition matrix which relates one to the other is explicitly defined. This matrix (Table 8.14) will be useful in separating the two uses of these terms.

The milestone definitions have the following boundaries. Software Analysis (milestone) includes all activities up to PDR; the date of PDR can be different for each CPCI. Software Design (milestone) ends at CDR, for each CPCI. Coding and Checkout (milestone) ends with the baselining of the source deck, for each module. Internal Test and Integration (milestone) ends with the beginning of the first qualification test. Qualification testing (milestone) ends with the acceptance of each CPCI at PCA. Installation (milestone) is then initiated, and ends with Initial Operational Test and Evaluation (IOT&E) at the operating sites. Operation and Support then commences.

Between Coding and Checkout (milestone) and Internal Test and Integration (milestone), we have defined a "moving milestone," indicated by the jagged separation in Fig. 2.3. That is, the end of Coding and Checkout is reported separately for each module (or CPC), as a means of tracking the progress of software development. No milestone or means of tracking now exists between these activities, so the addition will greatly assist Program Offices in the control of software development. This topic will be explained further in Sec. 8.

The flow of maintenance activities during the Operation and Support phase has also been refined significantly in Fig. 2.3. The effects of new requirements have been separated from those of fault detection, so that true "maintenance" activities can be identified. There remain, however, two sources of confusion between maintenance activities and new requirements. For one thing, an ECP that specifies

new requirements may also incorporate a number of corrections of non-critical errors. For another, the correction of errors detected during Qualification Testing of an ECP may be treated as maintenance activity and thus not be associated with the ECP that caused the errors to be introduced.

The foregoing has been an overview of the reasons for revising the initial process model into the form shown in Fig. 2.3. This revised model is the framework for the hypotheses concerning the factors that explain software costs (Sec. 3) and for the Work Breakdown Structure presented in Sec. 8. The following three subsections refer to Fig. 2.3 and give definitions for the terms used throughout the rest of this report.

2.3 DEVELOPMENT PHASE

The development phase of software begins with a general system specification, or system segment specification.[18] This system may be totally new or a modification of an existing system. In any case, the system specification is available before the start of software development.

The functional and performance requirements of the specification are then partitioned among one or more Computer Program Configuration Items (CPCIs). Each CPCI undergoes Software Analysis (milestone), which formulates a Part I Specification.[18] The Software Analysis (milestone) terminates, for each CPCI, at PDR after the Part I Specification for that CPCI is baselined. This baselining establishes the Allocated Configuration Identification and subjects the CPCI to formal configuration controls (MIL Std 480 and 483). This baselining should occur before PDR (Preliminary Design Review).

At PDR the Software Design (milestone) begins. During this period a complete draft of Part II Specifications[18] and a more detailed design

of the CPCI are generated. Software Design (milestone) for each CPCI terminates with a Critical Design Review (CDR) of its Part II Specification. The design reviews of all related CPCIs are generally held simultaneously. The Part II Specifications are usually subjected only to informal configuration controls between CDR and Physical Configuration Audit (PCA), despite the fact that all CPCI coding, debugging, testing, and integration occur between these two milestones.

At CDR, Coding and Checkout (milestone) begins. It terminates when the code for each module (or CPC) has been written, assembled or compiled, and separately tested (unit-tested) to the programmer's satisfaction. There is no single milestone for the completion of this activity; the "moving milestone" marks its completion for each module of software. The product of this phase is the version of the code to be used in testing and integration of the CPCI.

Functions during Internal Test and Integration (milestone) are not exclusively associated with individual CPCIs. The purpose is to informally test each CPCI, collections of CPCIs, and the software as a whole. During this period, special hardware may be introduced for some tests. The total cost of Internal Test and Integration must thus be viewed from the system level and is not always attributable to individual CPCIs or even exclusively to software.

Each CPCI has a set of test plans and procedures, usually relating back to the Part I Specification. Preliminary and Formal Qualification Tests assure that each CPCI can pass this specific set of test procedures. These are performed during Qualification Test (milestone). Like Internal Test and Integration, the costs during this period cannot be easily allocated to individual CPCIs.

After the Part I Specification and/or the Part II Specification are baselined, any subsequent change to the specifications is tracked

by Engineering Change Proposals (ECPs) and Specification Change Notices (SCNs).[18] Fig. 2.3 indicates flows from each milestone period back to the prior period. These feedbacks represent supplemental agreements[43] to modify baselined Part I or Part II Specifications and software before delivery of the first version of the system. Each modification of a specification requires repeating a number of the previous activities. Such changes can be internally generated or can arise for external reasons, in the form of new requirements.

Figure 2.3 also shows paths from Internal Test and Integration (milestone) to redesign and recoding. Conceptually, it is possible to exercise formal configuration control over the checked-out software during test and integration by using a development library. If a problem detected during testing required redesign, recoding, and retesting, these activities could then be attributed to design, coding, and testing activities respectively. Practically, however, it is not possible to require a contractor to directly measure (separate) and report these resources against these activities.

2.4 INSTALLATION PHASE (PRODUCTION)

The Installation (Production) phase of software includes the generation of multiple copies of the software and their installation in computer systems. For C^3I systems, installation may require site-dependent modifications of standard software packages. For avionics systems, installation usually means only distributing identical copies of the software. If multiple copies of the software are not required, then there is no installation (production) phase in the software life cycle, since the test site will presumably be the only operational site.

2.5 OPERATION AND SUPPORT PHASE

Operation and Support (maintenance) of software must be examined from two different points of view: error correction, and modification.

Error correction consists of repairing reported difficulties with the operation of the software. The errors that are corrected are generally those which prevent the software from accomplishing its allocated functions. By definition, error correction does not require changes to the Part I or Part II Specifications. Maintenance may be concurrent with operations. Errors may also be corrected simultaneously on multiple versions of software.

Modification of an existing model or version of software produces a new version of the software. It begins with revising the Part I or Part II Specifications, either to correct deficiencies in the software's performance or to expand its capabilities. Modifications generally also include correction of errors that were temporarily repaired, or deferred, as part of the error correction activities.

2.5.1 Error Correction

Error correction is embodied in three concurrent activities (Fig. 2.3): operations, fault isolation, and maintenance. Operations is that activity which exercises the software to accomplish the intended mission. In some applications, this activity may require computer operators, etc., the costs of which can be viewed as operation costs of the software. In other applications (such as avionics) the costs of operation have little to do with the software.

When the Operations activity encounters problems with the system, it reports them as faults. The Fault Isolation activity then screens the reported faults and traces their cause to the hardware or software. The costs of this activity, like those of Operations, cannot be entirely attributed to software since the faults may lie with hardware. Those faults that are traced to software are the responsibility of the Maintenance activity.

The Maintenance activity may alter either the object code or the source code or both. The definition of the CPCI requirements and functions, as expressed by the Part I and Part II Specifications, is not altered. This activity (unlike Operations and Fault Isolation) can be charged completely to software.

2.5.2 Modification

Modification of software is an activity that also occurs during operation and maintenance. Experience with the operation of a system will identify deficiencies in the original requirements and functions of the system. In addition, enhancements will be required because of changes in the mission of the system. These changes in requirements will potentially affect the software components of the system.

Deficiencies in the design of the software may be identified by the Fault Isolation activity. Such deficiencies are those that require changes in the specifications of the software, rather than corrections to make it accurately reflect the existing specifications. The design changes will normally be accumulated and merged with changes that reflect desired extensions of the software's functions.

A new model or version of the software may be commissioned as part of the Operations and Support phase. Developing the new model begins with an SCN directing changes to the software. Nominally, these changes will be incorporated through a series of activities identical to those of the Development phase. A major difference is that the new model of the software is derived by modifying existing bodies of code and specifications rather than developing the software from scratch.* Another difference between the modification and

*An example of the important distinction between these two activities is the Defense Satellite Program software. The multiple versions of the CONUS Ground Station software are modifications, while the Simplified Processing System (SPS) software constitutes a new development.

development activities is that modification incorporates some coding changes to repair outstanding problems with the software which were not deemed critical and therefore not formally incorporated during error correction.

3 EXPLANATORY FACTORS FOR COSTS OF INDIVIDUAL PHASES

This section presents the hypotheses we initially developed about explanatory factors for the costs of software Development, Installation, and Operation and Support. A major choice at the outset was to develop hypotheses for several generic types of software that are of interest to ESD, and to ignore the fine structure of the software. That is, we did not consider relationships that depended on, say, the number of input/output statements or the hierarchy of subroutine calls. As will be seen, a major explanatory variable we did choose is the overall size of the software (number of object-code statements, in general). Given the generic type of software, and the hardware on which it is to be used, this instruction count is estimated early in project development. It is a readily available descriptor of previously developed software, although it generally does not include non-deliverable code, such as support tools.*

This section discusses explanatory hypotheses only for man-hours. The other important resources--computer-hours and elapsed time--were not considered because of the shift of attention to the overriding issue of the relations among activities.

The framework for this discussion is the Process Model described in Sec. 2. The phases are assumed to be defined by milestones (PDR, CDR, etc.) rather than by activity divisions (e.g., design, coding, and test), since this definition makes it easier to collect data from previous software development. (If the reporting system described in Sec. 8 is adopted, future developments will record cost data by cost elements that correspond more naturally and accurately to the activities.)

---

*Non-deliverable code should be a part of the count, as resources are required for its development. However, it most surely is not included in historical data and is implicitly included in estimating relationships only because it causes an apparently smaller number of lines of code per man-month.

As we have explained earlier, our attempts to quantify the following hypotheses were not successful because of the significance of tradeoffs among phases. However, since the definition of hypotheses for the cost-determining factors of the separate phases in software development has not been attempted before, we consider it important to publish these hypotheses so that they can be subjected to critical review by the software community.

Our attempts to demonstrate the significance of some of these relations are reported in Sec. 6. In particular, the maintenance relationships are evaluated, and the effect of the choice of programming language is explored.

3.1 DEVELOPMENT PHASE

Recall that the Development Phase is divided into: (1) Analysis, (2) Design, (3) Coding and Checkout, (4) Internal Test and Integration, and (5) Qualification Testing.*

3.1.1 Analysis Phase

Analysis (milestone) is defined as the work, completed no later than PDR, that generates baselined Part I Specifications for each CPCI. It is assumed that the staffing of this phase is homogeneous,** so that the resources can be estimated in man-hours and then converted to dollars by a single cost-per-man-hour number. The factors that are selected to explain the labor resource are chosen because they are readily measured (for past projects) and estimated (for future projects); and because they are end products of the Analysis Phase and its successor phases.

---

*Milestone definitions.

**An inhomogeneous labor mix, as shown by Wolverton,[19] is significant in determining the overall cost of development. For individual software phases during Development, we believe that the inhomogeneity is insignificant in determining the cost per man-hour since software phases correspond closely to job skill. We therefore assert an average cost per man-hour for each phase (not, however, the same for all phases).

The first hypothesis is that the labor required for Analysis stratifies as a function of the type of software being developed. This hypothesis was selected rather than an estimation technique based on program structure. Five types of software of interest to ESD are considered:* (1) operational flight programs; (2) tactical mission control programs; (3) command, control, communications, and intelligence (C^3I) programs; (4) simulator/trainer programs; and (5) automatic test equipment software. Each of these five types of software represents a problem with unique development and operational characteristics. For example, operational flight programs are implemented on small embedded computer systems and are generally concerned with functions related to guidance and weapon systems. C^3I software is implemented on large ground-based computers. Tactical mission control programs, like operational flight programs, are implemented on small embedded computers, but are distinguished by their functions and the more frequent need for maintenance as missions change. It is hoped that this simple stratification will encompass and quantify many of the effects that are generally attributed to "problem complexity" and "function". Stratification will be represented by developing an estimation technique with a common mathematical form, but with different constants for each type of software.

The second hypothesis is that analysis labor is proportional to some power of the "size" of the problem being analyzed. Denoting the size by X_1**:

---

* This classification of ESD software projects was suggested by Mr. Dan Fitzgerald at Wright-Patterson AFB.

** The value of X_1 is assumed to be summed over all CPCIs. It is the total magnitude of the system that is assumed to contribute to the complexities of subdividing it into CPCIs and correctly defining their interfaces, interactions, and specifications.

$$\text{Man-Hours, Analysis} = a_1 X_1^{a_2}$$

where the constants a_1 and a_2 are separately determined for each of the five types of software.*

The size of the problem, X_1, is surely measurable in some sense by the size of the software that is to be developed; and the size of the software is most straightforwardly measured by the number of instructions, either in the source language or in the machine language (object language). Studies of total software-development cost have suggested that using the number of source-language statements as a measure produces estimates with smaller variance. (All total-cost estimators, however, have very large variances.) When estimating the costs of the software phases separately, it is possible to use source-language statements as a measure for some activities and object-language statements for others, as we do here.

For the Analysis phase, what is wanted is a consistent measure of the size of the problem to be solved, without reference to the programming (source) language that will be used to solve it. For this reason, we adopt object-language statements as the measure of size for this activity. Because different computers require different numbers of object-language statements to express the same operations, the number of object-language statements should be adjusted for the type of machine to be used (either by stratifying the data or by defining a multiplier that depends on machine type). Thus, X_1 is defined as the number of object-language statements in the delivered software, adjusted for machine type.**

---

*The power curve has been selected to accommodate apparent diseconomies of scale in the software development process. This nonlinear behavior was indicated in early SDC studies.

**If all programs were coded in the same source language, say JOVIAL, then source-language statements would be a better measure, since that would automatically compensate for different machine types.

The final hypothesis is that the relationship of analysis labor to product should depend on whether the Analysis phase is adapting an existing software specification to a new application or working from an entirely new specification. Thus, separate estimators of the same form should be developed for each of these two categories of projects; that is, the constants a_1 and a_2 also depend on this factor.

An alternative measure of the product might be the size of the Part I Specifications. As an independent variable, it could be substituted for the variable X_1 in the above equation (with corresponding adjustments to a_1 and a_2). One difficulty with using this measure is that a Program Office has some control over the format and content of Part I Specifications. This would invalidate the general use of the size of Part I Specifications in the comparison of projects. It is of some interest, however, to test the hypothesis that the size of the final product (measured by an instruction count) is related to the size of the Part I Specifications, and therefore interchangeable as the independent variable. This hypothesis could be tested by collecting data on a single project with multiple CPCIs, all subject tc the same specification formats. The demonstrated existence of a relationship would be the basis for further investigation of alternative measures of product magnitude.

3.1.2 Design Phase

The Design phase is defined as the work, beginning at PDR and ending with CDR, that generates initial Part II Specifications. It is again assumed that staffing can be viewed as homogeneous, but perhaps with a different cost per man-hour than for the Analysis phase. The motivation for choosing the explanatory factors is the same as before.

As with Analysis, the first hypothesis is that the labor requirements stratify as a function of the type of software being developed. Stratification is again represented by developing an estimation technique with a common mathematical form, but with different constants for each type of software. The same types of software listed before are assumed.

The second hypothesis is, again, that labor man-hours are proportional to some power of the size of the software. In the Design phase, as well as in Analysis, the programming language to be used should not affect the estimate; hence, we again adopt object-language statements as the measure of size. However, the Analysis phase often identifies some portions of the problem as suitable for implementation by existing software. These portions need not be considered in the Design phase. Hence, we define a size measure, X_2^*, as the number of object-language statements in the delivered software, adjusted as before for machine type, minus the number of statements copied from existing software. The relationship for the Design phase is then:

$$\text{Man-Hours, Design} = b_1 X_2^{b_2}$$

where b_1 and b_2 are constants separately determined for each of the types of software.

The third hypothesis is that the relationship of design labor to product depends on whether the system to be developed is an adaptation of an existing system or entirely new. Separate estimators of the same form should be developed for these two categories of projects; i.e., the constants b_1 and b_2 depend on this factor.

It is again of secondary interest to test for a relationship between the final size of the product of the software development (code) and that of the immediate product of the Design phase (a Part II Specification). It is hypothesized that the size of a Part II Specification varies with the form of specifications selected by the Program Office and with the functions of the software being specified. Using data from a project with multiple CPCIs, all with the same required form of specifications, it is

---

* In this equation X_2 is assumed to represent the individual instruction count for a CPCI. The total costs of Design are, of course, summed over all CPCIs.

possible to remove the effect of the first variable. A regression analysis can then be used to test if a relationship between the size of the Part II Specification and the original estimate of program size must also include a dependence on the functions implemented by the CPCI. It would also be of interest to test for a relationship between the sizes of a Part I and a Part II Specification in a similar manner.

The existence of such relationships, perhaps dependent on the form of the specifications, would be useful in monitoring the status of projects. Original estimates of program sizes could be compared with the size of the Specifications to signal items that may require re-estimation and management review.

3.1.3 Coding and Checkout Phase

The Coding and Checkout phase is defined as the work that begins after CDR and ends with the start of Internal Test and Integration.* It consists of translating Part II Specifications to source code that can

---

*It is recognized that no single point in time corresponds to the completion of this phase. Conventional "bottom-up" developments will generally code and unit-test all of the modules of a system, deferring any attempts to test the modules as an integrated unit. When all modules have been coded and tested, the system enters an internal integration and testing phase prior to qualification testing. Mills has pointed out that it is usually during this period of activities that difficulties arise. Structured programming and "top-down" developments try to avoid deferring integration testing until all modules have been coded and tested as individual units, by encouraging the integration and testing of the software, as a system, as soon as each new module is developed. This philosophy should manifest itself as a sequence of points at which modules transition from Coding and Checkout to Internal Test and Integration.

be translated and interpreted by the computer system, and unit-testing that source code.* It is again assumed for this activity that staffing is homogeneous so that the costs of labor can be averaged, perhaps with a different cost per man-hour than for the Analysis or Design activity. Again, the motivation for choosing the explanatory factors is the same as in the previous hypotheses.

The first hypothesis is that Coding and Checkout labor is proportional to some power of the size of the actual software coded and checked out. In this case the size, X_3**, is measured by the delivered source-language statements, excluding any code that is taken directly from another system. The form of the estimator is

$$\text{Man-Hours, Coding and Checkout} = c_1 X_3^{c_2}$$

The second hypothesis is that the relationship of labor to product should depend on the choice of programming language.† A variable X_4 is introduced whose value is zero if programming is done in assembly language and 1 if in a higher-order language. The form of the equation becomes

$$\text{Man-Hours, Coding and Checkout} = c_3^{X_4} c_1 X_3^{c_2}$$

---

*Unit testing consists of those tests that a programmer might apply to a single unit of code to convince himself (or herself) that the unit of code functions as specified. In particular, the programmer is not attempting to test the interfaces of the module with other modules of the system. This is contrasted to the testing described in Secs. 3.1.4 and 3.1.5 in which the programmer implicitly (or explicitly) has declared a unit of code to be functioning correctly and is testing the operation of that and other units of code when integrated to form the CPCI. This level of testing might include exhaustive logic and data range tests at the module level.

**X_3 is assumed to be measured with respect to a single unit of code. The total cost of coding and checkout activities is developed by summing costs over all of the units of code to be developed.

†This effect has been documented by a number of published results.[20]

If a system contains software in more than one language, estimates for the separate parts must be summed to estimate the total.

The final hypothesis is that the relationship of labor to product should also depend on whether the software system is severely limited for hardware resources. Coding takes considerably more effort if the limitations of the machine must be taken into account. This effect was first discussed in Ref. 21 and further quantified in Ref. 22 as a discrete variable. In Ref. 22, a system is considered to be so limited if more than 95% of the available memory is used. The same definition is proposed here. A variable X_5 is introduced whose value is zero if the hardware constraint is not present and 1 if it is:

$$\text{Man-Hours, Coding and Checkout} = c_4^{X_5} c_3^{X_4} c_1 X_3^{c_2}$$

Note that, for this phase, the estimator does not depend upon the type of software, since Analysis and Design activities have already been accomplished for the most part.

Testing the hypotheses about the Coding and Checkout phase requires that the boundaries of the activity be well defined. Currently, Air Force regulations do not identify a milestone that can be used as the termination boundary of this activity. Thus, the boundary between the Coding and Checkout phase and the Internal Test and Integration phase is ambiguous. We assume that data can be collected to describe projects that used the same definition of the boundary, and that none of those projects included major revisions in requirements or design. If so, the validity of the hypotheses can be tested by regression analyses. More careful data collection (as proposed in Sec. 8) will ultimately avoid this problem, but for now this is the best that can be done.

3.1.4 Internal Test and Integration Phase

The Internal Test and Integration phase begins with the completion of Coding and Checkout (for individual modules) and terminates with the start of Qualification Testing. It is often difficult to ascribe the cost of this phase entirely to software; in avionics systems, for example, Internal Test and Integration commonly addresses the functions of both hardware and software concurrently. Thus, one should view the proposed estimation technique as predicting the cost of only the software portion of the Internal Test and Integration activity.

The software testing activities might include several items: the integration of modules to form CPCIs, and the informal rehearsal of qualification tests that are to be formally applied to the software. These tests, when formally applied, would be considered part of the Qualification Testing phase discussed in the next section.

In this phase, we believe that it is not adequate to assume that the labor mix is homogeneous. The reason is the "hidden" redesign and recoding, discussed in Sec. 2, that often make up a large part of this phase. We have not proposed a hypothesis to account for this effect; it is intimately related to the problem of relationships among phases to be discussed in Sec. 4. If it were to be treated, an explanatory variable might be the duration of this phase. The longer the time spent in Internal Test and Integration, the more likely it is that higher-paid analysts and designers must be averaged into the labor mix.

The first hypothesis is that Internal Test and Integration labor requirements also stratify as a function of the type of software being developed. The stratification will again be represented by developing an estimation technique with a common mathematical form, but with different constants for each type of software. The same software types used for the Analysis phase will be assumed.

The second hypothesis is that the labor requirements vary as some power of the size of the component that must be subjected to testing. Each CPCI separately must pass a number of test procedures to qualify it for acceptance. The size is measured by the number of delivered object-language statements, X_1. The measurement is for a single CPCI. The form of the estimator is:

$$\text{Man-Hours, Internal Test and Integration (each CPCI)} = d_1 X_1^{d_2}$$

The third hypothesis is that the relationship between labor and product also depends on the programming language and on the constraints that may be imposed by limited computer resources (as discussed under Coding and Checkout). Each of these factors can modify the effort required to identify and reapir problems encountered during testing.

$$\text{Man-Hours, Internal Test and Integration (each CPCI)} = d_3^{X_4} d_4^{X_5} d_1 X_1^{d_2}$$

where X_4 and X_5 are the dummy variables defined under "Coding and Check-out."

The fourth hypothesis is that "top-down" development techniques should have an impact on the costs of Internal Test and Integration. This is modeled by introducing a new dummy variable, X_6, to capture the impact of structured programming and "top-down" development on software development costs. The variable X_6 takes the value 1 if "top-down" methods are employed and 0 if otherwise. The form of the equation becomes

$$\text{Man-Hours, Internal Test and Integration (each CPCI)} = d_5^{X_6} d_3^{X_4} d_4^{X_5} d_1 X_1^{d_2}$$

Since this estimator applies to the testing efforts for a single CPCI, the total labor must be summed over all the CPCIs in the product.

3.1.5 Qualification Testing Phase

The Qualification Testing activity begins with the completion of Internal Test and Integration and terminates with the acceptance of the CPCI at PCA. As with Internal Test and Integration, it is difficult to ascribe the costs of acceptance testing to software as distinct from hardware. Again, one should view the following estimation technique as applying only to the software component of Qualification Testing.

It will be assumed that the staffing of this activity is homogeneous; the average cost per man-hour may be different than those of the preceding phases. Unlike Internal Test and Integration, redesign and recoding should be minimal. If not, they are certainly visible to Air Force inspectors. The motivation for choosing explanatory factors is the same as that used in formulating the previous hypotheses for analysis and desgin.

Qualification Testing is generally much less expensive than Internal Test and Integration. Its component costs can be attributed to the testing of individual Computer Program Components and to the testing of the CPCI as a whole. Further, it is conjectured that the testing effort required is related to the size of the delivered product, measured in object-language instructions (adjusted as before for differences in machine type), and to the number of CPCIs whose interaction needs to be tested. This phase, like most of the others, should be stratified by the type of software being developed. The general form of the estimation technique is

Man-Hours, Qualification Testing

$$= e_1 X_1^{e_2} + e_3 (X_7 - 1)^{e_4}$$

where

X_7 = number of CPCIs needed in testing (including CPCI being tested).

3.1.6 Changes in Requirements and Design

Few software developments are free of changes in requirements and design, whether done at no (visible) cost or by renegotiation of the contract. Fig. 2.3 illustrates these with solid lines. It is conjectured that the change activity is significant in explaining the observed variations in software cost.

One approach to testing the conjecture would be to treat design changes as another activity to be costed separately. Any data used in testing the previously developed hypotheses would have to be adjusted for any negotiated changes in cost, and be based on the best estimate of the size the software would have been without the changes. The approach would require developing a labor estimation technique for changes in requirements and design. Such a technique would have to reflect the state of development at the time the change is introduced, and be responsive to the magnitude of the change.

Because changes may reduce, extend, or modify the scope of a development, a cost estimation technique would be difficult to develop. Changes that reduce scope eliminate costs that would be incurred by some fraction of the project, if development were to continue without the change. Extensions of the scope introduce new costs that would not have otherwise occurred, and new costs due to interfacing. Modifications of scope produce some combination of both effects.

In many instances, changes in requirements and design are performed "at no cost." It is believed that such changes do, in fact, have an impact and can be used to explain variations in observed costs.

We have attempted to develop cost hypotheses for changes regarded as another development activity. In Sec. 6, we do examine a hypothesis for aggregated development cost, with changes represented by a parameter:

$$\text{Man-Months, Development} = k_0 + k_1 Z_1 (1 + k_2 Z_2)$$

where Z_1 is a size measure (number of statements) and Z_2 is the number of changes (ECPs).

3.2 INSTALLATION (PRODUCTION) PHASE

Recall that the production phase of a software system consists of its installation on one or more computers. It is assumed that the staffing of this phase of activities is homogeneous; again, the cost of a unit of labor may be different than for other activities. The same motivations apply for the choice of independent parameters.

The first hypothesis is that the effort of installation depends linearly on the number of computers that are to receive the software, X_7. The form of the cost estimating relationship is:

$$\text{Man-Months, Installation} = f_0 + f_1 X_7$$

The second hypothesis is that the proportional relationship of cost to the number of computers is modified if software adaptations to different computers are required. Let X_8 take the value 0 or 1, depending on whether any computer requires specific modifications to be made in the software. The form of the cost estimating relationship is modified to

$$\text{Man-Hours, Installation} = f_0 + f_2^{X_8} f_1 X_7$$

The final hypothesis regarding installation cost is that the equation should be stratified by the type of software being installed; that is, the f's depend on software type.*

*Clearly, installing C^3I software with site-dependent modifications is different from installing avionics software in a wing of aircraft.

3.3 OPERATIONS AND SUPPORT PHASE

Recall that the Operations and Support phase of software can be viewed as two separate activities: error correction and modification. These are the only two activities that can be completely charged to the software life cycle. The steps of modification are nominally identical to those of development. It is hypothesized that the form of the cost estimating relationship for modification should be the same as for development. However, one should not expect the coefficients to be identical. The number of versions (modifications) required is hypothesized to be proportional to the operational life of the system, where the constant of proportionality depends on type of software.

For error correction, it is assumed that the staffing is homogeneous; the cost per man-hour may differ from that of other phases. The motivations for choosing independent parameters are the same.

The first hypothesis is that the cost of error correction is linearly dependent on the number of reported errors per unit time, $X_9(t)$. The form of the cost estimating equation is

$$\text{Man-Hours, Error Correction (per unit time)} = g_1 + g_2 X_9(t)$$

The expected number of errors to be corrected per unit time is clearly dependent on the system lifetime. It is hypothesized that the distribution of errors over the system lifetime is of the form specified in Fig. 3.1.

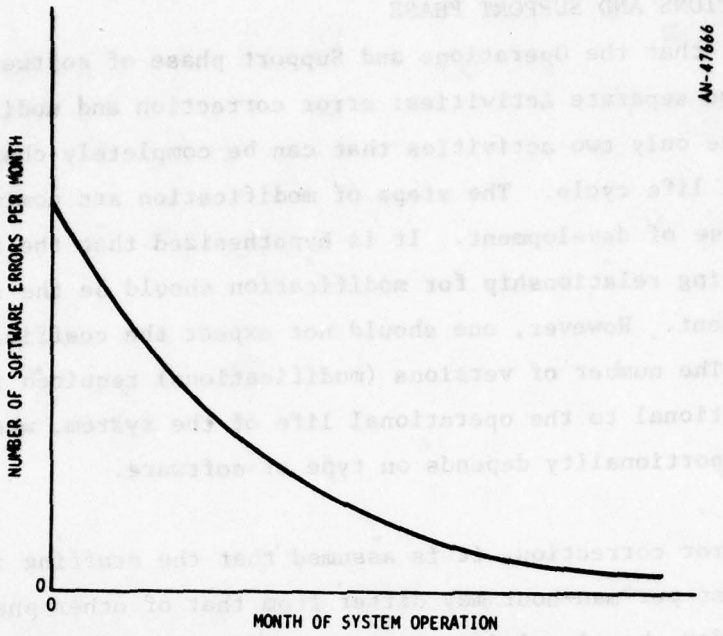


Figure 3.1. Number of Reported Errors Per Month

The second hypothesis is that the constant of proportionality in the above equation is dependent upon the size of the software package being maintained.* The form of the cost estimating equation is modified to:

$$\text{Man-Hours, Error Correction (per unit time)} = g_1 + g_3 X_1^{g_4} X_9(t)$$

where g_2 of the previous equation is given by

$$g_2 = g_3 X_1^{g_4}$$

and X_1 is the size of the software measured in object-language instructions, as before.

---

*The larger the software system, the more difficult it is to isolate the cause of the software problem.

The third hypothesis is that software maintenance costs are determined by the method of software development. In particular, the constant of proportionality is further modified by the use of a high-order programming language and by the presence of a hardware constraint. The cost estimating relationship is further modified to

$$\text{Man-Hours, Error Correction (per unit time)} = g_1 + g_5^{X_4} g_6^{X_5} g_3 X_1^{g_4} X_9(t)$$

where X_4 and X_5 are the dummy variables defined earlier to account for programming language and hardware constraint.

The final hypothesis is that software maintenance costs are determined, in part, by the type of software being maintained. Thus, the coefficients of the above equation must be derived for each type of software considered.

4 HYPOTHESIZED RELATIONSHIPS BETWEEN ACTIVITIES

The construction of estimating relationships by considering each activity independently implies that the relationships for one activity are not affected by parameters associated with a different activity. For example, we assumed in Sec. 3 that the man-months required for Coding and Checkout (milestone) could be predicted by software and computer resource parameters, and that they would not be significantly affected by the man-months spent in Design (milestone) or in Testing (milestone).

This section considers the construction of estimating relationships on the hypothesis that the separate phases or activities are tied together in predictable ways, and that it is not possible to develop estimating relationships without considering these interactions. (In fact, we can painfully attest to the inability to make sense out of man-hour data for individual phases or activities without considering these trade-offs).

This approach holds that, besides the relationships between the cost-driving parameters and the software-related characteristics, there are trade-offs between resources expended in one phase or activity and resources expended in another. For example, increasing the resources spent in design would tend to decrease the resources required for testing or maintenance. The objective of this section is to develop this type of relationship.

4.1 A SOFTWARE LIFE-CYCLE COST MODEL WITH RELATIONSHIPS BETWEEN ACTIVITIES OR PHASES

The justification for using a model with interrelated phases or activities for analyzing software cost derives from two propositions:

- Analyses using the simpler independent-phases approach have not succeeded. Man-hour variations within activities are large. This led us to try a more complex model.

- **Analysis of detailed data from software development projects provides substantial evidence of why and how the phases or activities affect one another.**

In this section, we examine some data on the activities in an actual data processing development. (The data were sufficient to use the "activity definitions" for Analysis, Design, etc.) We shall show that, in addition to the basic problem of translating a set of requirements into reliable, correct computer programs, there are influences caused by the need to conform to a development plan. The plan is an essential management tool for ensuring that needed resources are available to the project at the proper time and in the correct amounts. One would like to see how changes in the plan, caused either by changes in requirements or by failure to meet commitments, affect cost-driving parameters. Particularly, one would like to see how management actions influence the measurable project descriptors.* Understanding this relationship should permit a more accurate analysis of the cost-driving variables.

Figure 4.1 shows the time spans and levels of effort for the different phases of a software development project. (Data are derived from the PANAMIS data base described in Sec. 5.) Planned values are shown by solid lines and actual values by dashed lines. The example is a business program written in COBOL.

To begin with, there is considerable scheduled overlapping of the design and coding activities. A milestone definition of these activities, such as that used in Sec. 3, misses these overlaps. This overlapping is a common practice, but it increases the likelihood that changes in the design will require parts of the system to be recoded. Such a schedule might have been adopted because time was short or because certain people were only available at certain times. In either case, overlapping causes any problems or delays to have increased impact on the work.

---

*Project descriptors include man-hours for analysis, coding, testing, etc. (planned and actual), time span for the activities, numbers of personnel, application classification, etc.

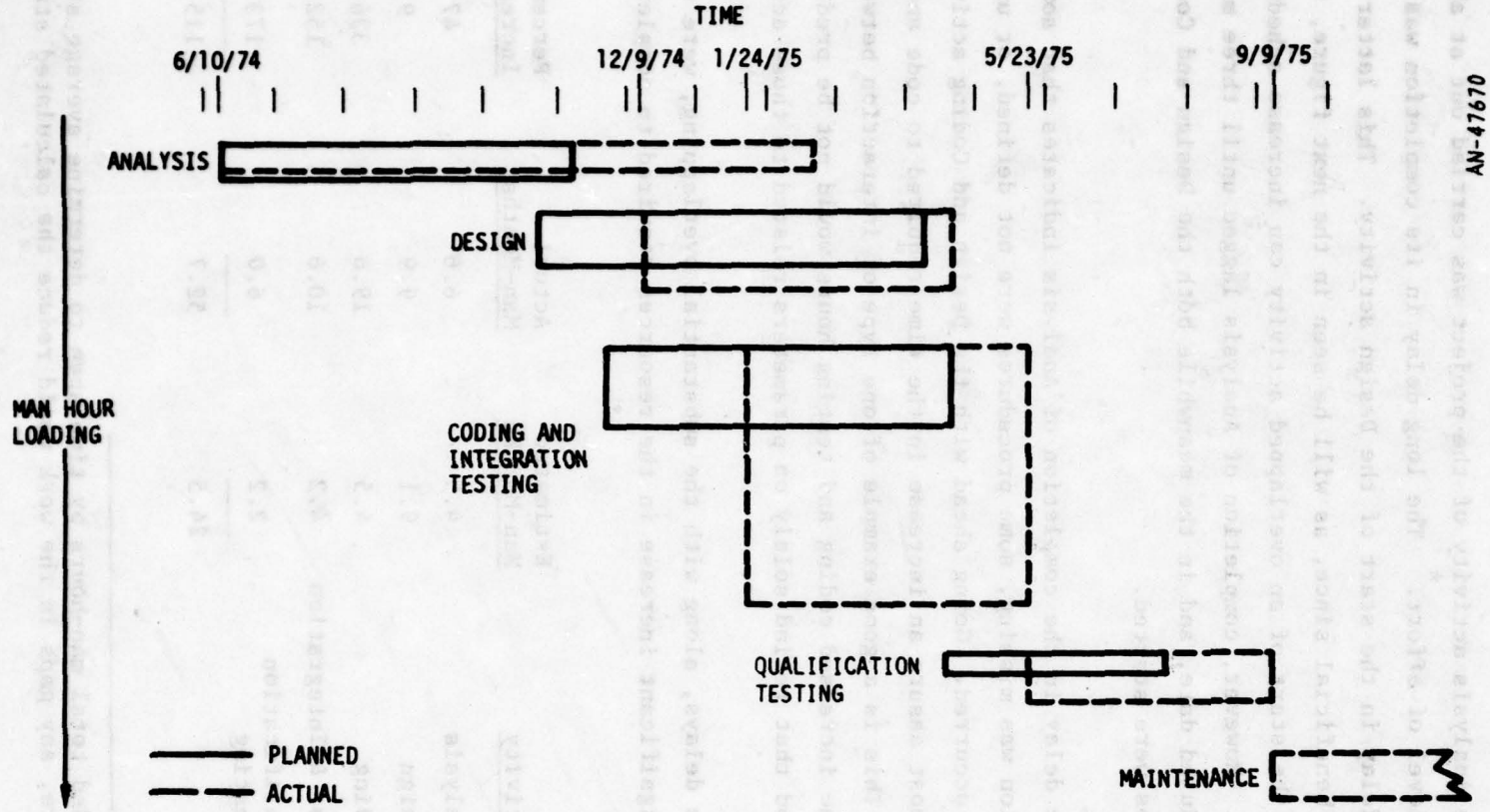


Figure 4.1. Scheduled and Actual Activities in a Software Development: Example 1

The Analysis activity of the project was carried out at about the planned level of effort.* The long delay in its completion was accompanied by some delay in the start of the Design activity. This latter delay was probably beneficial since, as will be seen in the next figure, failure to delay the start of an overlapped activity can increase scheduling problems. However, completion of Analysis lagged until three months after its scheduled date, and in the meanwhile both the Design and Coding activities were started.

The delay in the completion of Analysis indicates that some needed information was missing, some procedures were not defined, or unexpected problems occurred. Going ahead with the Design and Coding activities would almost assure an increase in the time required to code and test the system. This is a good example of one type of interaction between activities. The increased coding and testing hours would not be predicted by any method that relied solely on parameters related to those activities.

The delays, along with the substantial overlapping, were associated with a significant increase in the resources required to complete the project:

Activity	Estimated Man-Months	Actual Man-Months	Percent Increase
Analysis	4.5	6.6	47
Design	9.1	9.9	9
Coding	4.5	19.6	336
Test & Integration	4.2	10.6	152
Qualification Testing	2.2	6.0	173
	24.5	52.7	115

*We divided total man-hours by time span to determine average staffing. Therefore, any gaps in the work would reduce the calculated staffing.

The figures indicate that, for this project, delays and overlapped activities were associated with large increases in the consumption of resources over what was expected. The combination of delays and parallel activities has a compounding detrimental effect on a project schedule.* For example, when coding is begun before the completion of design, the designers are required to communicate their results to the programmers in a raw, unqualified state (hence significantly increasing the chance of design errors). Overlapping also raises the possibility that the designer may not change a poor procedure when he discovers it, because he has already committed himself to the programmer. Many times the programmer may fill in missing information by himself. By doing this he may introduce errors into the system that will not be discovered until late in the testing program when repairs will be time-consuming and expensive.

Adding programmers to a project that is behind schedule also introduces a communication problem and an associated decrease in productivity. Brooks's Law[20] is, "Adding manpower to a late software project makes it later." The existing staff must take time to lay out all the ground rules for the new members and describe all the details of the system. Since the system is behind schedule, the documentation is sparse and usually outdated. The result is that often it takes

---

*This is not to suggest that systems cannot be developed with overlapping activities. Many systems have distinct parts that can be coded before the entire design is completed. In a top-down design where coding is by tiers, the coding can often begin before the design is complete. These are planned developments that would permit the overlapping of these functions. We are concerned here with the situation where the press of the development schedule or the slippage of preceding activities results in overlapping activities that would have been accomplished better sequentially. Even in a planned implementation of parallel activities, however (and this includes top-down design), whenever the coding begins before the design is completed there is an increased risk of changes to the design or of mismatches in subsystem interfaces. The project management must weigh these risks in relation to the need for workload balancing and project scheduling.

, more time to explain what is to be done than to do it. Moreover, the situation is ripe for producing mistakes which can easily cause the group to be less productive than before the additional staff was assigned.

In the example we are considering, there was an additional reason for the increase in expended hours. The project was subjected to changes in the functional requirements during its development. It is difficult to say what proportion of the increases were caused by these changes, but we believe that the introduction of some changes is normal during project development. In this instance, changes made a bad situation much worse.

We are not yet trying to build a case for cause-and-effect relationships between delays and overlapping and increased consumption of resources. We are simply using an existing project history to illustrate how interactions among activities may be seen to influence the expected resource requirements. On the basis of a single project one could simply conclude that the project was poorly planned and executed. However, examining two other projects lends supporting evidence.

Figure 4.2 is for another business-application project. Here again there is heavy overlapping of activities. In this case, however, delays in the completion of analysis and design were not accompanied by a slip in the start of coding. Sometimes a project manager must assign personnel to his project or face the possibility of losing them. In many instances the programmers will be scheduled for another project on completion of the present effort and cannot delay the start of coding without jeopardizing the other project. If that was the case in this example, however, they still missed the start of the subsequent project by nearly three months.

As a final example, Figure 4.3 shows a project that was completed in a better fashion than the preceding two. The analysis and design

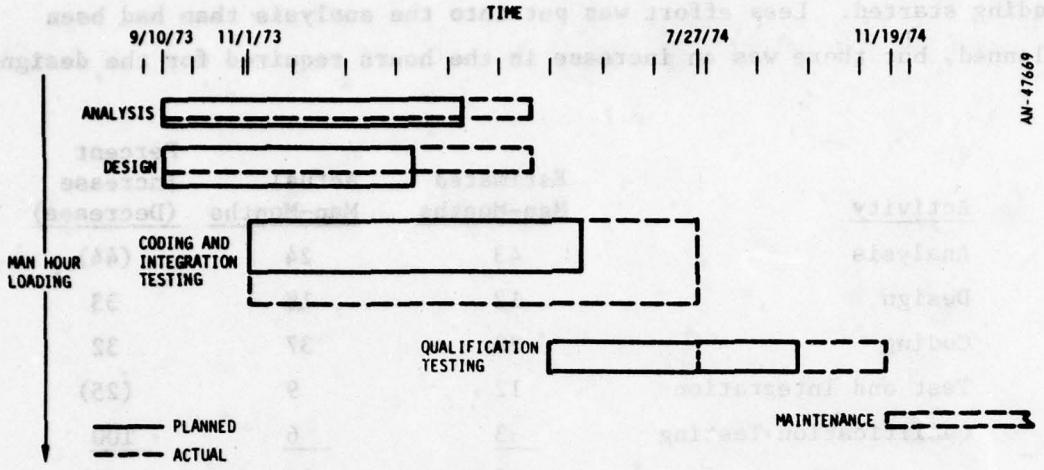


Figure 4.2. Scheduled and Actual Activities in a Software Development: Example 2

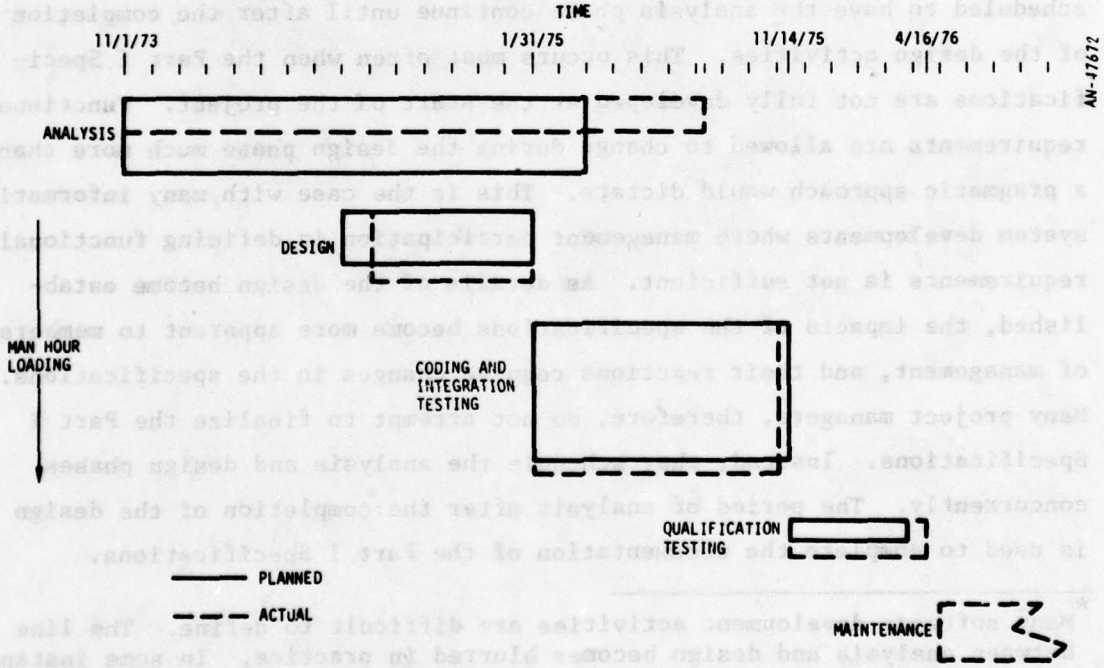


Figure 4.3. Scheduled and Actual Activities in a Software Development: Example 3

activities overlapped but, significantly, the design was completed before coding started. Less effort was put into the analysis than had been planned, but there was an increase in the hours required for the design:*

Activity	Estimated Man-Months	Actual Man-Months	Percent Increase (Decrease)
Analysis	43	24	(44)
Design	12	16	33
Coding	28	37	32
Test and Integration	12	9	(25)
Qualification Testing	3	6	100
	98	92	(6)

The project was completed on schedule.

Notice that the projects described in Figs. 4.2 and 4.3 were scheduled to have the analysis phase continue until after the completion of the design activities. This occurs most often when the Part I Specifications are not fully developed at the start of the project. Functional requirements are allowed to change during the design phase much more than a pragmatic approach would dictate. This is the case with many information-system developments where management participation in defining functional requirements is not sufficient. As details of the design become established, the impacts of the specifications become more apparent to members of management, and their reactions require changes in the specifications. Many project managers, therefore, do not attempt to finalize the Part I Specifications. Instead, they schedule the analysis and design phases concurrently. The period of analysis after the completion of the design is used to complete the documentation of the Part I Specifications.

*Many software development activities are difficult to define. The line between analysis and design becomes blurred in practice. In some instances both functions are performed by the same individual, who may also do some or all of the coding. It may be that in this instance some of the analysis hours were reported as design.

Business-oriented or information-system development projects are not unique in this practice of overlapping the analysis and design activities. It also occurs in non-business applications, C^3I, and other types of software development projects. It happens whenever the press of a schedule does not allow a proper definition of the Part I Specifications or when there is not sufficient knowledge of the requirements to formulate good specifications. If this situation exists, and a formal life-cycle development model is imposed on the project, the specification changes get reported as part of the coding and subsequent activities. Analysis of project data from this point of view suggests that the practice may be quite common. It is probably the most difficult problem to cope with when trying to use the reports of resource expenditures to determine the underlying controlling factors.

The preceding discussion has been presented in support of the contention that the relationships among the software development phases are extensive and very important to the consumption of resources. As will be shown later, these relationships extend into the operation phase.

In consideration of the arguments presented above, the model software development cycle shown in Fig. 4.4 will be used as the basis for formulating life-cycle man-hour relationships. The model is a generalization of the principal features identified in the examples taken from actual projects.

The analysis, design, and coding activities have been overlapped to represent a project constrained by its PCA completion date. The coding and integration-and-test activities have been staffed at a higher level than would have been used without the constraint. This would normally mean that a natural separation of the project along functional lines is altered to permit the assignment of additional programmers. The model shows a delay in completing the design, accompanied by the addition of more staff to the coding and testing effort. The completion of the testing phase is shown as slipping. The result of the way in

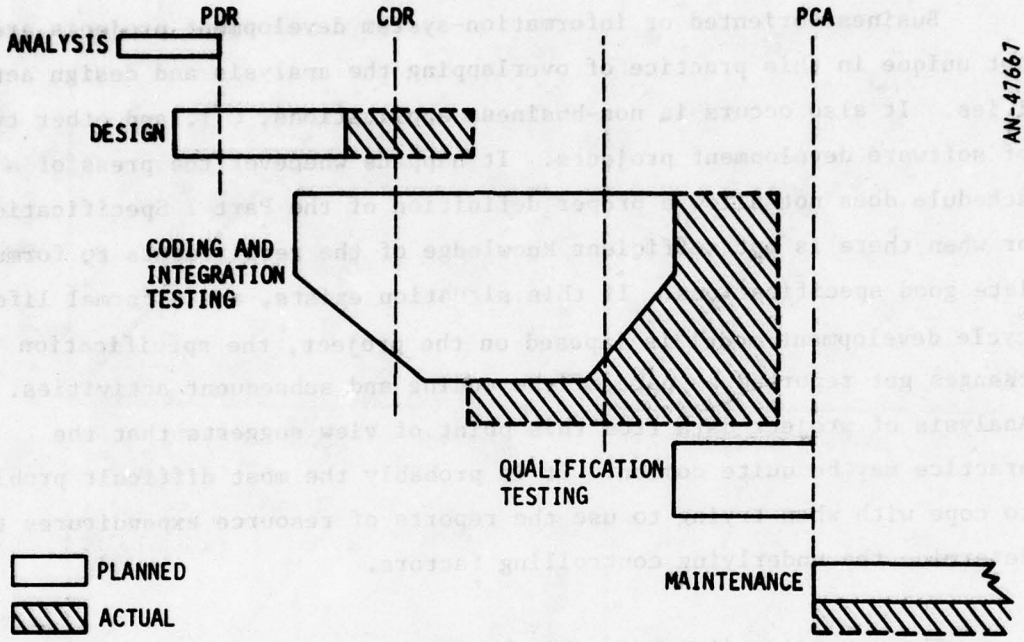


Figure 4.4 Model Software Development Cycle

which the project actually proceeds is an increased error rate in the developed software (over that planned for when originally assigning the maintenance staff). The maintenance staffing is correspondingly increased.

4.2 TRADE-OFFS DURING DEVELOPMENT

We have shown how interactions between project activities occur and some of the direct influences on resource requirements. In this section we discuss other consequences that result from deliberate actions by project managers. We are concerned with decisions to alter planned allocations of resources caused by departures from the development program schedule. We also consider the effects of calculated risks taken by managers in an attempt to bring slipping projects back on schedule.

In order to analyze the development trade-offs, it is useful to establish some hypothetical reference conditions or benchmarks. For each

software development activity, there is some ideal allocation of resources.* Figure 4.5 shows how these ideal allocations for each activity might look, taking the representative cost-driving parameter to be lines of source code.

In the first diagram, the linear function defines the man-months of analysis and design required for the estimated size of the system being developed. Because the design and coding activities overlap in the model development, the man-hours expended, represented by the + sign, is far below the ideal line. The project manager has accepted the risk that changes in design will delay the completion of coding, to avoid the schedule slip which would have resulted if coding had been delayed until design was complete.

The second chart indicates that the total expenditure on analysis and design was close to the ideal. However, because the completion of design took place later than planned (see Fig. 4.4), the coding effort suffered a decrease in productivity. Under these parallel development circumstances, the quality of the design would be expected to suffer.

The third chart shows that, as expected, the coding and testing activities consumed more resources than would have been indicated. The next chart indicates that the balance between the coding and testing activities was consistent with the ideal. The last two charts show that the scheduled maintenance effort was less than the ideal, while the reported faults were higher. The higher fault rate occurs because the testing effort was not sufficient to properly test the changes related to the delay in completing the design. The result of the mismatch between the maintenance effort and the error rate is a system that is forced to limp

*In order to describe the concept of trade-offs and phase relationships more easily, we have made two simplifying assumptions: first, we will assume that the single descriptor of the end product is the number of lines of source code; and second, we will assume that given the lines of source code, there is some ideal allocation of resources for each development activity. The discussion then describes how violating these hypothetical ideals influences the different activities.

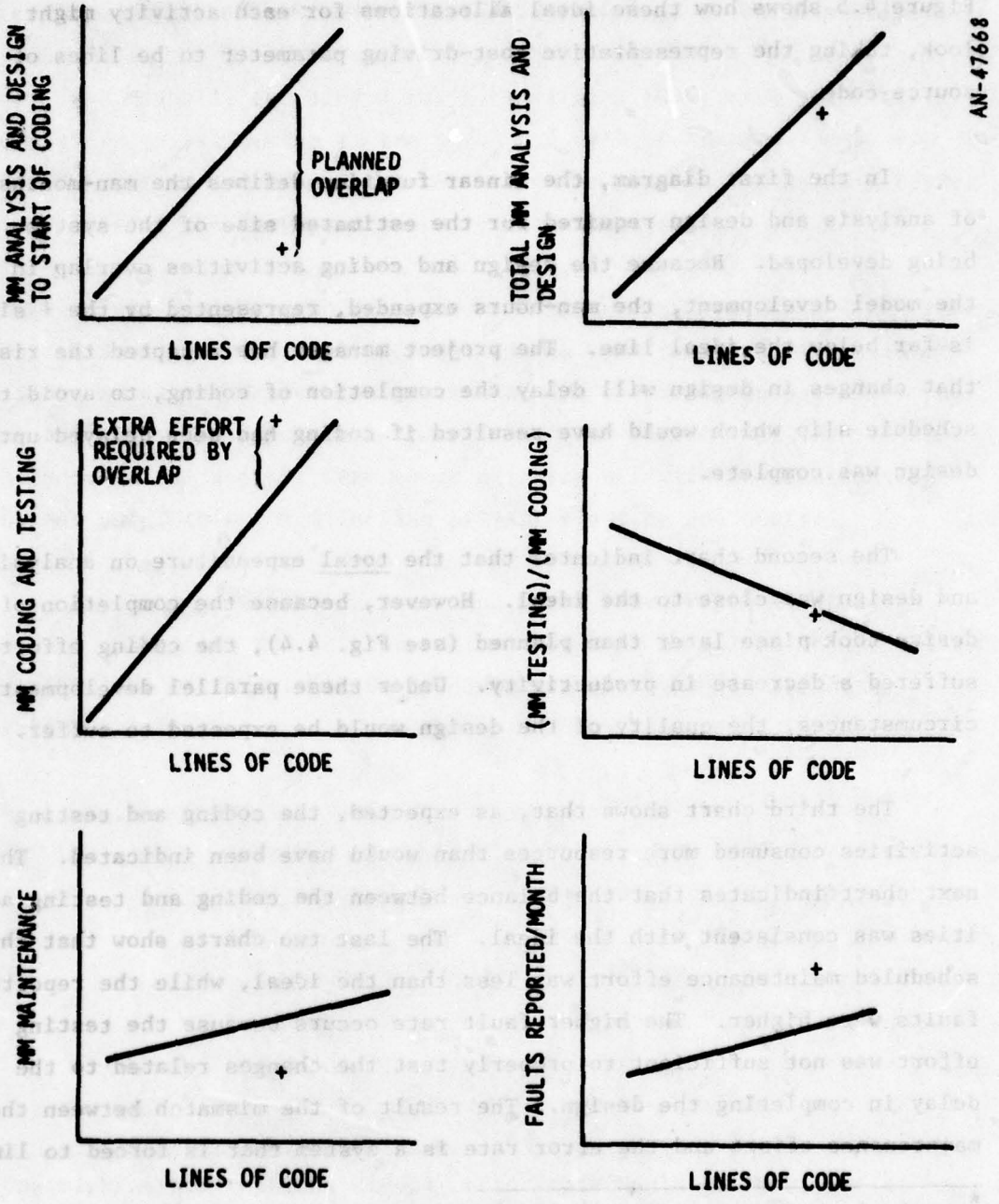


Figure 4.5. "Ideal" and Typical Resources Expended in Each Life-Cycle Activity

along with minor deficiencies while the major problems are patched by an overworked maintenance staff.

The preceding discussion was intended to illustrate how the assumed primary cost-driving parameters are influenced by decisions made during the development project. During each activity, management must make decisions that may increase development costs, cause slips in schedule, or risk problems later in the software life cycle. We have looked at one set of decisions that affect the cost-driving parameters through activity interrelationships. Many other sets of decisions are possible.

Figure 4.6 illustrates some other departures from the ideal which may occur, and how they may be reflected in the error rate of the delivered software, which is indicative of the reliability of the software. One external cost-driving parameter, changes (ECPs), has been added to those included in Fig. 4.5. The ">", "<", and "=" indicate whether the

	1	2	3	4	5	6	7	8
MM ANALYSIS AND DESIGN	>	=	=	<	=	>	=	=
MM CODING AND CHECKOUT	=	<	<	=	=	>	<	<
MM TESTING	=	=	>	=	=	>	<	=
MM/MO MAINTENANCE	=	=	=	=	=	=	>	>
CHANGES (ECPs)	NO	NO	NO	NO	YES	YES	NO	NO
REPORTED ERROR RATE	<	>	=	>	>	=	>	>

= EQUAL TO IDEAL
> GREATER THAN IDEAL
< LESS THAN IDEAL

Figure 4.6. Postulated Trade-offs Among Life-Cycle Man-Hour Parameters

project experience was above, below, or equal to the ideal for the given activity. Eight columns show eight different sets of relationships.

For example, the second column indicates that, with all other activities corresponding to the ideal and with no changes, less than ideal effort spent on coding and checkout would be expected to cause a higher error rate of the delivered software than the ideal.*

In conclusion: the development of low-risk, practical life-cycle cost estimating relationships requires the consideration of the interactions among the activities or phases. Furthermore, we postulate that any analysis that does not include these interactions will not succeed in reducing the scatter that makes existing software cost estimating schemes unsuitable for effective project planning and control.

Recognition of the interactions poses some practical problems in data collection and interpretation that must be solved before valid data will be obtained. These problems are discussed in the next section.

---

*One might argue that the "ideal" error rate would be zero; but a practical solution would be to avoid spending large amounts of resources to achieve zero errors. Therefore, it would be expected that proper planning would allow for some small acceptable error rate.

5 EVALUATING RELATIONS BETWEEN ACTIVITIES

Gathering enough data to evaluate the hypotheses proposed in Sec. 4 poses a problem. We had collected data from the Air Force Data System Design Center (AFDSDC) to test the relationships hypothesized in Sec. 3. This data base was selected because it contained detailed information on resource consumption, both planned and actual, for a number of software developments that used a common reporting system and the same programming language to produce software of the same generic type. With such a data base we could control for a number of variables that affect the man-hours.

When the data was tested, however, variances were large and it became clear that the model of Sec. 3 was too simplistic. The hypotheses that were stated in Sec. 4 evolved from studying this data. Unfortunately, the number of usable data points for testing these hypotheses is small, and the data base will have to be greatly expanded to yield statistically significant results. However, the data base is large enough to show the potential of this approach. This potential includes the development of rules for the optimal allocation of man-hours, in addition to the development of estimating relationships.

In this section we will first describe the data base we are using and its origin. Next, the data will be used directly to test, in a largely qualitative fashion, some of the relationships stated in Sec. 4. Finally, we will demonstrate how the data can be used to optimize resource allocations among activities or phases. Although the data base is too small to produce much confidence in the specific curves we show as examples, we believe that the technique and its potential are demonstrated.

5.1 DATA DESCRIPTION

5.1.1 The Air Force Data Systems Design Center (AFDSDC)

The Data Systems Design Center at Gunter Air Force Station, Alabama, has approximately 1300 personnel, 800 to 900 of whom are analysts and programmers. The mission of the Design Center is to develop and maintain all Air Force standard automated data processing systems; the Center does not have authority over systems that are unique to a command.

The Center is organized into functional directorates, each headed by a person experienced in that functional area. The Center supports over 140 data processing installations worldwide. At the Center there are three computer systems which can simulate any of the computer configurations used at these installations to implement the ADP systems developed at the Center. The primary system is the Burroughs B3500, which is called the base level computer. Supply systems are implemented on the Univac 1050. The Honeywell H6000 is used for applications at major-command installations. All applications are written in COBOL.

Requests from the different commands for new automated data processing systems are coordinated by the Air Staff, which authorizes the Design Center to work up a data processing plan for each approved proposal. A preliminary analysis is completed by the directorate having primary responsibility for the functional area, and the plan is prepared. The plan includes estimates of resources required for its development, implementation, and operation. Upon approval of the plan the directorate is responsible for developing, implementing, and maintaining the system.

When the developing organization has completed the development of a system or completed a major modification, the system is turned over to the Directorate of Systems Control for testing. The Quality Control

and Field Assistance Unit completes up to two levels of testing and maintains liaison between the developing organization and the field when operational problems occur.

The first test phase, Environmental Systems Test-I (EST-I), is completed at the Design Center and is intended to be an extensive checkout of the system in a simulated operational environment. For small systems EST-I is the end of testing before release. More complex new systems, or those subject to the National Privacy Act, are assigned to EST-II. This includes a complete testing and implementation plan and is conducted at from two to ten other installations to test the system in an operating environment.

There are two aspects to the tests: operational and functional. Operational testing means that, given certain data, the system will produce the specified reports and other actions. Functional testing is designed to verify that the information produced by the system is that desired by those who will use the system. Testing is considered complete when the Branch Chief receives from the test sites letters of certification on the operation and functioning of the system.

5.1.2 The Planning and Resource Management Information System (PARMIS)

Software development projects at the Design Center are supported by an automated system called the Planning and Resource Management Information System (PARMIS). The system is a project planning and management aid that enables project managers to enter estimated schedules and manpower allocations and to receive regular reports on actual utilizations.

PARMIS is activity-oriented. A project manager establishes a new project on PARMIS by estimating start and end times and man-hours for each activity. The breakdown of the project into specific activities is to a great extent up to the discretion of the manager. He may make the activities broad or narrow, according to the size and complexity of

the project and the number of persons involved. The selection of activity titles is facilitated by the PARMIS Catalog, which is a pre-programmed breakdown of activities to different levels of detail. One restriction imposed recently is that the activities must be consistent with the Design Center's standard life-cycle model. The PARMIS Catalog's designation of activity codes and activity group codes makes it possible to separate the development activities into activity groups. The activity group codes, presented in Table 5.1, were instrumental in assigning the resource utilization data to the process model used for this study.

PARMIS accepts planned and actual man-hours under four job classifications: Functional Analyst, Data Systems Analyst, Programmer, and Support Personnel. Actual expenditures of time are reported by activity and job classification. Both expected and actual start and completion dates are reported by the system for each activity. The project manager is free to revise his dates and man-hour estimates, but the system always maintains and reports the estimates made when the project was entered into the system.*

Actual expenditures of time are reported weekly. Each project manager is responsible for submitting reports of hours expended by person and activity. Regular reports are prepared at several levels beginning with the project level and extending to the Center level. When projects are completed, they are removed from the active data base and stored on history files.

---

*The relationship between estimated and actual man-hours has been previously investigated by Lt. Col. Gehring[45]. In particular, he correlated accuracy in predicting specific activities with accuracy in predicting total effort.

TABLE 5.1

PARMIS ACTIVITY GROUP CODES

Conception Phase

X010		Conception
	X011	Initial Processing
X020		ADP System Manager Evaluation
X030		Evaluation of Data Processing Requirements
X040		Preliminary Requirements Definition
X050		Preliminary Requirements Review

Definition Phase

X060		Definition of Requirements
	X061	Preparation of Data Processing Plan
	X062	Functional Description
	X063	Alternative Concepts
	X064	Economic Analysis
	X065	Preparation of SRR Document
	X066	Coordination of Functional Description
X070		System Requirements Review (SRR)

Development Phase

X080		System Design
	X081	Updated Data Processing Plan
	X082	System/Subsystem Specifications
	X083	Data Base Specifications
	X084	Data Requirements Document
	X085	Preparation of SDR Document
	X086	Hardware Specifications
X090		System Design Review (SDR)
X100		Program Development
	X101	Programming
	X102	Program Specifications
	X103	System Testing
	X104	Program Development Document
	X105	Test Plan
	X106	Computer Operation Manual
	X107	Program Maintenance Manual
	X108	Preliminary Implementation Requirements
	X109	User's Manual
X110		Technical Test Review
X120		System Status Review

Test Phase

X130		Environmental System Test I
X140		Environmental System Test II
X150		System Validation Review
X160		Worldwide Release

Operations Phase

X170		System Implementation
	X175	Final Operational Evaluation

Other

X350		Continuous Projects
	X351	Minor Changes
	X352	Individual Minor Projects
X360		Future Projects
X400		AF-Provided Software
X410		Vendor-Provided Software

5.1.3 Description of the Data Base

PARMIS has been in use at the Design Center since 1970. Since that time more than 2000 project summaries have been accumulated in the history files.

The History File for each project is the activity-level file as it existed when the last activity was reported completed. There is a separate record for each activity and each activity group. Table 5.2 lists the elements of the History File. History Files are maintained on a fiscal year basis. A project is entered into the History File for the fiscal year in which its last activity was completed.

The History Files reside on magnetic tapes. The preparation of special reports is greatly facilitated by a special report generator system.

Projects to be included in this study were selected by first examining summary reports on all projects completed during Fiscal Years 1975 and 1976. Special reports on candidate projects were then prepared by using the History Files and the report generator.

5.1.4 Data Collection Procedure

Collection of the project data was a three-step process. First, detailed information was collected for candidate projects. This included details about estimated and actual dates and man-hours for each activity in each project. Second, a detailed questionnaire was prepared and interviews were conducted with project managers to obtain information not available in PARMIS: program size, management techniques, documentation, and other items describing the product and its development environment. Finally, the error reports maintained by the Field Assistance Branch were studied to determine the numbers and types of errors reported by users and when they occurred.

TABLE 5.2

SUMMARY OF HISTORY-FILE DATA ITEMS

(EACH ACTIVITY)

1. Project originator number
2. Activity group number
3. Control number
4. Activity number
5. Activity description
6. ADP system number
7. ADS number
8. Management category
9. Milestone number
10. Type of computer
11. Work category
12. Data systems designator number
13. System code
14. Type of system
15. Program action code
16. Program number
17. Schedule indicator
18. Responsible individual for data
19. Privacy key
20. All predecessor activities (control and activity numbers)
21. All successor activities (control and activity numbers)
22. Activity description
23. Start date
24. New start date
25. Estimated completion date
26. New estimated completion date
27. Actual completion date
28. Plan change date
29. Span days
30. New span days
31. Remaining span days
32. Estimated man-hours by skill and total
33. New estimated man-hours by skill and total
34. Expended man-hours by skill and total
35. Monthly expended man-hours by skill and total
36. Current expended man-hours (since plan change date) by skill and total

PARMIS Data Collection. Summary reports from the history files for Fiscal Years 1975 and 1976 (year-to-date) were examined. These reports contain start and completion dates and estimated and actual man-hours by job classification and by activity group for each project. Personnel in the Project Management Division, Operations Branch provided consultation on use of the history files and prepared the computer runs for obtaining the project summaries. The following criteria were used to select projects for the study.

1. Project completed between January 1974 and April 1976. This was to insure that some error history would be available, but that the project was not completed so long ago that getting management information would be difficult.
2. Activity descriptions including entire software development life cycle.
3. Projects greater than 2,000 actual man-hours and six months duration.

Each project is identified on the history file by its ten-character Project Originator Number (PON) which is established at the time the project is authorized. Using this number as the primary search key, a special file was created and sorted by PON, contributing organization, activity groups, and control number. This sequencing had the effect of separating each project by organization and development phase. Using the activity group code and control number as breakpoints, totals of estimated and actual hours were automatically prepared for each development phase.* Figure 5.1 shows a page from the special history report.

Project Manager Data. The personnel of the Project Analysis Branch obtained the names of the project managers for the 20 projects

*In some projects the project managers used slightly different definitions of the development phases. In these cases revised totals were calculated manually.

PREPARED 16 SEP 83 R-L-I-S R-E-P-O-R-T CAN # 47358 AS OF 791A1 PCN N11A007

BEGIN PARMIS II-30JUN75 HISTORY TAPE STUDY

PGNB ORG GROUP ACTIVITY-WBR ACTIVITY-DESCRIPTION	EST-FA EST-ST	EST-OSA NEW-ST	EST-PGM EST-CP	EST-SPT ACT-CP	EST-TOT	EXP-FA	EXP-OSA	EXP-PGM	EXP-SPT	EXP-TOT
73080 ACF V030 7300766D012 FLOW CHART INTERFACE	0 780501	0 780531	50 780819	0 780830	50	0	0	135	0	135
73080 ACF V030 7300766D013 FLOW CHART UPDATE	0 780513	0 780513	35 780819	0 780830	35	0	0	202	0	202
73080 ACF V030 7300766D014 FLOW CHART REPORT PGMR	0 780513	0 780513	35 780819	0 780901	35	0	0	34	0	34
73080 ACF V030 7300766D019 CODE INTERFACE	0 780301	110 780301	0 780531	0 780830	110	0	0	401	0	401
73080 ACF V030 7300766D021 CODE INTERFACE	0 780301	0 780331	250 780531	0 780918	250	0	0	123	0	123
73080 ACF V030 7300766D022 CODE UPDATE	0 780401	200 780331	120 780807	0 780907	320	0	0	22	0	22
73080 ACF V030 7300766D023 CODE UPDATE	0 780401	0 780401	0 780807	0 780830	0	0	0	181	0	181
73080 ACF V030 7300766D024 CODE REPORT PGMS	0 780401	80 780401	0 780808	0 780517	80	0	0	32	0	32
73080 ACF V030 7300766D025 CODE REPORT PGMS	0 780301	0 780331	100 780808	0 780517	100	0	0	15	0	15
73080 ACF V030 7300766D026 CODE REPORT PROGRAMS	0 780301	60 780331	0 780808	0 780901	60	0	5	186	0	191
73080 ACF V030 7300766D032 KEYPUNCH INTERFACE	0 780401	0 780401	20 780530	0 780830	20	0	0	53	0	53
73080 ACF V030 7300766D033 KEYPUNCH UPDATE	0 780401	0 780401	40 780807	0 780830	40	0	0	32	0	32
73080 ACF V030 7300766D034 KEYPUNCH REPORT PGMS	0 780401	15 780401	0 780808	0 780517	15	0	0	10	0	10
73080 ACF V030 7300766D035 KEYPUNCH REPORT PGMS	0 780301	10 780301	0 780808	0 780901	10	0	10	76	0	86
73080 ACF V030 7300766D041 COMPILE REPORT PGMS	0 780301	0 780501	80 780805	0 780901	80	0	0	74	0	74
73080 ACF V030 7300766D042	0	60	0	0	60	0	0	18	0	18

Figure 5.1. Sample Printout From the PARMIS Special History Report

whose summaries were obtained from the PARMIS data base. They coordinated initial interviews with the managers (or other persons familiar with the projects) and circulated copies of the questionnaires (see Appendix A).

About a week after the questionnaires were distributed, they were collected and the contents were discussed. In some cases problems were discovered and entries were changed or new data were indicated. Several respondents needed additional time to complete the forms.

In all, 17 completed forms were obtained. Of the three other systems for which information was requested, one had been replaced by a newer system and all related records destroyed; the other two were modifications to a large logistics system, and their records were not separable from those of the primary system and therefore were not collected for this study.

Of the 17 forms that were returned, 10 were missing program-size information. This information is not usually recorded, and the systems for which we were collecting the data had been modified to the extent that the present program sizes were not indicative of the originals.

Therefore, of the 20 systems for which information was requested, usable data were obtained for only seven.

Error Data Collection. Reports of system difficulties are processed by the Field Assistance Branch of the Systems Control Directorate. Each trouble call or difficulty report on an operational system is logged. Referrals are made to the directorate responsible for system maintenance, and follow-up contacts are maintained until the report is determined to be false or a duplicate of a previously reported error, or until a correction is released. A very complete description of all reported problems and their disposition is maintained. Summary reports are released regularly.

Unfortunately, records on individual systems are maintained for only the preceding 12 months. As a result, error reports were not available for one of the systems. In addition, it was learned that errors in another system are not reported through the Field Assistance Branch because the system is used only at the Design Center. This left error summaries for five of the seven systems.

5.1.5 Data Quality

The data obtained from PARMIS describing activity dates and hours should be of as high quality as can be obtained for software development analysis. It was constructed according to well-established definitions and procedures of long standing. It was recorded weekly as it happened, and reflects activity definitions that are directly applicable to our analyses.

Errors in the PARMIS data could come from poor management reporting, misuse of activity definitions in establishing and reporting projects, misleading representations of project status submitted to hide slippages, and dumping of idle time into active projects. However, these errors would exist in any project reporting system and should be minimized by the procedures in effect at the Design Center.

The error reporting data are part of a very extensive system of quality control in effect at the Desgin Center. The complete logging and follow-up of each difficulty, made by a unit that is separate from the developing unit, insures the quality of these data.

The project manager (questionnaire) data is the weakest. The questions asked for very detailed information that is not part of the records kept by the managers. Furthermore, military personnel are transferred frequently, and several of the project managers had been transferred. The Design Center uses a project-oriented organizational structure within the directorates, and therefore it was sometimes

difficult to obtain information because persons who worked on the system were scattered.

However, we did locate knowledgeable individuals for each project, and through their cooperation and patience it was possible to locate the records required. Old program listings were searched for program size information, notes were dusted off, and telephone discussions were held with other persons who had worked on the projects. By this process it was possible to obtain much of the data that was asked for.

5.1.6 Data Conditioning

The results of the data collection effort are presented in Table 5.3. The first 19 items for each project were derived from the PARMIS history files; items 20 through 27 were obtained from the project-manager questionnaires and item 28 from the error reports; and items 29 through 34 were derived from the other items.

Estimated and actual values were determined from the activity-level printouts from the history files. Activity descriptions were checked to assure that the activities were included in the phase that had been established by this study's process model.

In some projects, the managers had entered activities designated as PDR or CDR. The dates were recorded for these events. Otherwise, the dates for completion of the analysis activities and the design activities, respectively, were recorded as the PDR and CDR dates.

To establish the hours of analysis and design completed before the start of coding, all the coding activities were scanned to determine the earliest actual starting date. The estimated and actual hours for analysis and design activities before this date were summed to arrive at that entry. If an analysis or design activity spanned the start-coding date, it was proportioned between the "before" and "after" totals, assuming a constant level of effort.

TABLE 5.3

SUMMARY OF DATA FROM DATA SYSTEMS DESIGN CENTER

Data From PARMIS Records	[1] Estimated	[1] Actual	[2] Estimated	[2] Actual	[3] Estimated	[3] Actual	[4] Estimated	[4] Actual	[5] Estimated	[5] Actual	[6] Estimated	[6] Actual	[7] Estimated	[7] Actual
Dates														
1. Start Project		10/10/74		6/10/74		11/1/73		9/10/73		1/21/74		11/1/73		8/7/72
2. Complete Analysis	6/2/75	6/2/75	11/20/74	2/21/75	8/1/74	7/26/74	3/11/74	4/19/74	7/3/74	6/27/74	3/31/75	8/8/75	4/15/74	8/15/74
3. PDR	1/28/75	2/4/75	11/14/74	1/17/75	7/1/75	7/1/74	3/11/74	4/19/74	5/13/74	5/31/74	3/31/75	6/11/75	12/5/73	5/25/73
4. Start Design	11/4/74	11/4/74	10/23/74	12/9/74	6/17/74	6/17/74	9/10/73	9/10/73	2/4/74	2/4/74	7/1/74	8/1/74	3/8/74	3/8/74
5. Complete Design	3/3/75	5/1/75	4/15/75	4/22/75	1/17/75	1/30/75	2/6/74	4/19/74	7/3/74	7/3/74	1/31/75	1/31/75	7/17/74	8/15/74
6. CDR	6/2/75	6/2/75	1/15/75	4/1/75	1/17/75	1/30/75	2/1/74	4/19/74	7/3/74	7/3/74	1/31/75	1/31/75	3/14/74	6/18/74
7. Start Coding	11/18/74	11/18/74	11/25/74	1/24/75	2/3/75	2/3/75	11/1/73	11/1/73	3/11/74	3/11/75	2/3/75	2/3/75	5/23/73	5/23/73
8. Complete Test & Int.	7/1/75	7/9/75	4/15/75	5/23/75	9/15/75	8/8/75	5/13/74	7/27/74	8/14/74	10/1/74	11/17/75	11/14/75	9/5/74	9/10/74
9. Complete Qual. Test	Unk	4/15/76	8/15/75	9/9/75	12/12/75	11/14/75	10/15/74	11/19/74	8/7/74	10/16/74	4/16/76	4/16/76	10/16/74	10/16/74
Man-Months														
10. Analysis	1.25	0.63	4.51	6.58	14.74	7.81	5.32	4.34	2.78	5.28	43.31	23.61	20.90	18.50
11. Design	9.13	5.40	9.06	9.92	8.08	5.92	9.41	9.19	1.35	0.80	11.72	16.07	6.11	6.11
12. Coding & Checkout	24.72	29.05	4.53	19.62	8.13	5.58	12.01	15.42	10.42	8.65	27.75	36.95	29.08	25.25
13. Test & Integration	11.19	25.78	4.18	10.57	1.49	3.30	6.94	14.81	2.22	3.08	11.56	8.69	11.67	9.73
14. Qualif. Testing	1.18	1.75	2.22	5.97	1.67	3.38	5.73	5.39	1.35	1.24	2.57	5.74	2.65	2.53
15. Documentation	5.03	5.89	3.76	5.76	7.08	3.57	2.36	5.19	2.50	2.72	18.01	15.83	4.81	5.24
16. Development, total	53.20	68.50	28.26	58.42	41.19	29.56	41.77	54.34	20.62	21.77	114.92	106.89	75.22	67.36
17. Analysis + Design, total	10.38	6.03	13.57	16.50	22.82	13.73	14.73	13.53	4.13	6.08	55.03	39.68	27.01	24.61
18. Analysis + Design, before coding		1.73		11.68	22.82	13.73	3.17	2.95	4.13	6.08	55.03	39.68	17.01	15.81
19. Coding + Testing		56.58		36.16		12.26		39.42		12.97		51.38		37.51

TABLE 5.3 (Contd.)

	1	2	3	4	5	6	7
Data from Questionnaires							
20. No. changes during development	3	0	0	13	Unk	0	0
21. MM maintenance per month	0.42	1.45	0.083	0.59	0.38	2.14	6.83
22. Computer hours for development	Unk	51	22	Unk	Unk	128	Unk
23. Number of programs	33	10	5	23	24	32	29
24. Lines of source code	36,050*	18,246	5,100	37,754	36,224	25,460	25,900
25. New lines of source code	14,500	18,246	5,100	24,500	36,224	17,500	25,900
26. Percent based on existing analysis	90	0	75	35	65	0	0
27. Percent based on existing design	90	0	25	35	25	0	0
28. Reported faults/month	19/4 (4.75)	2/11 (0.18)	2/9 (0.22)	0(1st yr. missing)	Unk	6/4 (1.50)	5/10 (0.50)
Derived Quantities							
29. Effective lines of source, for analysis	3,600	18,246	1,300	25,000	13,000	17,500	25,900
30. Effective lines of source, for design	3,600	18,246	3,800	25,000	27,000	17,500	25,900
31. Effective lines of source, avg. for analysis + design	3,600	18,246	2,550	25,000	20,000	17,500	25,900
32. Effective man-months, analysis + design	14.43	16.50	16.28	13.04	21.38	39.68	24.61
33. Effective man-months, analysis + design before coding	10.13	11.68	16.28	2.48	21.38	39.68	15.81
34 Proportions: eff. A+D/C+C/Test	20/41/39	31/38/31	56/20/24	27/32/41	62/25/13	43/41/16	39/41/20

*Assembly language; all others COBOL.

Several of the systems contained parts that were taken from previously developed systems. Also, some of the analysis and design had been done under previous projects. Adjustments to the raw data were made to relate all the analysis and design and coding and testing hours to the new product developed and not necessarily the end or delivered product.

Corrections for Existing Analysis and Design. Systems 1, 3, 4, and 5 used existing analyses and designs to develop parts of the programs. The proportions are given for each case (lines 26 and 27). Of the total source-code lines delivered (line 24), only a portion were derived from the man-months of effort shown in lines 10 and 11. The percentages given in lines 26 and 27 were used to make the "effective" source-code lines shown in lines 29 through 31 consistent with the level of effort. These data were used to relate the analysis and design hours to program size (see Fig. 5.2). The procedure used in making the corrections is described next.

Corrections for Program Size. In the following analyses, resources expended during the different development activities will be compared. We will attempt to compare man-months of time required to complete each activity, design changes, and errors, with product descriptors in order to discover if quantitative relationships can be established. To make such comparisons, it is necessary to have some measure of the product; we have selected program size as the single measure. Since we want to compare the different resource requirements for a given end product, it is necessary that all the resources for a given program development be consistent with that measure. Unfortunately, some of the programs in Table 5.3 incorporated existing code or designs. For those that included existing code, only the new code (line 25) was counted, since this is the product produced by the development resources expended.

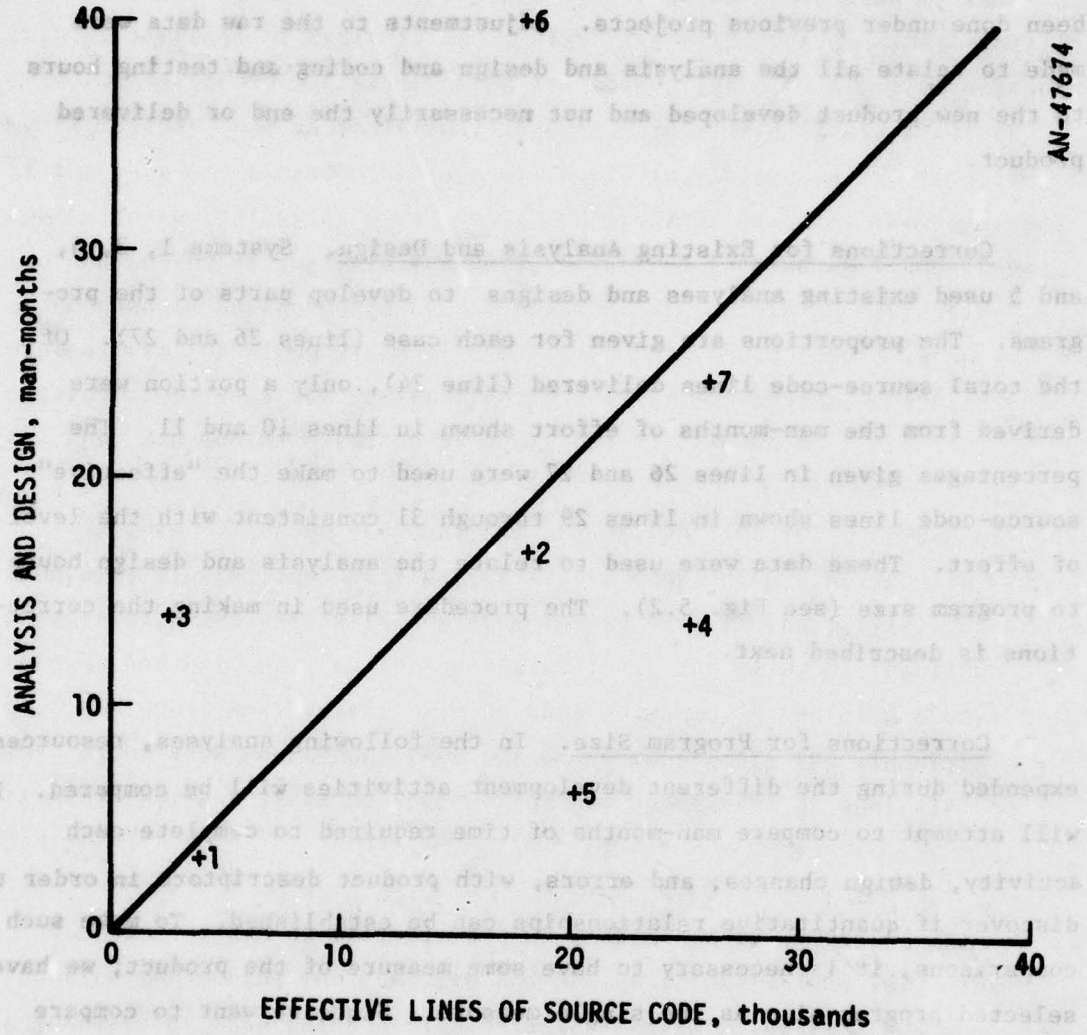


Figure 5.2. Relationship Between Analysis and Design Time and Program Size

In the case of existing design or analysis, it was reasoned that the recorded man-months were not as much as would have been needed to produce the new lines of code from scratch. Therefore, we adjusted the hours of design or analysis so that they represented what would have been required to produce the new lines of code. Only by making these corrections was it possible to make comparisons among the different activities for the same program development. These adjusted values are used only for examining the analysis and design trade-offs with programming and testing presented in Sec. 5.3; uncorrected values are used to establish the baselines for each development activity

Two methods were considered for adjusting the analysis and design man-months for program size. One possibility is to use the percentage figures on each project to calculate the effort that would have been required if analysis and design had been done from scratch. For example: suppose a program has 2,000 new lines of source code, and two man-months were spent in design, and 50 percent of the program was based on an existing design. Then the reported design man-months were associated with the production of 1,000 lines of code, and we would calculate a rate of two man-months per 1,000 lines of code. If we accept this method of correction, we would describe the "effective" design time for the 2,000 lines of new code as four man-months.

The problem with this method is that it greatly increases the analysis and design hours when the new lines are only a small part of the programs. But it is reasonable to expect that the relationship is not linear for small analysis and design efforts; that is, some amount of analysis and design is necessary, however little new code is to be written.

Because of this, we chose to use the slope of the regression line in Fig. 5.2 (one man-month per 1,000 lines) as an indication of the rate of increase of analysis and design time with program size. This measure has the advantage of being representative of a group of measurements, rather than an individual case, and is a more likely predictor of the rate of change. The rate of change is applied to the reported months in the following way. Using the previous example, the reported two man-months of analysis and design required to produce 1,000 lines of code would be increased by one man-month to obtain an "effective" expenditure for 2,000 lines of new code. This would result in an effective analysis and design effort of three man-months, compared to the four man-months obtained by using the first method.

Figure 5.2 shows the analysis and design time for a number of programs, plotted against lines of source code. The line results from a regression analysis with the line forced through the origin. Data point 5 was not included in the regression. The slope of the line (1.0 MM/1000 lines) is the same as that obtained in the NASA study.[22] This gives some confidence that the slope represents the linear relationship between lines of source code and man-months of analysis and design.

Lines 32 and 33 of Table 5.3 were obtained by adjusting the reported man-months for analysis and design (lines 17 and 18). The adjustment of data point 3 will be described to illustrate how the entry for line 32 was obtained. (The corresponding entry for line 33 is obtained in the same manner except that the entry on line 18 is used instead of line 17.)

The delivered product has 5,100 new lines of source code (line 25). According to the questionnaire, 75 percent of the code was written using an existing Part I specification (line 26) and 25 percent was written from an existing Part II specification (line 27). Therefore, the reported man-months for analyis and design represent the effort required for end products of 1,300 and 3,800 lines of source code (lines 29 and 30). The mean value, 2,550 lines of code, is used to represent the combined analysis and design effort (line 31).

As was described above, the "effective" analysis and design man-months consistent with the end product value of 5,100 lines of code were then obtained by calculating the additional effort that would have been required to produce the program if none of the analysis and design had existed at the start. An additional analysis and design effort equal to that required to produce 2,550 lines of source code (5,100 - 2,550) would have been required. At a rate of one man-month per 1,000 lines, this would require 2.55 additional man-months. Therefore, an effective value of 16.28 man-months (13.73 + 2.55) is entered in line 32. This value is used in subsequent analyses when analysis and design time is compared with coding and testing time.

Corrections for Programming Language. All the systems except number 1 are written in COBOL; System 1 is written principally in assembly language. Our results from studying the ADPREP data (Fig. 6.2) were used to correct the man-months presented for System 1 and to make them consistent with the other data. Accordingly, man-months for System 1 are multiplied by 0.54 (see equations following Fig. 6.2) when they are compared with the other systems.

5.2 EVALUATING RELATIONS AMONG ACTIVITIES

The presentation of the Air Force Data Systems Design Center data is designed to support the study of the relations among activities

discussed in Sec. 4. In accordance with the previous discussion, we are attempting to show the following relationships:

1. Extending the analysis and design activities beyond the start of the coding activities increases the probability of more hours of programming and testing, or increased operational errors.

2. Increasing analysis and design time tends to decrease the programming and testing time or the number of operational errors.

We are attempting in these comparisons to demonstrate how one activity of the software life cycle is affected by and in turn affects other activities. Obviously, most of these relationships cannot be analyzed independently. That is, we cannot compare analysis and design time with programming and testing time without at the same time considering the number of operational errors. As was indicated by Fig. 4.6, there are many components to each comparison. With the limited data that are available it will not be possible to test the various relationships rigorously. In this section we attempt to show that the basic relationships indicated by the hypotheses are supported by the available data. In the next section we will then show what quantitative results can be obtained and by so doing indicate the direction for future work.

Figures 5.3 to 5.8 are plots of the man-months expended for the various activities of each of the seven projects against program size. The clearest conclusion that can be drawn from these plots is that the wide scatter (particularly in the Analysis and Design plots) makes it futile to attempt to derive estimating relationships for the separate activities. The fit lines shown in Figs. 5.3, 5.4, and 5.7 will be discussed later.

This scatter is really quite surprising. We remind the reader that these seven data points represent projects all completed under the

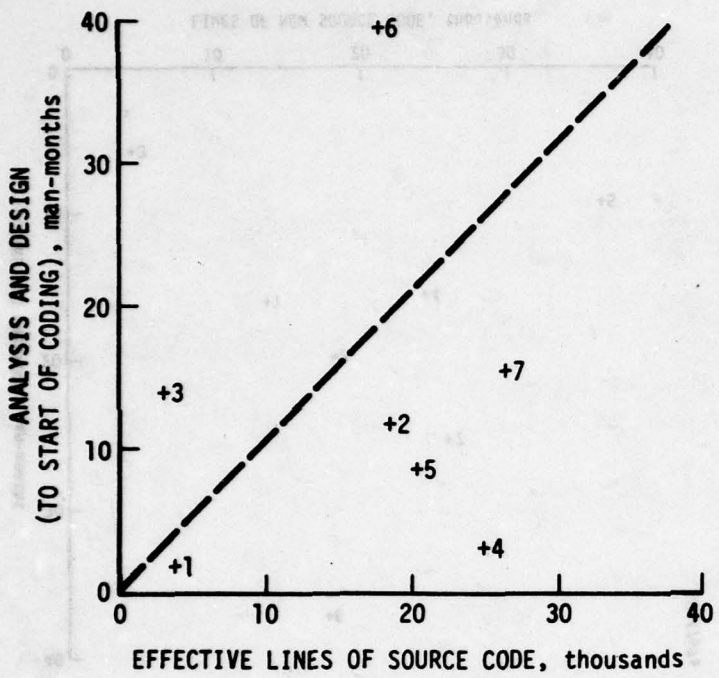


Figure 5.3. Analysis and Design Time, Before Coding

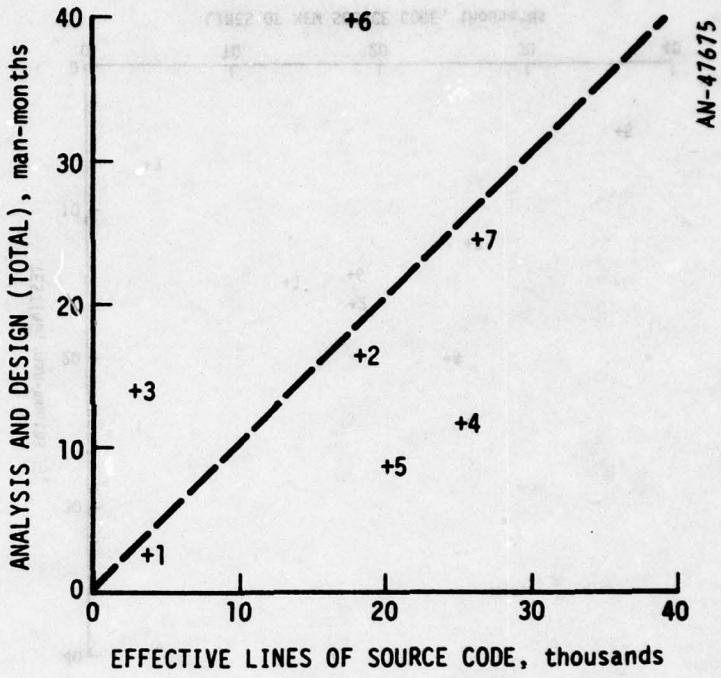


Figure 5.4. Analysis and Design Time, Total

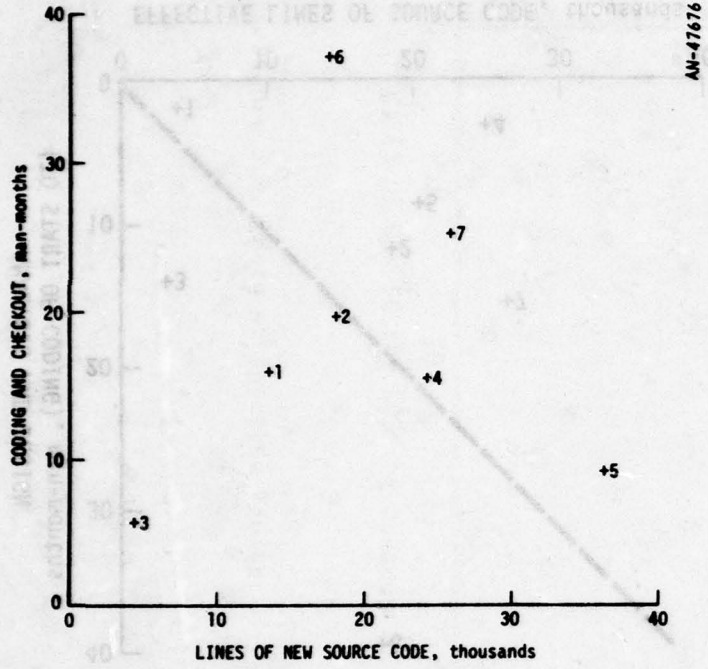


Figure 5.5. Coding Time

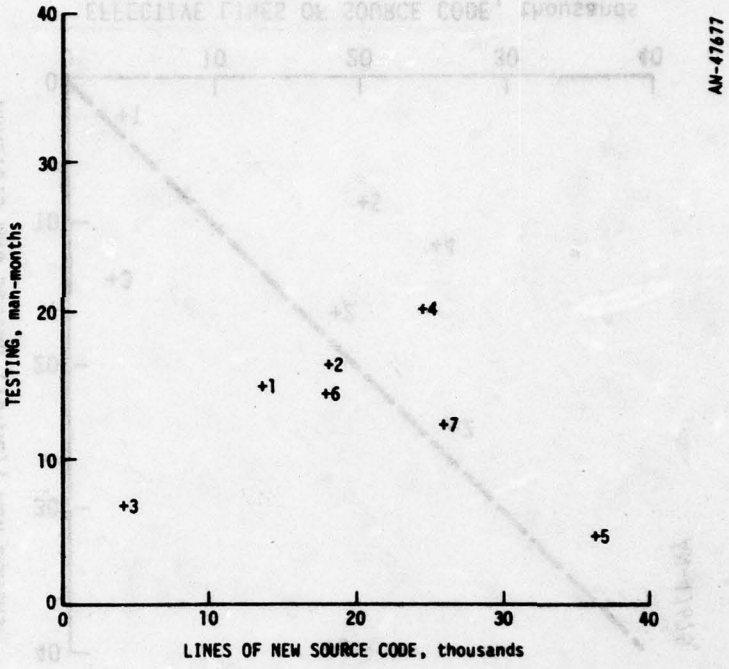
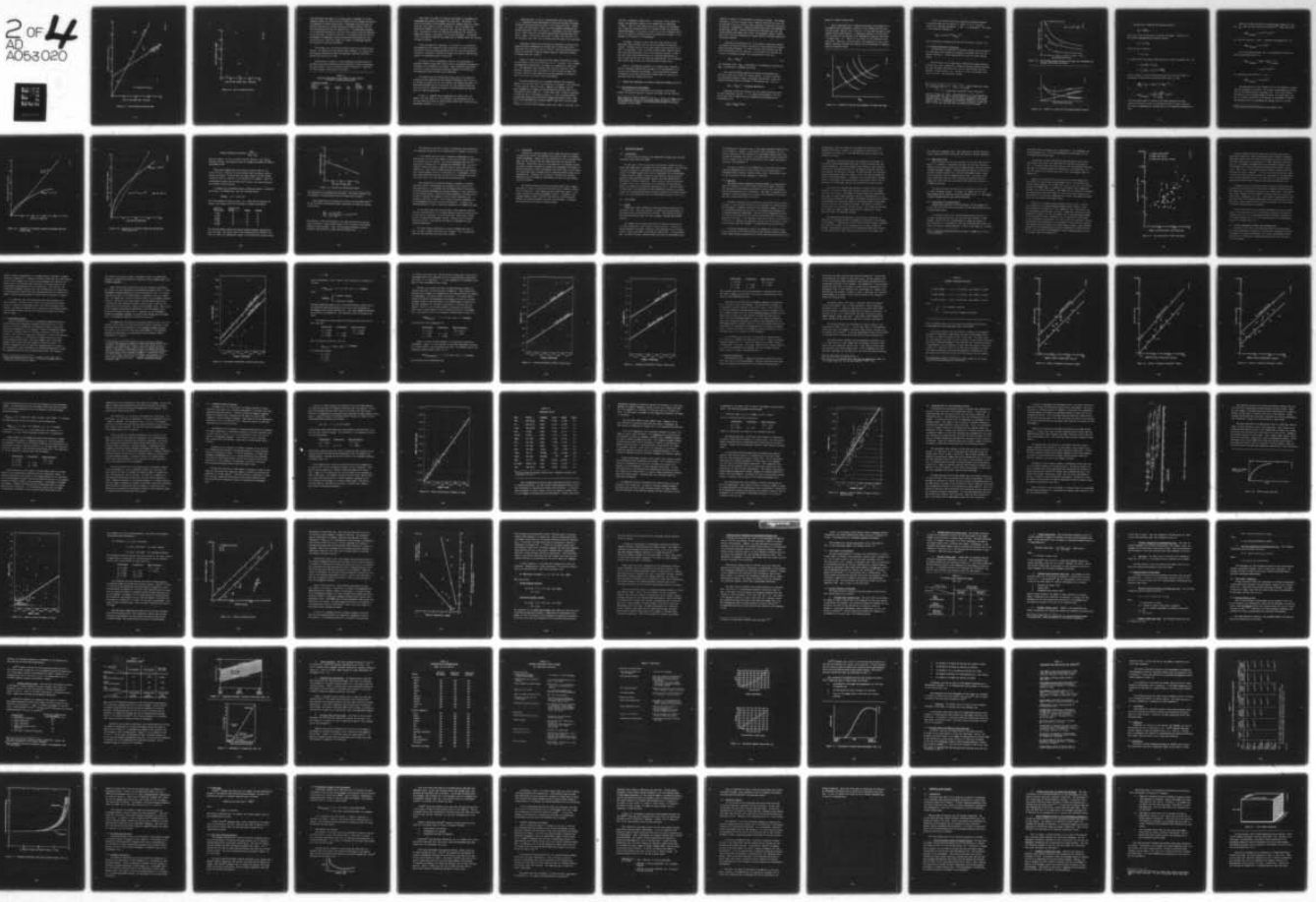


Figure 5.6. Testing Time

AD-A053 020 GENERAL RESEARCH CORP SANTA BARBARA CALIF F/G 9/2
COST REPORTING ELEMENTS AND ACTIVITY COST TRADEOFFS FOR DEFENSE--ETC(U)
MAY 77 C A GRAVER, W M CARRIERE F19628-76-C-0180
UNCLASSIFIED CR-1-721-VOL-1 ESD-TR-77-262-VOL-1 NL

2 OF 4
AD
A053020



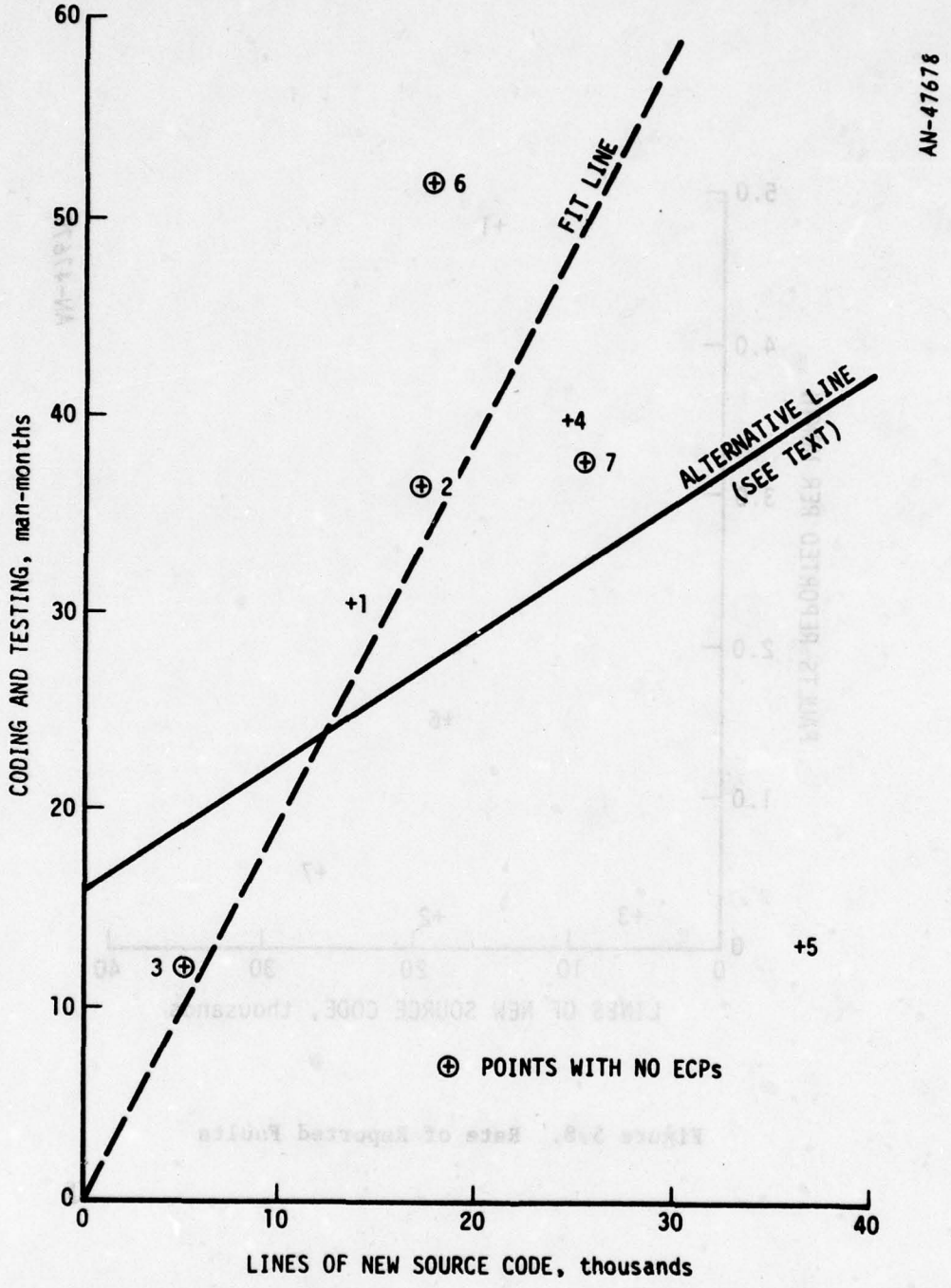


Figure 5.7. Total Coding and Testing Time

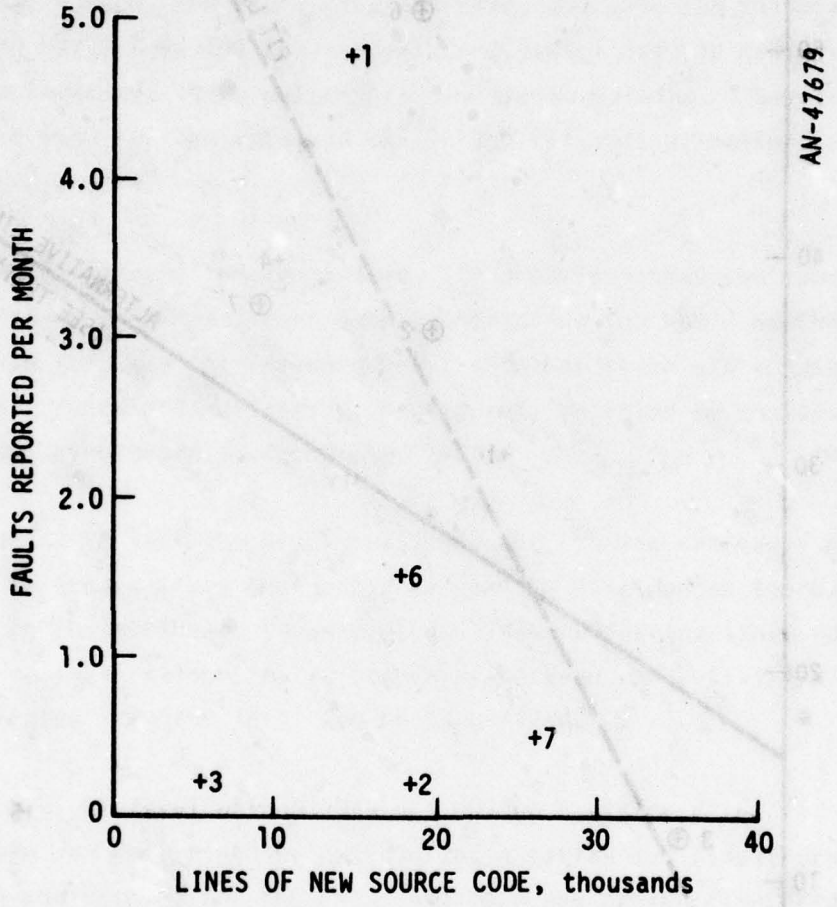


Figure 5.8. Rate of Reported Faults

same development philosophy at the same agency; programmed in the same language (except one, which has been adjusted for the language difference); and designed for similar applications. Adjustments have been made for projects that built on previous analysis and design efforts, with the data representing only the new coding effort. If independent estimates of the resources required for each life cycle activity could be supported by any data base, it should be this one. We therefore conclude that the development of independent resource-estimating relationships for each activity does not offer any promise as a means of estimating resource requirements for software development.

This brings us to the second conclusion that can be drawn from the figures. On examining all the figures taken together, it is clear that the relations between activities are important, and that trade-offs between the activities do occur.

Perhaps this is easiest to see by ignoring the program sizes for the moment, and simply ranking the seven programs in order of man-months for each activity. Such an ordering is shown in Table 5.4. Note how the order changes from activity to activity.

TABLE 5.4

MAN-HOUR RANK ORDER OF DATA POINTS IN EACH ACTIVITY
(In Order of Decreasing Man-Hours)

Analysis & Design					
Prior to Coding	Total	Coding	Test	Total Coding and Test	Error Rate
6	6	6	4	6	1
7	7	7	2	4	6
3	2	2	1	7	7
2	3	1	6	2	3
5	4	4	7	1	2
4	5	5	3	5	?
1	1	3	5	3	?

Data point 1 is lowest for analysis and design, in the middle for programming and testing, and highest for error rate. Data point 2 is about in the middle of the programs except for testing, where it is high, and error rate, where it is low. Data point 3 is in the middle for analysis and design, low for coding and test, and also low for error rate. Data point 4 is low for analysis and design and for coding, and highest for testing. Data point 5 is low for everything. Data points 6 and 7 are high for analysis and design and for coding, medium for testing, and high for errors.

From this simplified comparison, it is obvious that the man-hour requirements are not consistent across activities (as would be the case if size were the only driving parameter) and that there are apparently trade-offs. Therefore, utilizing the figures, we tried to qualitatively evaluate the hypotheses posed on page 5-20.

In order to test the hypotheses, however, it was necessary to establish reference lines that could be used to distinguish increases and decreases in the variables to be tested. These reference lines should represent an ideal allocation of resources between the activities so that departures from the ideal can be identified.

The nominal level of analysis and design hours as a function of program size was determined by calculating a regression line (forced to go through the origin) for the total analysis and design time (Fig. 5.4). Differences between specific data points for time expended before and after the start of coding were assumed to indicate departures from the norm for a given program size.

In Fig. 5.7, showing total programming and testing time, the nominal level was taken to be the regression line determined for the four data points for which no design changes (ECPs) were reported during development (numbers 2, 3, 6, and 7).

Comparing Figs. 5.3 and 5.4 indicates that four data points (1, 2, 4, 7) represent system developments in which the analysis and design activities continued after the start of the coding activities. Figure 5.7 shows that data points 1 and 2 lie on the nominal programming and testing line while 4 and 7 are below it. According to the first hypothesis (p. 5-20), then, all four points should show relatively high error rates. Figure 5.8 indicates that point 1 is high, error data for 4 is not available, and 2 and 7 are low--which does not support the hypothesis.

Hypothesis 2 holds that increasing the investment in analysis and design decreases programming and testing time or operational errors. (Obviously this can only be true to a point. After the problem has been properly defined and a detailed design completed, additional analysis and design hours are a waste.)

Figure 5.4 shows two data points with higher investments in analysis and design (3, 6) and two with less (4, 5). Of these, 3 has a nominal programming and testing time, 6 is high, 4 is slightly low, and 5 very low. The hypothesis would indicate that the error rates for 3 and 6 should be nominal or less, and those for 4 and 5 should be higher than the norm. The error data supports the second hypothesis for points 3 and 6; error data for points 4 and 5 are missing.

Thus this analysis has not in general supported the hypotheses. However, note how this form of investigation is dependent on the selection of ideal resource allocation lines. Suppose, for example, that the ideal expenditure of coding and testing man-hours was that shown by the "alternative" line in Fig. 5.7. In that case, point 3 could be explained as being low due to "overdesign;" points 2, 4, and 7 as being high due to beginning coding before the end of analysis and design; point 1 (and also 4) as being high due to design changes*; and point 4 as also being

*Points 2, 3, 6 and 7 had no design changes (Ref. Table 5.3 and Fig. 5.7).

high due to inadequate design effort. Only points 5 and 6 would not support the hypotheses advanced (or be explainable as responding to ECPs).* That is, only these points do not support the hypotheses that coding and testing man-hours (1) increase if ECPs are introduced during development, (2) increase if there is too little analysis and design or if coding is initiated before the completion of design, or (3) decrease if the project is "overdesigned."

In summary, however, we must allow that this analysis of the hypotheses using the small data set has not produced any conclusive results. A larger data set, on the other hand, would contribute in two ways to the analysis. First, it would establish ideal resource allocation lines more accurately, a key to the trade-off analysis. Second, more data would allow stratification into different populations, so that the many effects that are operating simultaneously can be separated. Then, within these strata, the trade-offs should become clear.

Even so, the graphical approach is of limited value, leading to explicit hypothesis formulation as opposed to the establishment of an explicit cost estimating relationship. A more quantitative approach is developed in the next section which demonstrates the potential of analyzing trade-offs between activities.

5.3 QUANTITATIVE TECHNIQUES TO ESTABLISH TRADE-OFFS BETWEEN ACTIVITIES

5.3.1 Establishing the Relationships

To demonstrate this technique we will develop a relationship between the resources devoted to analysis and design and the resources

*These points are rather anomalous at any rate. Point 5 is PARMIS itself, which apparently was done on a stringent budget by experts. Point 6, on the other hand, appears to have had generous manpower allocations throughout the life cycle.

required to complete the programming and testing activities. The working hypothesis is that as a system is more completely described and more time is available to work out details, fewer revisions to developed code should be required. Furthermore, a complete design should allow the programmer to proceed more quickly. Personal experience has indicated that when very little analysis and design work is done, the programmer spends a significant part of his time completing the design instead of actually developing the system routines.

Therefore we hypothesize that, as the amount of man-hours devoted to analysis and design increases, the amount of time required for programming and testing decreases. In order to satisfy the condition that any valid relationship would be asymptotic to the ordinate and abscissa and to keep the mathematics simple, only relationships of the form:

$$MM_{PT} = a(MM_{AD})^{-b} \tag{5.1}$$

are considered, where MM_{PT} is man-months of programming and testing and MM_{AD} is man-months of analysis and design.

The trade-off between analysis and design and programming and testing is only really valid for a given program. In order to develop the relationship across a number of programs, descriptions of the other program differences must be included in the equation form. That is,

$$MM_{PT} = a(MM_{AD})^{-b} \times f(\text{Program Descriptors}) \tag{5.2}$$

Subsequent studies may indicate that the program descriptors must be multi-dimensional and that f is a complicated expression. For now we will use lines of source code as a descriptor and hypothesize the relationship to be

$$MM_{PT} = a(MM_{AD})^{-b}(LS)^{c} \tag{5.3}$$

where LS = lines of source code.

With a large data base, a regression analysis could easily be run to establish the relationship. However, we are working here with only five data points (points 5 and 6 from Sec. 5.2 have been excluded for the reasons cited in that section). A regression analysis on so few points could not sort out the effects of the hypothesized trade-off from those of program size, LS . Figure 5.9 is a sketch of the situation we are faced with. With only a few data points, the dependence on size, shown dashed, completely obscures the trade-off we are interested in examining, shown by the solid curves.

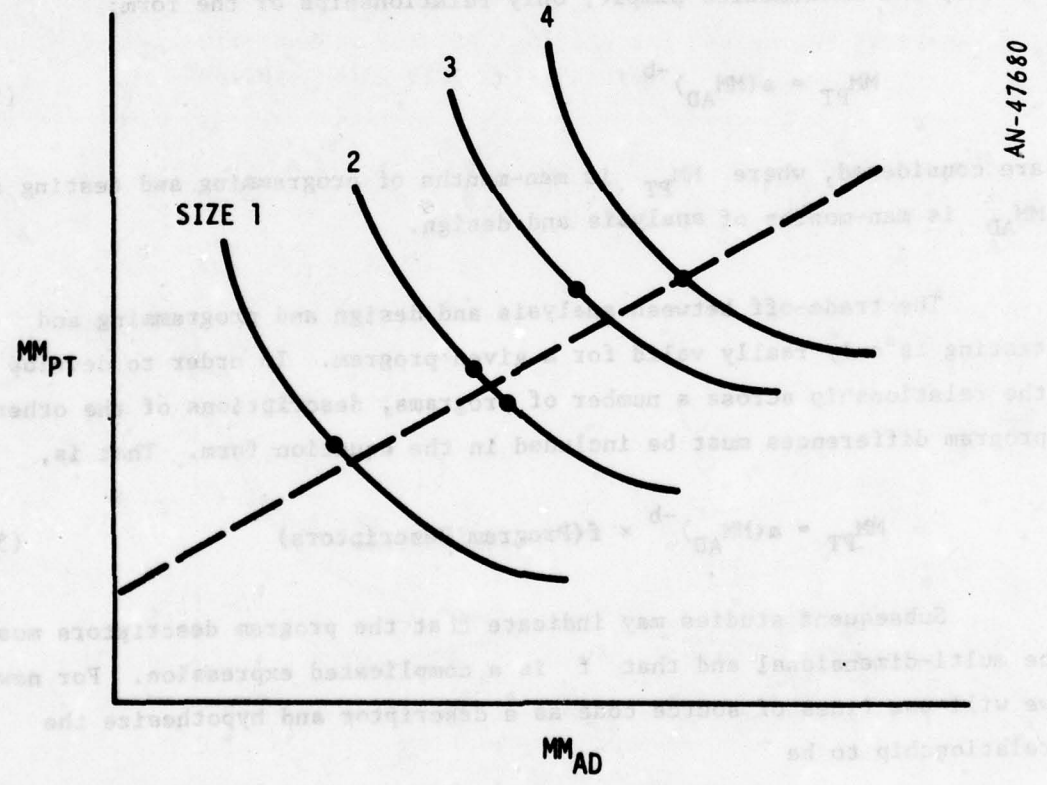


Figure 5.9. Trade-Off Curves and Size Parameter for Small Data Base

For this illustrative exercise, therefore, we simply postulate a plausible value for the exponent b, namely $b = 0.5$, and then evaluate the remaining parameters a and c by regression.* The result of the regression analysis is

$$MM_{PT} = 15.3(LS)^{0.753}(MM_{AD})^{-0.5} \qquad (5.4)$$

This relationship is graphed, with the five data points, in Fig. 5.10.

5.3.2 Optimum Allocation of Resources

The existence of a trade-off in man-hours between two activities of the development cycle implies the existence of optimum values for the man-hour variables. In this section we will solve for those optimum values.

The form of Eq. 5.4 is such that as analysis and design man-hours increase for a given program size, coding and testing man-hours decrease. The total man-hours (and hence cost) will at first decrease and then increase, as shown in Fig. 5.11. We want to locate the minimum of the total cost curve. The derivation follows.

---

*As a check we tried $b = 0.2$ and $b = 0.8$. Neither made much change in the computed value of c. Also, if we define

$$R^2 = 1 - (\text{unexplained variation})^2/(\text{total variation})^2$$

then Eq. 5.4 has $R^2 = .824$. This definition of R^2 is calculated on the original data set rather than the form of the equation used in the regression model. As such it can be directly compared to other model forms and avoids the problems of fit indexes (of which R^2 is one) discussed in Ref. 24. Although a good R^2 is not sufficient to prove the model, it does show that the data is consistent with the model.

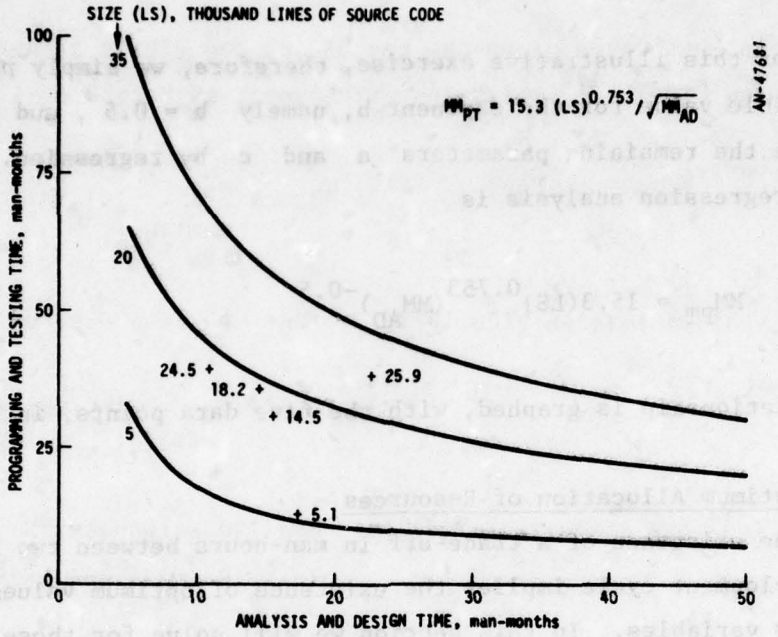


Figure 5.10. Relationship Between Analysis and Design and Programming and Testing, Using Five Data Points

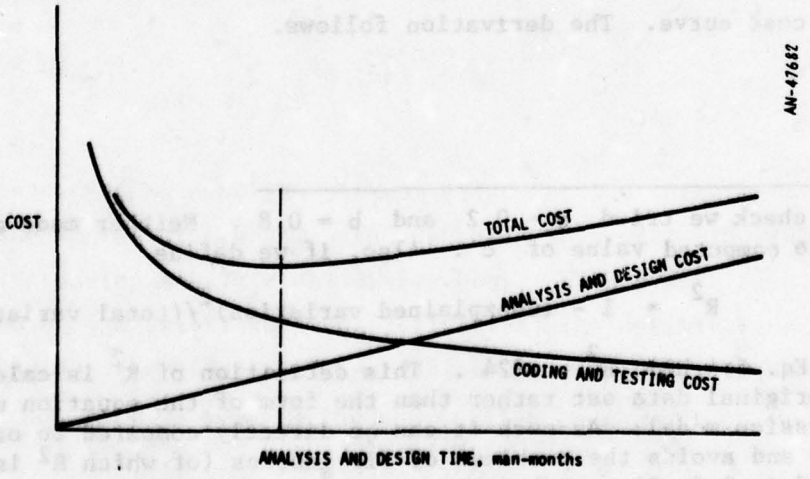


Figure 5.11. Behavior of Total Cost With Design/Coding Trade-off

Let the cost of analysis and design be given by

$$C_{AD} = C'_{AD} MM_{AD}$$

where C'_{AD} = cost per man-hour of analysis and design. Similarly, let the cost of programming and testing be given by

$$C_{PT} = C'_{PT} MM_{PT}$$

Now total cost is given by

$$C = C_{AD} + C_{PT}$$

or, substituting the previous relations and the trade-off equation (Eq. 5.4):

$$C = C'_{AD} MM_{AD} + C'_{PT} MM_{PT}$$

$$C = C'_{AD} MM_{AD} + C'_{PT} a(LS)^c / \sqrt{MM_{AD}}$$

Now the optimum is found by differentiating total cost with respect to man-months of analysis and setting the result equal to zero:

$$\frac{\partial C}{\partial MM_{AD}} = C'_{AD} - a\, C'_{PT} (LS)^c \frac{1}{2} (MM_{AD})^{-3/2} = 0$$

or

$$MM_{AD}\Big|_{\text{optimum}} = 0.63 \left[a \frac{C'_{PT}}{C'_{AD}} \right]^{2/3} (LS)^{2c/3}$$

Thus the optimum number of analysis and design man-months has been found as a function of size (in thousands of lines of source code) and the ratio of programming and testing unit cost to analysis and design unit cost.

Values of $7,020 and $6,560 per man-month were computed for C'_{PT} and C'_{AD} from cost data reported by Wolverton.[19*] These values give

$$MM_{AD}\Big|_{optimum} = 0.66\ a^{2/3}\ (LS)^{2c/3}$$

or, with the values of a and c previously determined (Eq. 5.4),

$$MM_{AD}\Big|_{optimum} = 4.07\ (LS)^{0.5}$$

The corresponding optimum time for programming and testing can now be calculated from Eq. 5.4:

$$MM_{PT}\Big|_{optimum} = a(LS)^{c} / \sqrt{MM_{AD}\Big|_{optimum}}$$

$$= 1.23\ a^{2/3}\ (LS)^{2c/3}$$

or, substituting the values of a and c :

$$MM_{PT}\Big|_{optimum} = 7.58\ (LS)^{0.5}$$

The expressions give the optimum value for the investment considering only one degree of freedom: the decrease in programming and testing cost for increasing analysis and design cost. With a more complete description of the relations among activities, the optimization would have to consider the effects of increasing the investment in analysis and design not only on programming and testing costs, but on such

---

*These values include both personnel and computer costs.

life-cycle parameters as total cost, cost of errors, and effects on the development schedule. The inclusion of these factors in future studies offers the promise of giving program managers quantitative measures for making decisions during software development.

It is useful to compare the predicted man-months for the optimum with actual relationships derived in a prior study.[22] Figures 5.12 and 5.13 compare the predictions with data on some large software developments compiled for NASA.

The expression for the optimum relationships derived in this section has markedly different results from the NASA data, particularly for programs larger than about 25,000 lines of source code. The dependence on $LS^{0.5}$ --a square root relationship--implies economies of scale, which have not been evident in the historical data. This is probably a direct consequence of the sample size and the range of the data within the sample (maximum equals 36,000 new lines of source code). However, if the square root rule holds with a larger data base, then the message is quite different. Historically, we have not been achieving the economies implied, due perhaps to poor allocation of resources.

5.3.3 Comparison of the Optimum Hours with the 40-20-40 Rule

There are many references in the literature to the distribution of resources among design (and analysis), coding, and testing. Although there is considerable variation among the published distributions,[22] many writers cite an average distribution of resources of 40% analysis and design, 20% coding, and 40% testing. It is interesting to develop a comparable distribution using the optimum values developed in the preceding sections.

The ratio of optimum coding and testing time to optimum design and analysis time can be calculated. Surprisingly, the answer is independent of size (lines of source code):

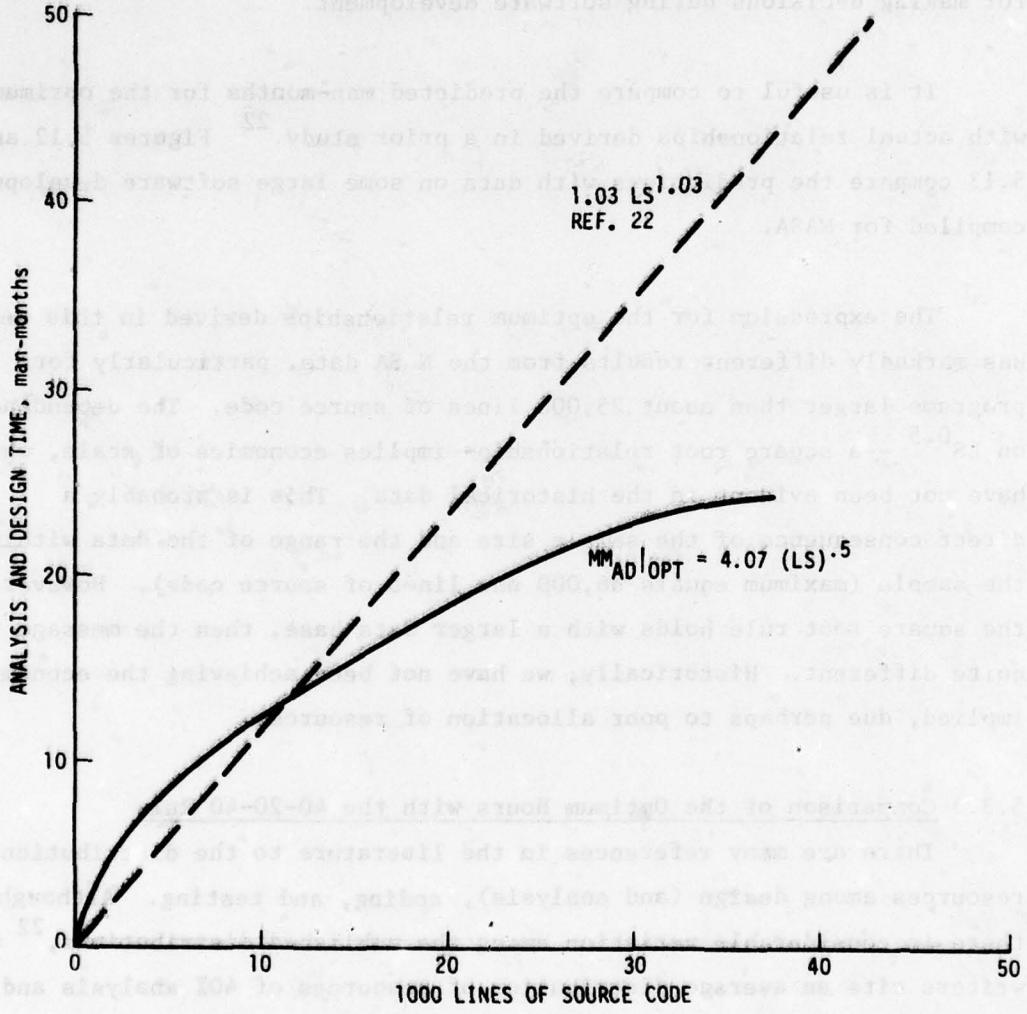


Figure 5.12. Comparison of "Optimum" Analysis and Design Time with Project Data

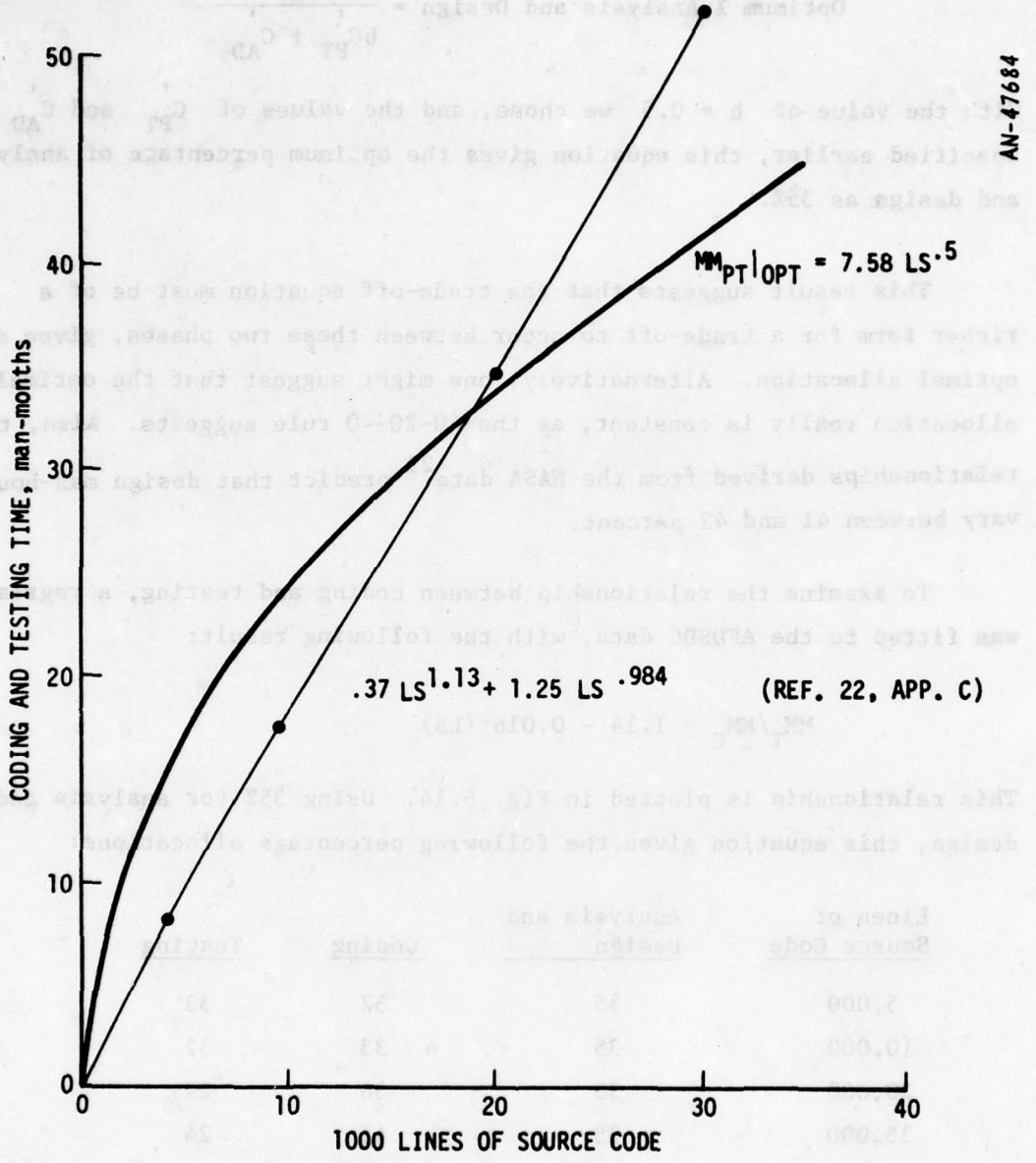


Figure 5.13. Comparison of "Optimum" Coding and Testing Time With Project Data

$$\text{Optimum \% Analysis and Design} = \frac{bC'_{PT}}{bC'_{PT} + C'_{AD}}$$

With the value of $b = 0.5$ we chose, and the values of C'_{PT} and C'_{AD} specified earlier, this equation gives the optimum percentage of analysis and design as 35%.

This result suggests that the trade-off equation must be of a richer form for a trade-off to occur between these two phases, given an optimal allocation. Alternatively, one might suggest that the optimal allocation really is constant, as the 40-20-40 rule suggests. Also, the relationships derived from the NASA data[22] predict that design man-hours vary between 41 and 42 percent.

To examine the relationship between coding and testing, a regression was fitted to the AFDSDC data, with the following result:

$$MM_T/MM_C = 1.14 - 0.016\ (LS)$$

This relationship is plotted in Fig. 5.14. Using 35% for analysis and design, this equation gives the following percentage allocations:

Lines of Source Code	Analysis and Design	Coding	Testing
5,000	35	32	33
10,000	35	33	32
20,000	35	36	29
35,000	35	41	24
50,000	35	48	17

The division between coding and testing resources changes significantly over the range. Not only do none of the results match the 40-20-40 rule well, but there is a significant change in the distribution between coding

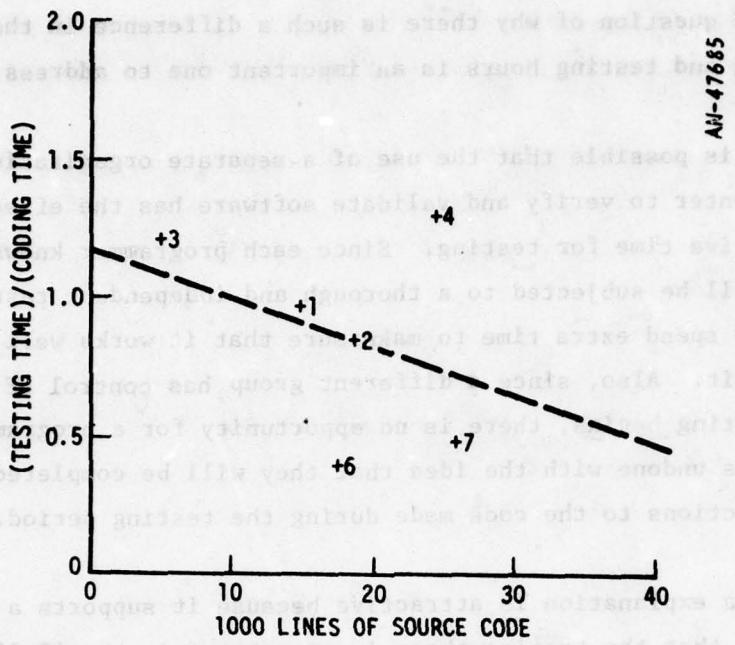


Figure 5.14 Testing and Coding Relationship

and testing hours as program size increases. The results indicate that the emphasis on testing becomes less as program size increases.

Again analyzing the trendline data from our previous NASA study,[22] we see an inverse relationship between man-months of coding and testing. In particular,

$$\frac{MM_T}{MM_C} = \frac{1.25\ (LS)^{0.934}}{0.373\ (LS)^{1.13}} = 3.35\ (LS)^{-0.196}$$

The direction of this relationship is the same as before; but over the range of interest, coding varies between 17 and 23 percent, and testing between 42 and 35 percent. Thus, the percentage allocation is quite different, and much more in line with the 40-20-40 rule.

The question of why there is such a difference in the distribution of coding and testing hours is an important one to address at this point.

It is possible that the use of a separate organization at the Design Center to verify and validate software has the effect of decreasing the relative time for testing. Since each programmer knows that his system will be subjected to a thorough and independent test, he is more likely to spend extra time to make sure that it works well before he releases it. Also, since a different group has control of the programs after testing begins, there is no opportunity for a programmer to leave some items undone with the idea that they will be completed along with any corrections to the code made during the testing period.

This explanation is attractive because it supports a long-held suspicion that the testing phase is overstated in the 40-20-40 rule. More importantly, it is possible that the Design Center's distribution of effort between programming and testing represents a truer description of the relative efforts. Hours are treated separately, and there is no pressure to declare the program to be in internal integration and testing at one point in time. Thus testing of already programmed code and programming of new code can go on concurrently. In effect, the "moving milestone" of Fig. 2.3 has been captured in the PARMIS data base.

On the other hand, data sources using a fixed milestone approach may well allow man-hours to be recorded against testing once some portion of the code is in testing. If this is the case, then initial coding hours are being reported against testing, a practice which certainly distorts the data. The 40-20-40 rule may be a consequence of this reporting distortion.

Of course, another possibility is that the PARMIS data base is just too small to make these particular estimates. Only a larger data base will help alleviate that particular concern.

5.4 CONCLUSIONS

Studying the relations among project phases has made it possible to develop the form of relationships that would be useful for software project management. Results are only illustrative, due to the small data base, but the technique has been demonstrated and the potential is promising. Using these types of relationships for the optimum division of resources between analysis and design and programming and testing, it should be possible to lower development costs and to have a better basis for evaluating project proposals and for making decisions during project implementation. Extension of the method to include other software dimensions such as reliability should make the study of relationships even more valuable.

The results derived from the Data Systems Design Center suggest that there is reason to question the 40-20-40 rule of thumb for distribution of resources. This finding may have significant impact on the future planning of software development projects. Furthermore, the findings support previous results[22] that indicate that the relative emphasis on testing decreases as project size increases. Realizing this expectation could lead to sizable savings.

6 QUANTITATIVE RESULTS

6.1 INTRODUCTION

In this section we report on the additional analyses that were made, using data bases other than PARMIS.

We were able to draw on some data from software contractors and to obtain some high-quality information from one Program Office. Our principal sources of data were the ADP Resource Estimating Procedures (ADPREP) study,[12] the SAMSO Program Office that is responsible for the development of general-purpose support software for the Satellite Control Facility, some data from another GRC software project, and some published data dealing with software reliability. These sources of data are described in detail in Sec. 6.2. Sections 6.3 to 6.5 discuss how the data were used to test some of the hypotheses discussed in Sec. 3, and the results of these tasks. The importance of some explanatory variables is covered in Sec. 6.3; relationships between alternative measures of software products in Sec. 6.4; and characteristics of the Maintenance activity in Sec. 6.5.

6.2 DATA SOURCES

6.2.1 ADPREP

ADPREP was a study performed by Planning Research Corporation for the Army in 1975. The study reported on 38 data processing systems: 18 developed by the Air Force and 20 by the Army. These systems included both business-oriented (ADP) and other programs, although ADP programs dominate.

The ADPREP study included interviews and reviews of project data to produce summaries of each project's history and cost. The data reported addresses development, operation, and maintenance of the software systems. Some of the types of information reported are: (1) maintenance data, including staffing, computer usage, and types of improvements made;

(2) program size, including counts of both source and object instructions; (3) the number of changes in requirements during development; (4) the project schedule; (5) the staffing of the development phase, broken down by types of personnel; (6) effort, also broken down by types of personnel; and (7) computer usage, broken down by major project phase (development, operation, and maintenance).

The major limitation of the ADPREP data is that effort and computer usage are aggregated over the whole project and not broken down by development phase (analysis, design, coding, test), and therefore cannot be used in any but the most aggregated of analyses.

6.2.2 AOES Data

The Advanced Orbital Ephemeris Subsystem (AOES) is a large ground-based C^3I system that is used to support satellite systems. The AOES data describes the maintenance and operation of more than 400 programs written and maintained by two contractors. The types of programs include compilers, operating systems, data reduction and presentation utilities, and orbit planning and analysis utilities.

The AOES is maintained by two associate contractors and an integration contractor. The associate contractors are responsible for the maintenance and improvement of separate portions of the software; roughly, one portion is the operating system and the other is the applications programs used to maintain satellite ephemerides. The role of the integration contractor is to review the products of the associate contractors and to perform the system-level testing. A very important additional function of the integration contractor is to maintain configuration control over the versions or models of the software that are released for operational use by the Air Force.

To achieve configuration control, the integration contractor maintains a data base of the characteristics of the various software packages of the system (program components) and records describing their modifications.

Unfortunately, that data cannot be correlated with the costs of the component activities. However, it was possible to use the data to investigate the properties of the operations and support phase of the software life cycle.

The data we collected describes four revisions or "releases" of the system, and the maintenance histories of those revisions once they became operational. For each program of each release of the system, we collected (1) the delivered size, (2) the number of problems reported and resolved, (3) the size of the Part II Specifications, (4) the design changes incorporated with that release, and (5) the system integration tests of the design changes. In addition, we collected data on the man-months of development and the man-months of maintenance effort, and the schedules for developing, testing, and operating the system releases.

The size of the software was measured in terms of twelve characteristics, of which we found lines of source code and lines of object code to be most useful. The number of design changes was determined by the number of Design Change Requests approved for implementation in the release being considered. The number of problems corrected by the release was determined by counting the number of Discrepancy Report Forms (DRFs) closed by the start of integration testing. The number of problems reported with the integration testing and use of the release was obtained by counting the number of DRFs opened (even if subsequently closed) after the start of integration testing.

The applications and analyses of the data were limited by the difficulty of relating the effort expended by contractors to the changes in the software. Specifically, only the final delivered sizes of programs were reported; how much of the programs was new in each release was unknown. Maintenance activities of the contractors began with the start of system-level tests rather than operational use of the system, and

were reported in aggregate form. More importantly, revision and maintenance often took place concurrently and were not reported separately.

6.2.3 GRC Internal Data

This collection of data was published in an earlier GRC study for NASA.[22] It consists of information obtained from the open literature, reports on and interviews with specific NASA projects, and data provided by Boeing Aerospace Company. The data generally reports on the total dollars or effort required for a software development project, whose size is reported in object-language instructions. The dollar costs were converted to effort by first inflating the dollar values to a common year (1974) and then converting to man-years with a factor of $50,000 per man-year.

The data collected also reported the allocation of total costs to three development phases: (1) design, (2) coding, and (3) test. These phases are not identical with those defined in Sec. 2. For example, "design" includes what we have called "analysis".

6.2.4 Consistency of the Data Sources

Before using this data to test hypotheses, we will examine all of the data used in this study and compare the ADP applications with the aerospace applications.

Of the data we have used, some describes software written for aerospace defense applications, and some describes business or "ADP" systems. One of the first considerations in using this mixture of data is whether it is reasonable to use data on ADP systems to study and predict the costs of defense systems.* Costs in some phases of a software

---

*This is especially important with the reliance on PARMIS data in the previous section.

development obviously depend upon the application. This dependence was noted throughout Sec. 3, and was accommodated by stratifying the cost relationships by software type.

The ADP and aerospace software systems considered in this report have similar life cycles that break down into the phases discussed in Sec. 2. We believe that the general forms of the relationships describing the costs of individual phases are independent of the application, but that the values of the coefficients used in the relationships depend on application.

Figure 6.1 illustrates how the various data sources compare with one another in terms of size and development effort. The defense (aerospace) programs examined in the NASA study cited in Sec. 6.2.3 are plotted as dots; the ADPREP data, on a mixture of "scientific" and "commercial" programs, are plotted as squares. The PARMIS data are plotted as triangles. The same general trends are evident for each of the three groups of data: development effort increases something like linearly with computer program size, but the variance is extremely large.

Generally, the defense-system programs are more costly. However, both "commercial" and defense-system programs exhibit similar trends, with somewhat more dispersion in the "commercial" data. We believe that the programming processes used to develop "commercial" and defense-system programs are similar, and that the higher cost of defense-system programs can be explained by their complexity; this is reflected in coefficients of cost estimating relationships. We believe that the functional relationships between the components of the development activity and the variables that explain effort are similar for either type of system. Thus, valuable insights can be obtained from the study of "commercial" or "ADP" developments. They also offer the advantage of clearly distinguishing the costs of software from other system costs.

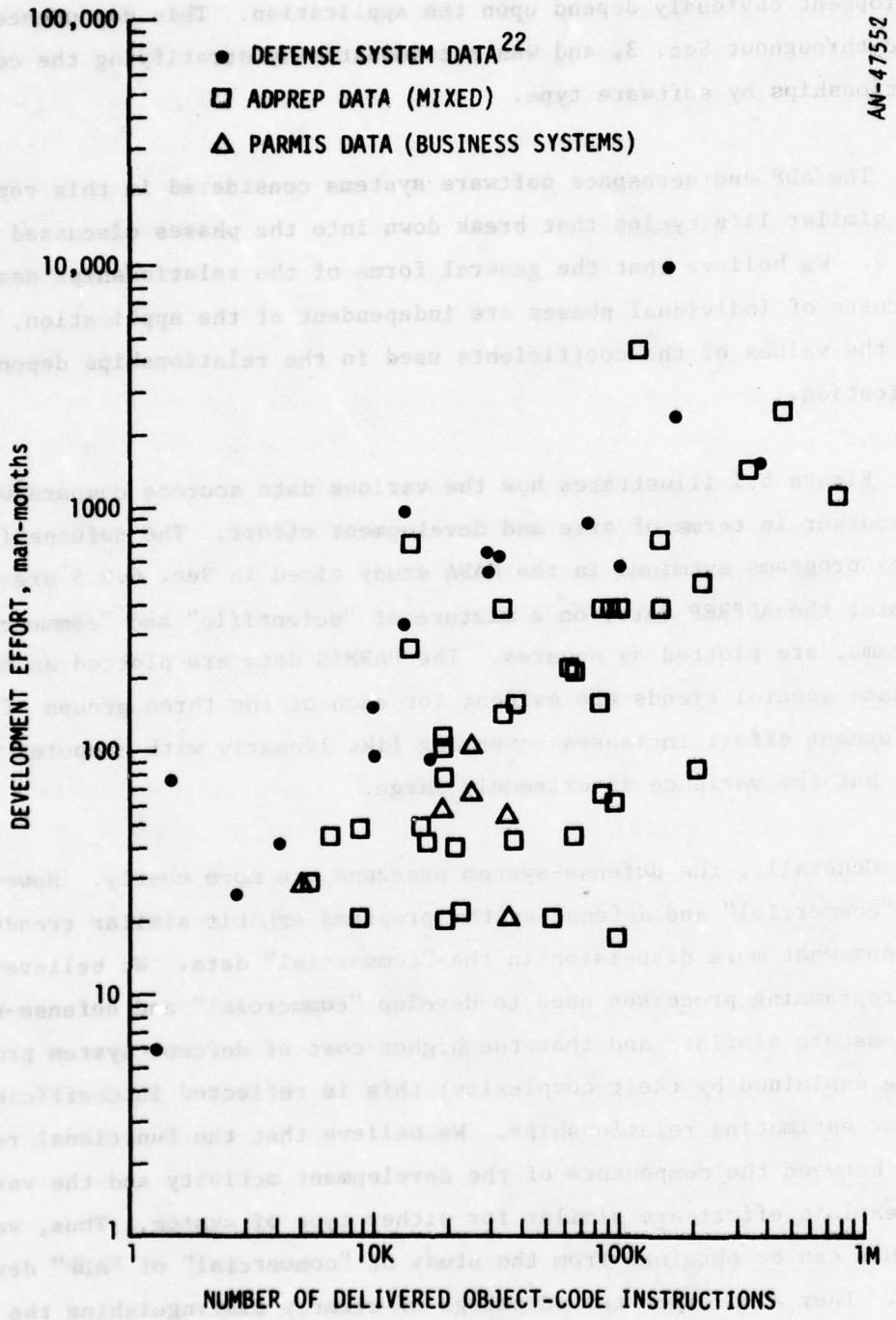


Figure 6.1. Size and Effort in Three Data Bases

Several comments need to be made about the quality and nature of these data collections. These observations tend to lessen the apparent differences between the unit costs of the defense system or "scientific" data points and those of the "business" or "ADP" data points plotted in Fig. 6.1. The PARMIS and ADPREP data points represent software, mostly of the "ADP" type, that was generally developed by the Army or the Air Force. Almost all of the defense-system data points represent software developed by private companies. It is not clear that the services report clerical and support man-hours against the development in the same manner that a private contractor would. If they do not, the ADP (Government) applications should be more expensive than they were reported to be.

Secondly, some of the defense-system data points were derived from total dollar costs inflated to 1974 dollars, assuming a $50,000 man-year. The composition of the dollars reported is not always known. Some reports may contain computer costs or hardware integration and test costs which would tend to exaggerate unit costs. Thus, there is some possibility that the trend line for the defense-system data should be moved closer to the ADP trend. These conjectures have not been confirmed, as all the details of the data reported were not available.

The data described in the preceding paragraphs were used to obtain results that demonstrate some of the important explanatory variables of software cost (Sec. 6.3), relationships between alternative measures of the software product (Sec. 6.4), and characteristics of the maintenance activity of software (Sec. 6.5).

6.3 TESTING EXPLANATORY VARIABLES FOR SOFTWARE COSTS

The bulk of this data is not detailed enough to test hypotheses that relate explanatory variables to the resources consumed by the specific phases of a software development, or to their relationships, which we believe to be important in explaining variations in the life-cycle costs of systems.

The ADPREP, AOES, PARMIS, and GRC data were used to determine the significance of five effects that were hypothesized (in Sec. 3) to be important in explaining the development and maintenance costs of software: (1) size of the product, (2) the type of problem being solved, (3) the programming language used in the software development, (4) constraints imposed by limited hardware resources, and (5) changes in requirements during the development of the software.

6.3.1 Product Size

Throughout Sec. 3, the hypothesized relationships defining software life-cycle costs employ an independent variable that is related to the size of the delivered software. In most cases, the estimate of size is taken as object-code instructions, so that any effect of programming language is minimized. It is therefore important that the data available to this study demonstrate a clear relationship between the size of the software and the effort required to develop the software. Since man-hours were, in general, not broken down to the phase level, this aspect of the study considered total development effort as a function of delivered software size measured in object instructions for most of the data points to be used. These data include ADPREP, GRC, and PARMIS. The AOES data was omitted, since the size of the delivered software could not be quantified and related to effort.

As Fig. 6.1 indicates, there is an apparent relationship between size of the software and the effort required to develop that software. However, the data varies so much that a relationship based solely on such a trend is not useful as a cost estimator. As Sec. 4 indicated, there is evidence to support the theory that part of the variation is explained by trade-offs between the phases required to produce the software. In addition, it is believed that there are other variables that will explain the differences between the samples. The first of these is the type of problem. The data available is meager, and thus it is not possible to define the types of software systems that are in the sample space.

However, a gross categorization of the sample space is possible: defense system and "commercial."* Analysis of the data shows that the trend relating size to effort suggests that defense system programs are produced at the average rate of 10 man-months per 1000 object code instructions in the final program (approximately $50 per instruction). The trend for the "commercial" programs indicates that they are produced at an average rate of 3 man-months per 1000 instructions (approximately $17 per instruction). Roughly, defense system programs are three times as expensive to develop.

It is important that the dollars-per-instruction figures not be taken literally. As noted earlier, there may be important differences in what is reported as development effort that would invalidate these figures. The data are presented only to show that a relationship between size and effort exists, and that the values are consistent with known rules of thumb.

6.3.2 Programming Language

Section 3 discussed the role of programming language extensively. It was asserted that programming language should have little, if any, effect on the cost of phases such as design and analysis, since their products are independent of the language. However, programming language should have a dramatic effect on the effort required to code and test software. One property of high-level languages is a large "code expansion factor": one statement in the high-level language expands into many object-language instructions. This would suggest that a high-level language is more labor-efficient during the coding activities. In addition, programs written in high-level languages should be easier to test and maintain, since they express the design implementation in terms more compatible with the actual problem being solved. (For instance,

---

*Defense system applications are C^3 or avionics (real time), while "commercial" applications are "ADP" or "business" programs that run in either batch or interactive modes.

the concept of an array is used to represent a vector in a high-level language as compared with the mere consecutive words of memory used in an assembly language.)

Again, sufficiently detailed data was not available to determine the validity of the hypothesized relationships discussed in Sec. 3. Since programming language permeates the entire discussion of Sec. 3, it seems reasonable to test for the significance of language if only in a limited form. The ADPREP data contains information about the programming language employed. The data is not broken down to the development phase level. It was decided that regression analysis could be used to test for the significance of this variable, using total development costs. What results is not a useful estimating relationship for total development effort, since the R^2 values are small; the T statistic for the independent variable (programming language) can provide some insight about the importance of programming language in explaining variations.*

The ADPREP data was analyzed from several points of view. The first analytic results (Fig. 6.2) show the extent to which variations in total development effort are explained by the programming language employed. The figure is the result of using the program BMDSP.[25] The regression analysis was performed on the logarithms of the variables so that a relationship of the form

---

*In this section we will be using the typical statistical measures associated with regression analysis to define the quality of the relationships. R^2 measures the goodness of fit, with values close to 1 showing a good relationship. F and T statistics measure the significance of the model in explaining the data; F compares the entire model to a model based on the data average, while T statistics measure the significance of each individual parameter. Significant values for F and T depend on the size of the sample, but in general, if F is greater than 6 and T greater than 2, you have a significant result. For further information consult Lindgren, Statistical Theory, MacMillan, 1962.

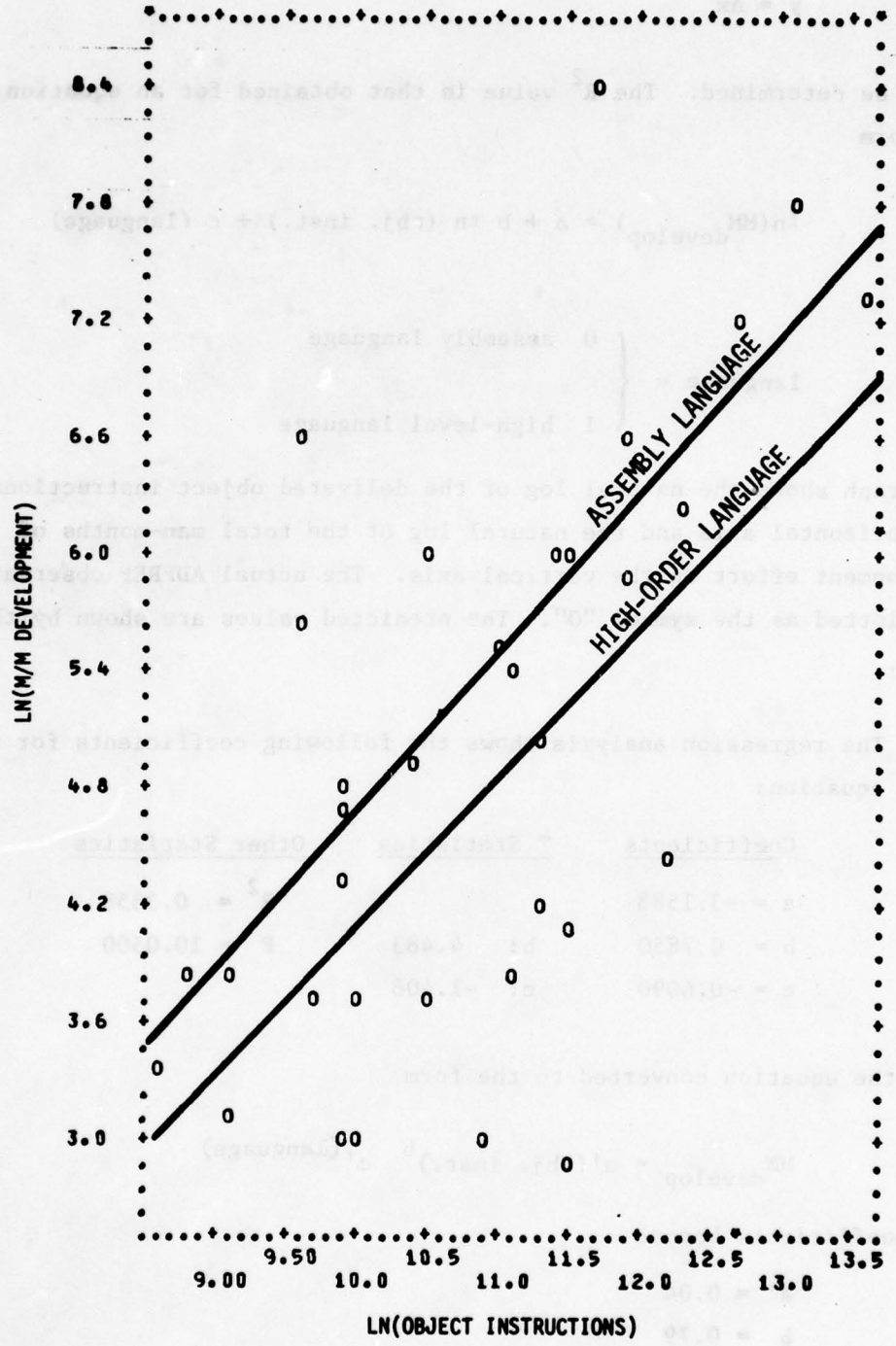


Figure 6.2. Development Man-Months Vs Object Instructions

$$y = ax^b$$

could be determined. The R^2 value is that obtained for an equation of the form

$$\ell n(MM_{develop}) = a + b\ \ell n\ (obj.\ inst.) + c\ (language)$$

where

$$language = \begin{cases} 0 & \text{assembly language} \\ 1 & \text{high-level language} \end{cases}$$

The graph shows the natural log of the delivered object instructions on the horizontal axis and the natural log of the total man-months of development effort on the vertical axis. The actual ADPREP observations are plotted as the symbol "O". The predicted values are shown by the lines.

The regression analysis shows the following coefficients for the above equation:

Coefficients	T Statistics	Other Statistics
a = -3.1588		R^2 = 0.3858
b = 0.7850	b: 4.483	F = 10.0500
c = -0.6090	c: -1.408	

With the equation converted to the form

$$MM_{develop} = a'(obj.\ inst.)^b\ \ c'^{(language)}$$

the coefficients become

$$a' = 0.04$$
$$b = 0.79$$
$$c' = 0.54$$

It should be noted that the T and R^2 statistics apply only to the natural logarithms used in the regression and not to these modified coefficients. Also while b is significant, c is only marginally so (confidence level 80%), so the relationship is not strong.

The next two illustrations show the same hypothesis examined in terms of the effort in particular skill categories. These results were obtained with a subset of the ADPREP data that reported effort by analysts and programmers. We first consider the analysts. The analyst makes a primary contribution during analysis and design activities of a project, although in many of the data points of the ADPREP sample, analysts were employed throughout the lifetime of the project to code and test the product. Thus, it is not always possible to associate the analyst's effort with a unique subset of the life-cycle activities of the ADPREP data. The equation used in the regression analysis shown in Fig. 6.3 is

$$\ell n(MM_{analysts}) = a + b\ \ell n\ (obj.\ inst.) + c\ (language)$$

The statistics of the regression are

Coefficients	T Statistics	Other Statistics
a = -1.7246		R^2 = 0.3142
b = 0.550	b: 1.472	F = 2.749
c = -1.407	c: -2.309	

Figure 6.4 shows the same analysis of the man-hours reported for programmers. Again, it is not possible to attribute programmer activities to a unique subset of the life-cycle activities. The equation used in the regression analysis shown in Fig. 6.4 is

$$\ell n(MM_{programmers}) = a + b\ \ell n\ (obj.\ inst.) + c\ (language)$$

The statistics of the regression are

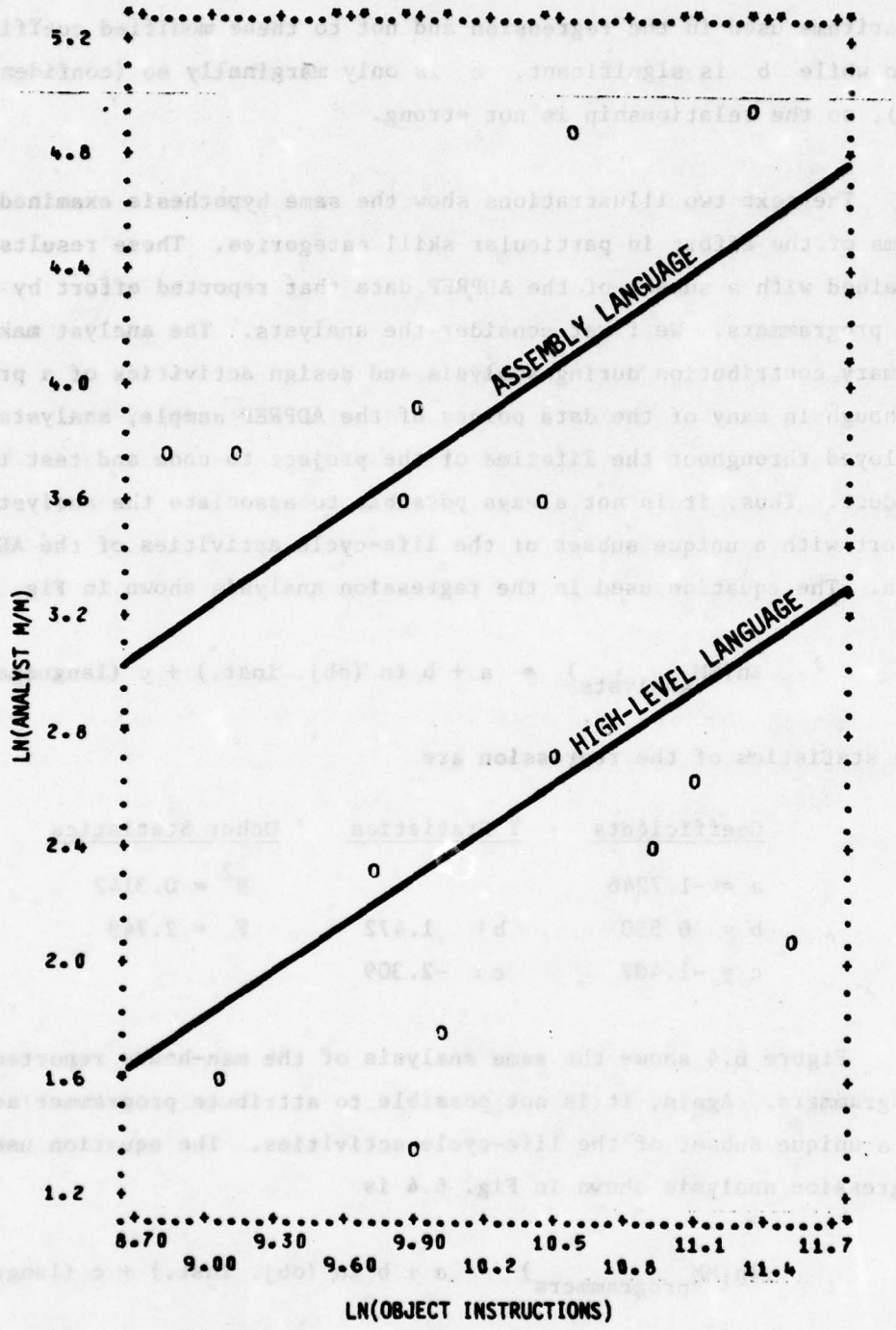


Figure 6.3. Analyst Man-Months Vs Object Instructions

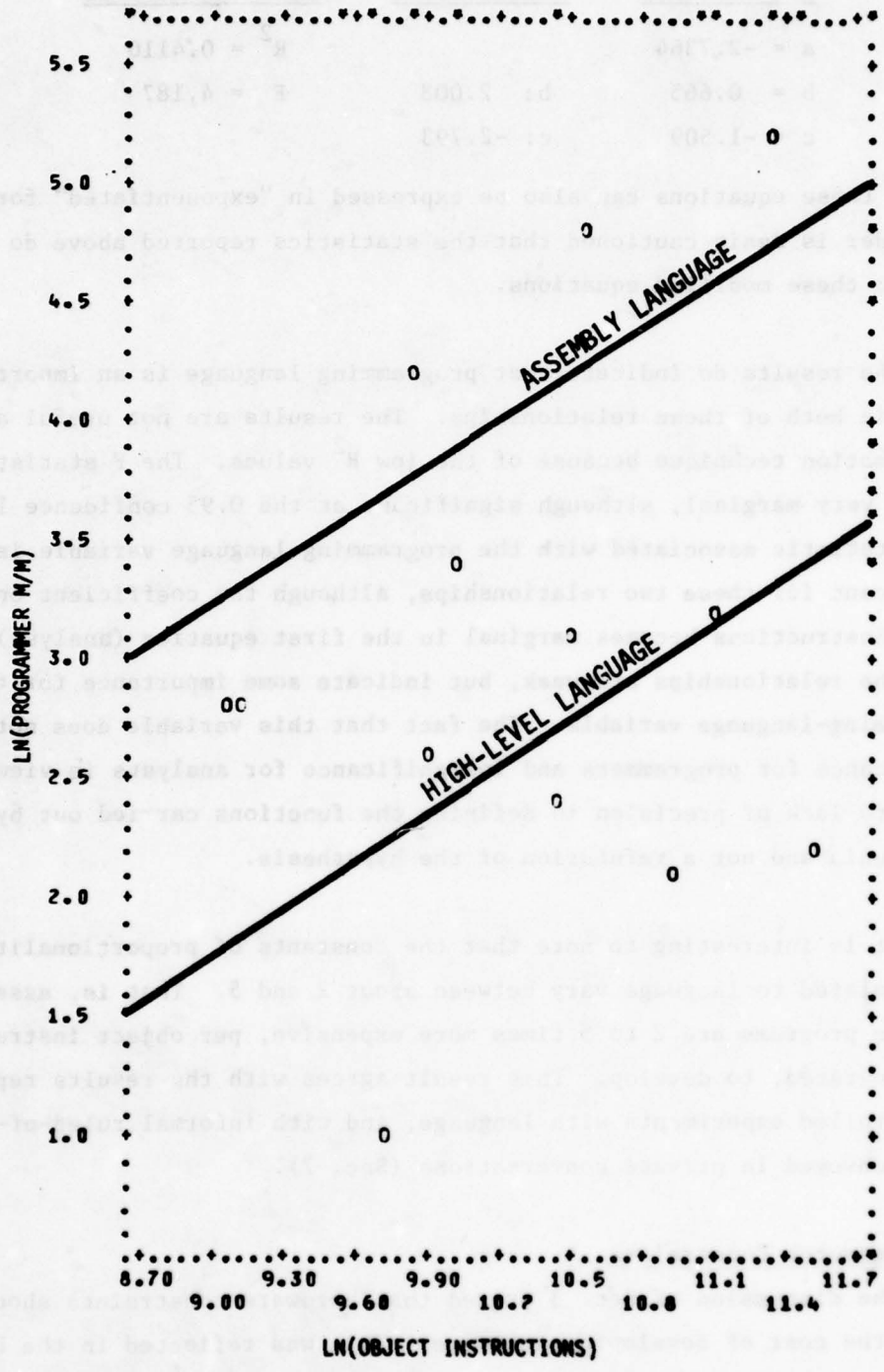


Figure 6.4 Programmer Man-Months Vs Object Instructions

Coefficients	T Statistics	Other Statistics
a = -2.7364		R^2 = 0.4110
b = 0.665	b: 2.008	F = 4.187
c = -1.509	c: -2.793	

Each of these equations can also be expressed in "exponentiated" form. The reader is again cautioned that the statistics reported above do not apply to these modified equations.

The results do indicate that programming language is an important factor in both of these relationships. The results are not useful as an estimation technique because of the low R^2 values. The F statistic is also very marginal, although significant at the 0.95 confidence level. The T statistic associated with the programming-language variable is also significant for these two relationships, although the coefficient on object instructions becomes marginal in the first equation (analyst). Hence the relationships are weak, but indicate some importance for the programming-language variable. The fact that this variable does not show significance for programmers and insignificance for analysts is viewed as due to lack of precision in defining the functions carried out by the individuals and not a refutation of the hypothesis.

It is interesting to note that the constants of proportionality (e^{-c}) related to language vary between about 2 and 5. That is, assembly-language programs are 2 to 5 times more expensive, per object instruction generated, to develop. This result agrees with the results reported in controlled experiments with language, and with informal rules-of-thumb conveyed in private conversations (Sec. 7).

6.3.3 Resource Constraints

The discussion of Sec. 3 argued that hardware constraints should affect the cost of developing software. This was reflected in the hypothesized estimating relationships for the coding and testing phases.

This effect has been previously observed and documented,[21] and has been incorporated in the results of many researchers. The most general form of this relationship shows development costs growing exponentially with increasing utilization of the hardware. Our examination of the literature showed that there were no attempts to quantify this relationship, so the hypotheses of Sec. 3 have simplified the relationship to a single factor representing the presence or absence of such a constraint, the constraint being assumed to be present if more than 95 percent of the computer's memory was being utilized.

No specific data on the costs of individual phases (as defined by the process model of Sec. 2) were available to test the specific hypotheses of Sec. 3. However, some data from an earlier GRC study[22] had been applied to estimate the importance of resource constraints. As discussed in Sec. 6.2.3, these earlier data were allocated to three phases (design, coding, and test).* This earlier study examined the effort required in each phase as a function of resource constraints.

The results obtained for each phase are shown in Table 6.1 and illustrated in Figs. 6.5 to 6.7. In each case, effort is approximately linear with size, and hardware constraints, when present, increase costs by a factor of 5 or more.** It is also interesting to note that the ratio of testing effort to coding effort gets smaller with increasing size.

The results differ somewhat from the hypotheses presented in Sec. 3. In that section, it was claimed that the hardware constraint would affect only the coding and testing costs. These results indicate that the constraint also drives design costs. It is recommended that this additional hypothesis not be added until the exact accounting principles

---

*Not the same phases as defined in Sec. 2.

**The factor of 5 is observed when converting the equations in Table 6.1 to a linear form; e.g., in the last equation $e^{1.640} = 5.16$.

TABLE 6.1

HARDWARE-CONSTRAINT EQUATIONS

$$\ell n\ (\text{M-Y design}) = -1.17 + 1.03\ [\ell n(\text{obj. inst.}/1000)] + 1.730\ X$$

$$\ell n\ (\text{M-Y coding}) = -2.3 + 1.13\ [\ell n(\text{obj. inst.}/1000)] + 2.120\ X$$

$$\ell n\ (\text{M-Y testing}) = -0.86 + 0.934[\ell n(\text{obj. inst.}/1000)] + 1.640\ X$$

where

$$X = \begin{cases} 0 & \text{if no hardware constraint} \\ 1 & \text{if more than 95\% of memory is utilized} \end{cases}$$

---

used to determine costs in that data base are understood and related to the phases of the process model used in this report.

The results indicate that the presence or absence of the hardware constraint is important in explaining variations in development costs. Its effect is clearly secondary to that of code size. The coefficients of X that prescribe the effect of the constraint on costs differ considerably from results obtained elsewhere,* which indicate that constrained software is approximately twice as expensive, rather than five times, as indicated in Table 6.1. Since only three of the data points considered in these relationships represent constrained developments, differences are not surprising, and one should not take the actual factor of five literally.

*Doty Associates claimed a significantly smaller impact at the second Technical Direction Meeting for this project.

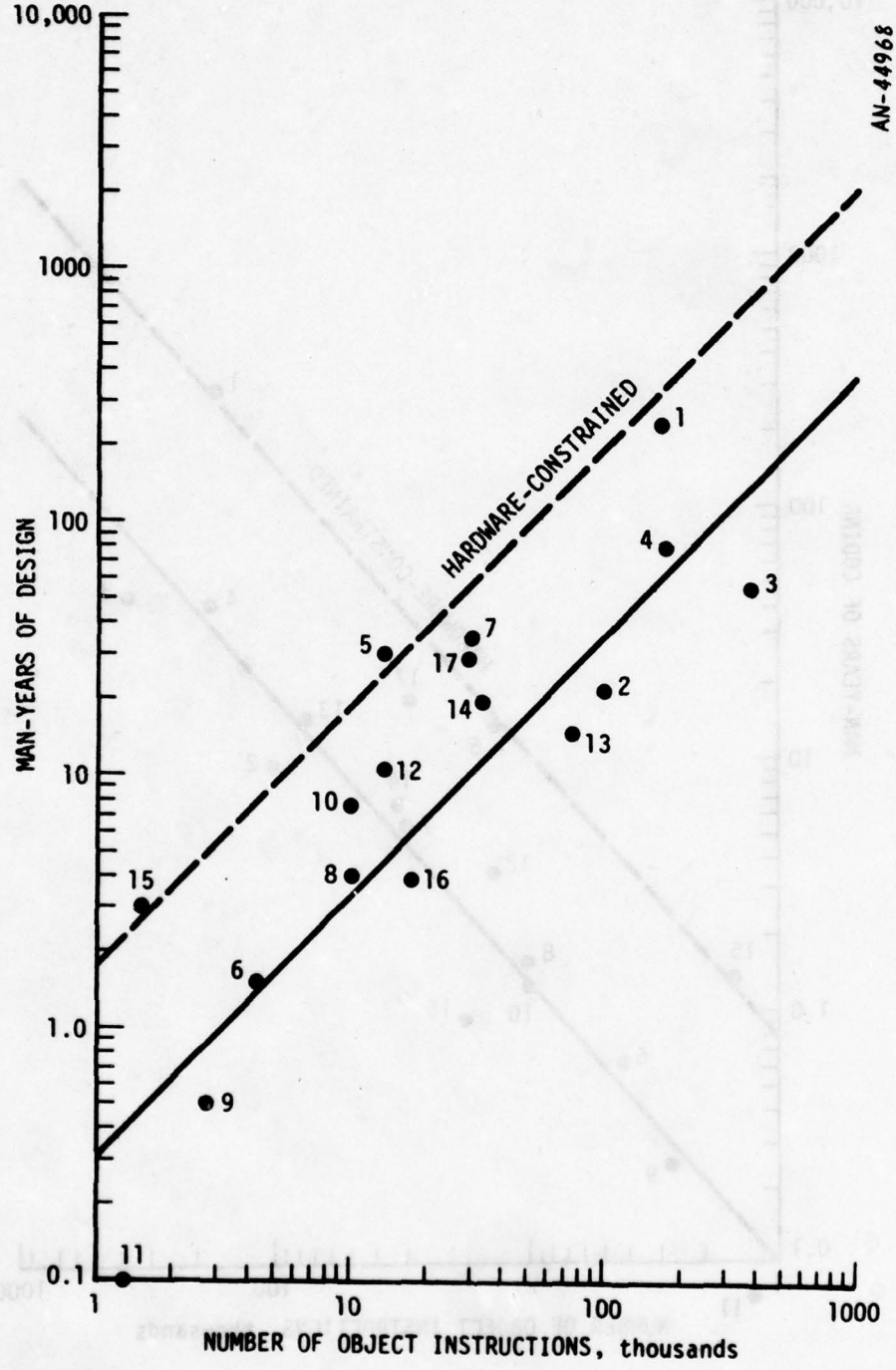


Figure 6.5. Effect of Hardware Constraint: Design

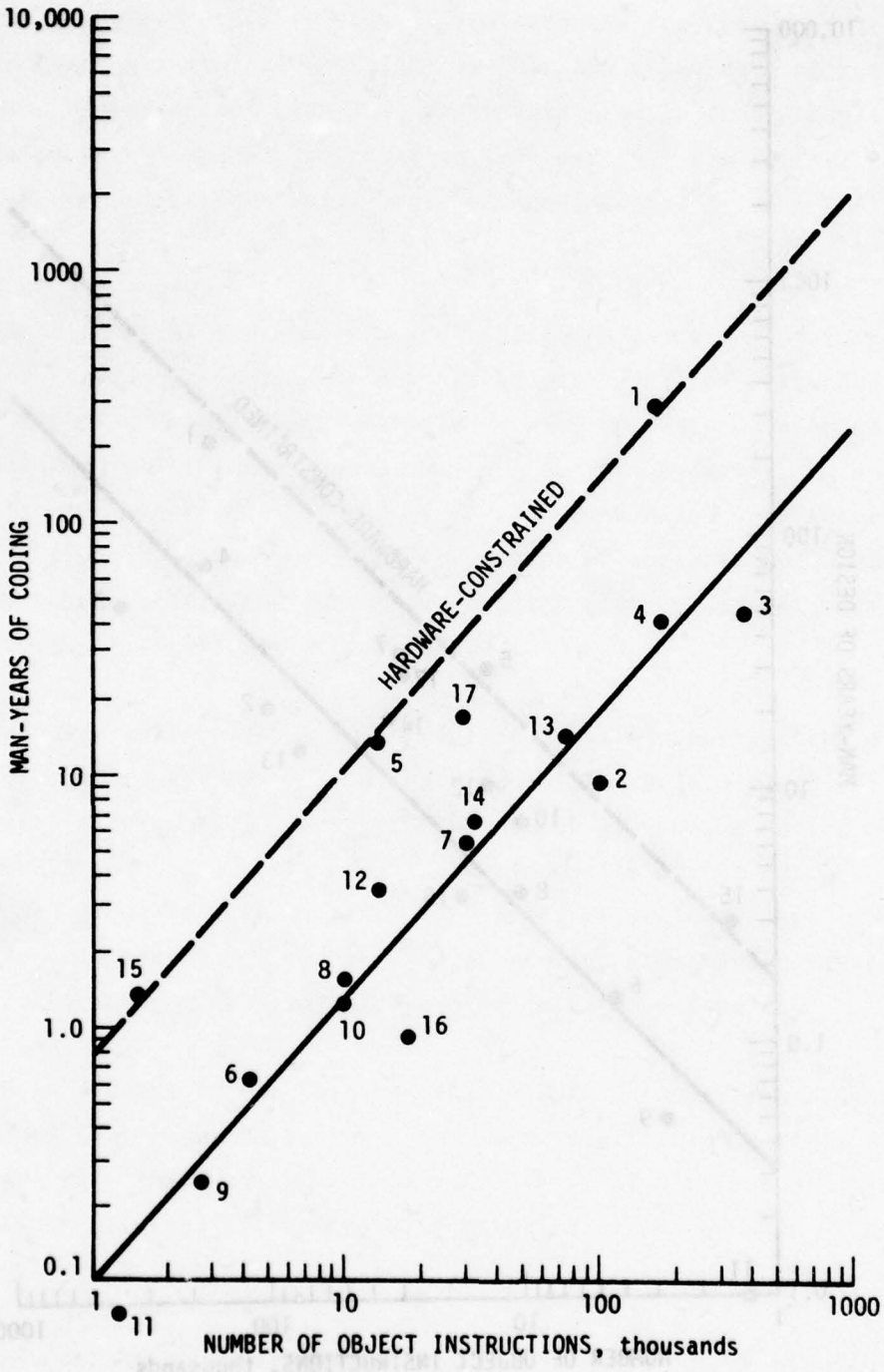


Figure 6.6. Effect of Hardware Constraint: Coding

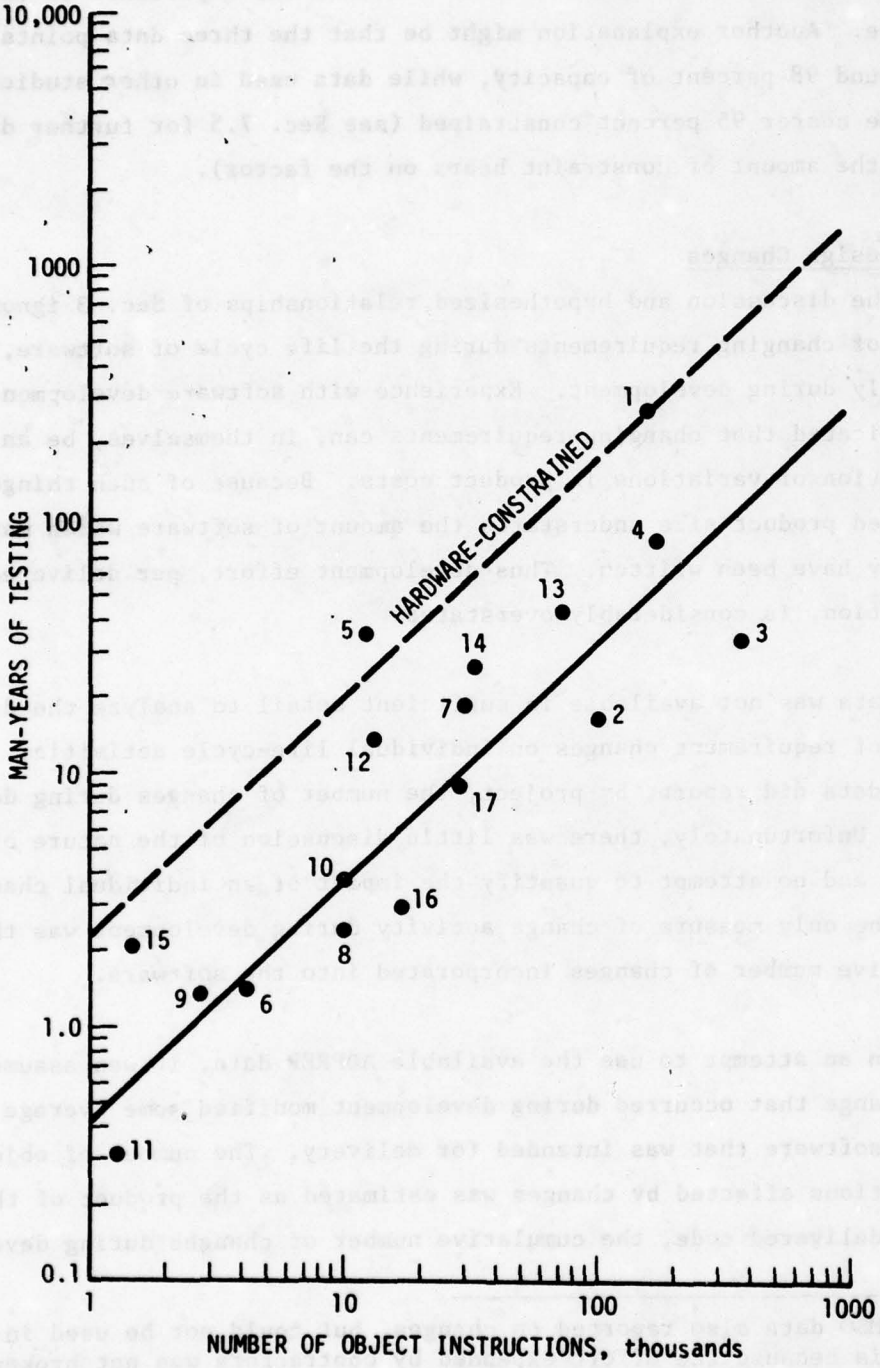


Figure 6.7. Effect of Hardware Constraint: Testing

Both this study and other results do indicate the importance of this variable. Another explanation might be that the three data points are all around 98 percent of capacity, while data used in other studies could be nearer 95 percent constrained (see Sec. 7.5 for further details on how the amount of constraint bears on the factor).

6.3.4 Design Changes

The discussion and hypothesized relationships of Sec. 3 ignore the impact of changing requirements during the life cycle of software, particularly during development. Experience with software developments has indicated that changing requirements can, in themselves, be an explanation of variations in product costs. Because of such things, the delivered product size understates the amount of software which may actually have been written. Thus development effort, per delivered instruction, is considerably overstated.

Data was not available in sufficient detail to analyze the direct impact of requirement changes on individual life-cycle activities. The ADPREP data did report, by project, the number of changes during development.* Unfortunately, there was little discussion of the nature of the changes and no attempt to quantify the impact of an individual change. Thus, the only measure of change activity during development was the cumulative number of changes incorporated into the software.

In an attempt to use the available ADPREP data, it was assumed that each change that occurred during development modified some average fraction of the software that was intended for delivery. The number of object instructions affected by changes was estimated as the product of the size of the delivered code, the cumulative number of changes during development,

*The SAMSO data also reported on changes, but could not be used in this analysis because the effort expended by contractors was not broken down by CPCI, but aggregated over the entire effort.

and the estimated fraction of the code that changed with each requirement change. It was hypothesized that development effort was linearly related both to delivered program size and to the amount of code that changed during development:

$$MM_{develop} = a_o + a_1(\text{obj.inst./1000}) + a_2[(\text{obj. inst./1000}) \times \text{no. changes}]$$

The above equation can also be written in the following form

$$MM_{develop} = a_o + [\text{obj. inst./1000}][a_1 + a_2 \times \text{no. of changes}]$$

It can then be seen that a_2 is the added effort per change to the coefficient a_1 that relates programmer productivity to program size.

Using a subset of the ADPREP data points that reported on changes to the development project, a regression analysis of this relationship was performed. The R^2 value is certainly promising, and indicates that the relationship is useful, and the T value for a_2 is significant, thus indicating the importance of the variable in explaining costs. The coefficient a_1 is not significant, and the term could be dropped.

Coefficients	T Statistics	Other Statistics
$a_o = 6.6102$		$R^2 = 0.65$
$a_1 = 0.678$	a_1: 1.105	F = 12.30
$a_2 = 0.110$	a_2: 3.998	

These results can also be used to provide insight about the average amount of the program that is modified with each requirement change. It should be obvious that the effects of the change process are not stationary over the development lifetime, and that the expected fraction of the program changed will depend on when, in the development, a requirement is changed. For example, a change occurring during the design phase does not require that any code be rewritten. It does, however, require that the design be reassessed to determine if the interfaces and interactions of the

components are still reasonable, in the light of the change. On the other hand, a change during coding may require that the design be reviewed, and that new code be introduced and integrated with the system. This would clearly require more effort.

The coefficient a_1 has units of man-months per instruction. The coefficient a_2 has units of (man-months × fraction changed)/(instruction × number of changes). By taking the raio of a_2 to a_1, the average fraction of the delivered program that would be altered with each requirement change can be estimated. For the data cited in the ADPREP study, this estimate works out to 16 percent.

Factors related to changing requirements can be applied in two ways. The first and most important application is in normalizing data that will be used to establish cost estimation techniques. An initial cost estimation technique is oriented towards predicting the costs of a project that is expected to proceed normally. Developing such an estimator from data in which changing requirements have had a significant impact on cost is clearly the wrong approach. A nominal estimation technique assumes the existence of a separate estimation technique for unforeseen design changes. A separate estimator for design changes would allow costs and benefits to be considered for each change prior to its incorporation into the development.

The second application of information about requirements changes is the development of a cost estimating technique for life-cycle cost that requires the user to estimate the expected number of changes that will occur during the project lifetime. The impact of change can be anticipated and explicitly incorporated into the initial estimate of costs. Making an estimate of the expected changes to a project requires that the number of changes be related to some other, more easily estimated, characteristics of the project such as duration or application area. Without such an auxiliary relationship, estimation of the expected number of changes would be nothing more than a guess.

6.4 ALTERNATIVE PRODUCT MEASURES

Throughout Sec. 3 of this report, an attempt was made to relate measures of the product of a software development to the effort and costs associated with that product. The most obvious and commonly suggested measure is software size, expressed as the number of either source-language or object-language statements. Each seems to have an appropriate application, and it is important that a consistent measure be used when comparing data on various projects.

Inconsistency in this measure of program characteristics is probably, by itself, significant in explaining variation in the data reported, since it could vary by a factor of 2 to 5 (as discussed in Sec. 6.3.2 on the impact of programming language). Since there is no way of checking most reported data, the possible inconsistency of this measure must be accepted.

Assuming that all measures of computer programs can be converted to a normalized form through a scaling relationship, a further possible source of inconsistency exists. It can be argued that, for high-level languages, programming style (i.e., coding conventions and techniques) can cause the relationship between source code and object code to vary significantly from programmer to programmer, therefore invalidating any scaling relationship between source and object codes.

The AOES data described a large number of programs in terms of both number of source-code card images and number of object-code instructions generated by a JOVIAL compiler. This data offered an opportunity to test the claim that significant variations in the relation between source-code and object-code instruction counts are introduced by programmer style.

The set of JOVIAL programs considered all run on the same computer, so variations introduced by computer instruction sets are removed. Furthermore, the programs perform a wide range of functions, from orbit determination to compilation and operating system functions, and are written by many authors. Using the subset of programs written entirely in JOVIAL, the following regression analysis was performed:

$$\text{obj. inst.} = a_o + a_1(\text{card images})$$

A very strong linear relationship was found between source and object code for this set of JOVIAL programs. The statistics of the regression analysis are very significant.

Coefficients	T Statistics	Other Statistics
$a_o = -30.495$		$R^2 = 0.94$
$a_1 = 2.4$	a_1: 73.21	$F = 5360.0$

These results indicate that the claim is invalid and that stylistic variations in the use of JOVIAL become insignificant over the length of an average program. The data used and the results of the regression analysis are shown in Fig. 6.8.

This suggests that a table of conversion factors for language can be compiled for various programming languages and host computers. These factors could be applied to construct normalized measures of product size. Table 6.2 is a compilation of such data based on the information used in this study. For example, the above equation would indicate that for the CDC 3800 JOVIAL system, a JOVIAL statement (assuming one per card) results in an average of 2.4 object instructions. The table is far from complete, and should be extended to include other commonly used computer systems and languages.

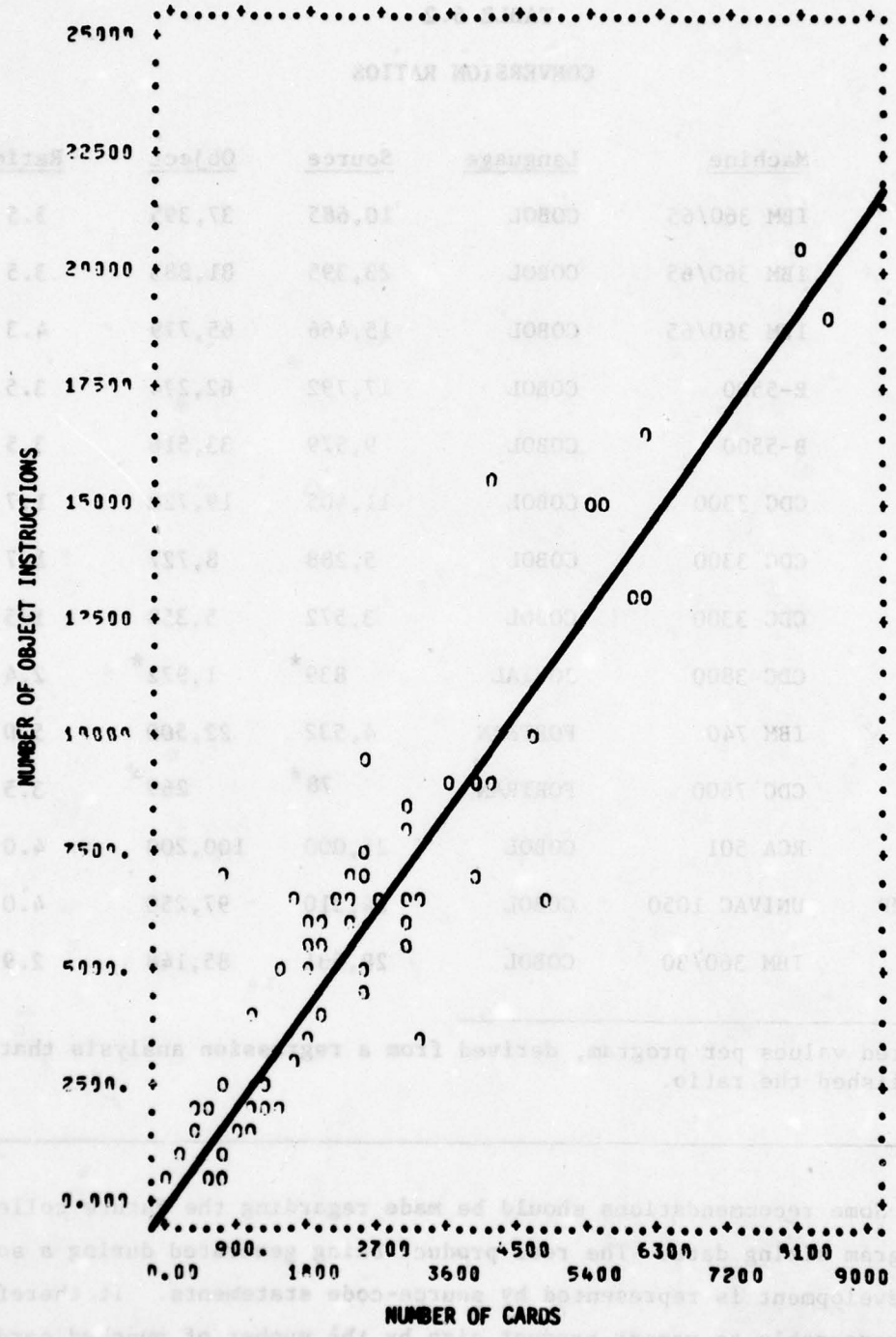


Figure 6.8. Object Instructions Vs Number of Cards

TABLE 6.2

CONVERSION RATIOS

Name	Machine	Language	Source	Object	Ratio
SAMS	IBM 360/65	COBOL	10,685	37,395	3.5
SCS	IBM 360/65	COBOL	23,395	81,883	3.5
TAADS	IBM 360/65	COBOL	15,466	65,779	4.3
MACE	B-5500	COBOL	17,792	62,274	3.5
STATEM	B-5500	COBOL	9,579	33,516	3.5
ADMSS	CDC 3300	COBOL	11,405	19,720	1.7
CRFS	CDC 3300	COBOL	5,288	8,727	1.7
DIS	CDC 3300	COBOL	3,572	5,358	1.5
AOES	CDC 3800	JOVIAL	839*	1,972*	2.4
ADOBE	IBM 740	FORTRAN	4,532	22,500	5.0
TRAID	CDC 7600	FORTRAN	78*	269*	3.5
MAFR	RCA 501	COBOL	25,000	100,200	4.0
MILSTAMP	UNIVAC 1050	COBOL	24,310	97,250	4.0
MPAS	IBM 360/30	COBOL	29,631	85,148	2.9

*Expected values per program, derived from a regression analysis that established the ratio.

Some recommendations should be made regarding the future collection of program sizing data. The real product being generated during a software development is represented by source-code statements. It therefore seems reasonable to report product size by the number of punched cards (or card images) containing source-code statements. However, while many

programming languages are generally punched one statement to a card (e.g., FORTRAN and assembly languages), others are not (e.g., JOVIAL and ALGOL). For such languages it is possible to have more than one statement per card. In these cases, the number of card images may understate the number of statements.

The card count should include comment cards. Comments can be very effective in describing algorithms and aiding in debugging, and they should be considered an integral part of the program being delivered.

It is our recommendation that programs be measured in terms of the number of source cards delivered. It is important that scaling factors be developed to relate one language to another (e.g., FORTRAN statements per card to JOVIAL statements per card). For assembly-language programs, scaling factors should also be developed to account for differences in machine instruction sets. Such differences can also affect the number of statements required. These factors are important in arriving at normalized measures of product that are language-independent, for comparison of programs written in different languages for different systems.

Software development produces other products than software, notably the documentation that accompanies the software. It is possible that a measure of documentation can serve as a surrogate for product size measured in source-language or object-language instructions. To test this conjecture, it was assumed that documentation describing a project was related to the final product, so long as common documentation requirements and formats applied. Relaxing this last constraint makes any comparison meaningless.

The AOES data provided information that could be used to test the validity of this conjecture. For each of the many programs examined, the page counts for Part II Specifications (subject to common formats which are described in MIL STD 483, Appendix IV) were available, as well

as measures of the product size in terms of the number of source statements. The following regression was performed:

$$\ell n(\text{source cards}) = a_o + a_1 \ell n(\text{pages of Part II Specs})$$

with the following significant statistics:

Coefficients	T Statistics	Other Statistics
$a_o = 7.37$		$R^2 = 0.56$
$a_1 = 1.403$	a_1: 15.9	F = 252.0

Figure 6.9 illustrates the results of the regression analysis.

The relationship has a low R^2 value that indicates it should not be used to replace existing measures of software size. The relationship might be improved upon by considering other factors such as the type of software. (Recall that the AOES data contains a wide variety of software types.)

A relationship between documentation and program size might be used to screen initial estimates of program size at the time of CDR when the first documentation exists in draft form. Such a relationship would allow the original estimates of software sizes to be compared with the sizes of corresponding documentation. This comparison could identify initial estimates that are at significant variance with that predicted by the size of their documentation. The sizing of the program product might be subjected to review, leading to a possible revision of project costs and schedule. One should view such a relationship as being unique to a project, and serving only as a general management tool.

The relationship also offers promise in directly calculating the cost of documentation. Since lines of code are related to number of pages, an estimating equation for documentation could be calculated. Alternatively, the documentation estimate could indirectly be related to lines of code by estimating it as a percentage of development man-hours, if the latter is based on lines of code.

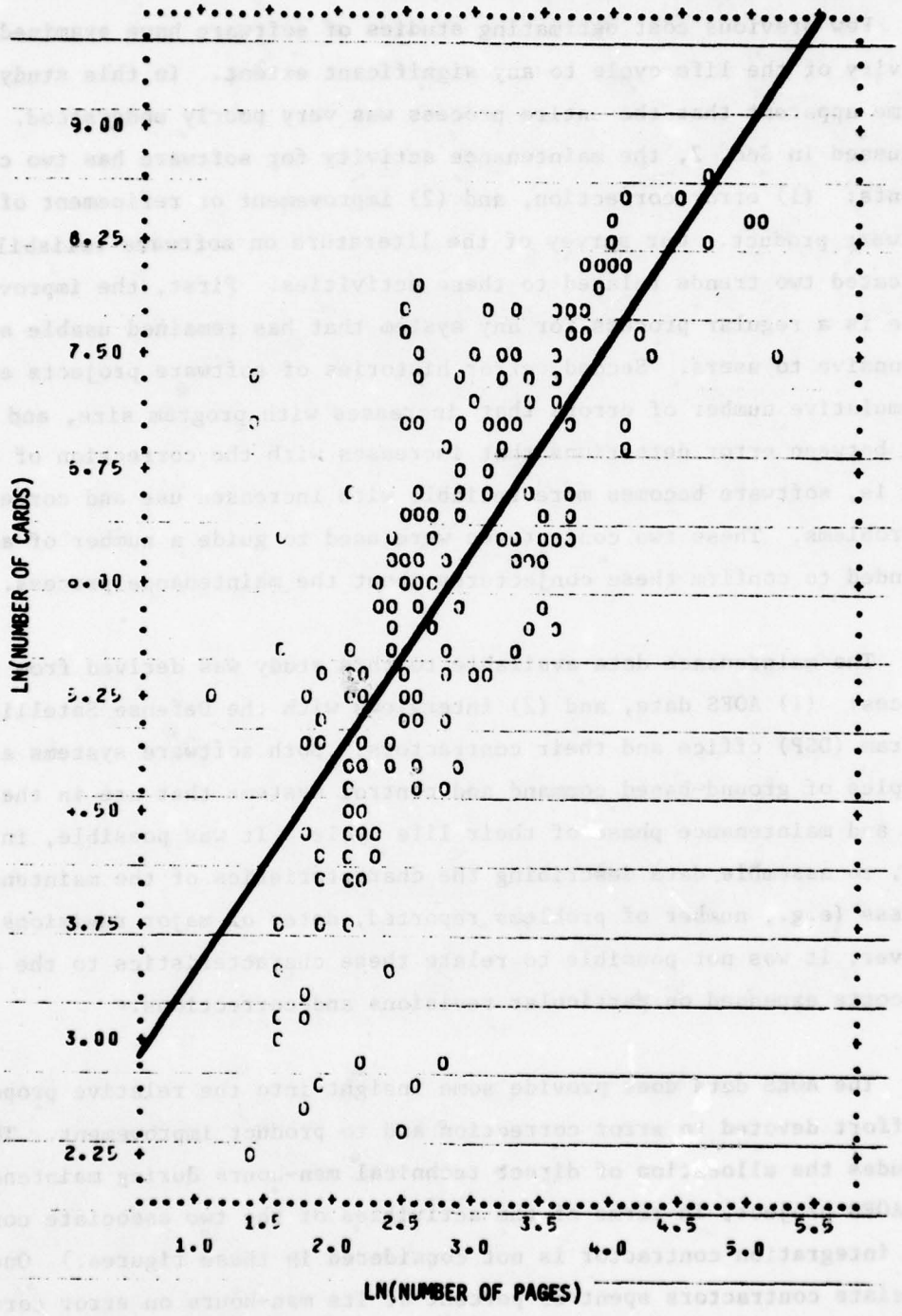


Figure 6.9. Number of Cards Vs Number of Pages of Part II Specifications

6.5 CHARACTERISTICS OF THE MAINTENANCE ACTIVITY

Few previous cost estimating studies of software have examined this activity of the life cycle to any significant extent. In this study, it became apparent that the entire process was very poorly understood. As discussed in Sec. 2, the maintenance activity for software has two components: (1) error correction, and (2) improvement or refinement of the software product. Our survey of the literature on software reliability indicated two trends related to these activities. First, the improvement cycle is a regular process for any system that has remained usable and responsive to users. Second, error histories of software projects exhibit a cumulative number of errors that increases with program size, and a mean time between error detections that increases with the correction of errors. That is, software becomes more reliable with increased use and correction of problems. These two conjectures were used to guide a number of analyses intended to confirm these conjectures about the maintenance process.

The maintenance data available to this study was derived from two sources: (1) AOES data, and (2) interviews with the Defense Satellite Program (DSP) office and their contractors. Both software systems are examples of ground-based command and control systems that are in the operation and maintenance phase of their life cycle. It was possible, in each case, to assemble data describing the characteristics of the maintenance process (e.g., number of problems reported, dates of major revisions, etc.). However, it was not possible to relate these characteristics to the effort and costs expended on particular revisions and corrections.

The AOES data does provide some insight into the relative proportions of effort devoted to error correction and to product improvement. The data includes the allocation of direct technical man-hours during maintenance for the AOES project, in terms of the activities of the two associate contractors. (The integration contractor is not considered in these figures.) One of the associate contractors spent 25 percent of its man-hours on error correction and 75 percent on product improvement. For the other associate contractor, the division was 33 percent and 67 percent. Thus, for both contractors, most of the effort went to improvement rather than error correction.

In order to understand the maintenance effort, the events (revisions and errors) within the maintenance activity were investigated. The cost of revisions to existing versions of software to incorporate new requirements could presumably be estimated by techniques similar to those used for development activities. Such techniques would be based upon characteristics of the newly written parts of the code, but would also reflect the existence of a body of code and previous experience that would not be present in a new development.

Figure 6.10 illustrates histories of revisions in the AOES and DSP projects. It shows time lines for DSP's Overseas Ground Station (OGS) and CONUS Ground Station (CGS) programs, and AOES's Satellite Control Facility (SCF) System Support Tape. Each time line shows the major releases of each system.

For CGS and OGS, the revisions to the system occurred at fairly regular intervals until version 6. At this point a very extensive reorganization of both systems was made that required a longer time. For SCF, the minor revisions (indicated by letters appended to the version number) occurred at close intervals and the major system revisions (SCF-15.0, SCF-15.1, SCF-15.2) occurred at much longer intervals.

The point is that a regular pattern of modification takes place and is to be expected during the operations period. The pattern observed is quite consistent with recently published data describing the evolution of other large software systems.[26] However, these modifications have to do largely with product improvement, rather than the initial development's success or failure in meeting original requirements. Therefore no trade-off exists, and a decision to fund the improvement should be based on costs and benefits perceived at that time.

Error correction, on the other hand, is clearly driven by the problems reported during operation of a system and the response times required to correct the reported errors.

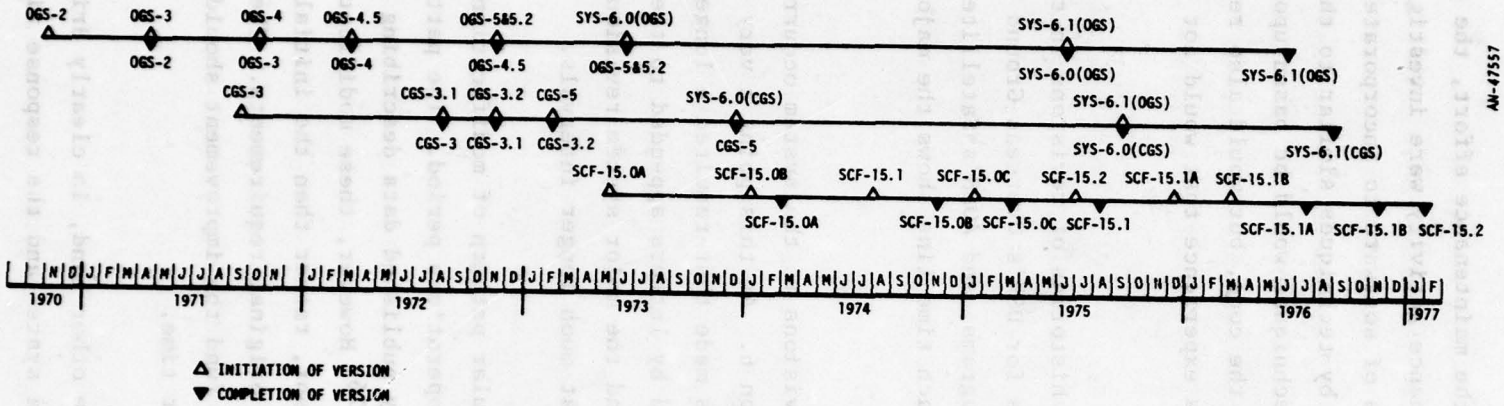


Figure 6.10. Comparison of System Version Development Time Frames

The technical literature on software reliability indicates that the error process of operational software exhibits two major trends. First, the total number of problems encountered with a given model of software is limited, and the limit is related to the size of the software. Second, the mean time between error detections increases with use and corresponding error correction. Figure 6.11 indicates both of these trends.

The data collected for the AOES project were at a sufficient level of detail to allow testing of some of these assertions. Each release of the system was described by records of the changes made to produce the release, and the errors encountered with operational use of the release. Changes to the program resulted from two things. First, design change requests (DCRs) could be incorporated. Effectively, these are engineering changes. Second, discrepancy report forms (DRFs) associated with the base system used in generating the revision could be corrected in the new release. These uncorrected errors could have been temporarily repaired with "octal correctors,"

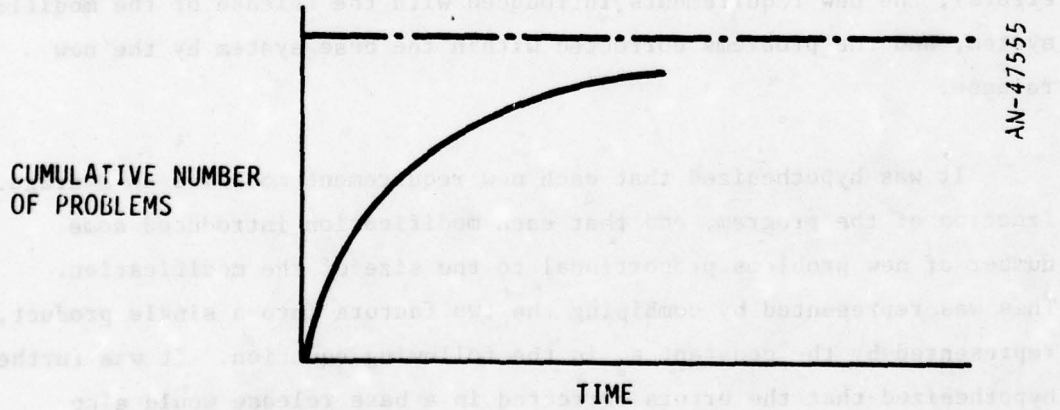


Figure 6.11. MTBF Increase with Time

or been deferred by using operating procedures that avoided the problem. Both the DCRs and DRFs caused the source code forming the new release to be modified. Each DCR and DRF could be associated with unique computer programs in the system, so that a complete history of modification could be developed by program. Errors encountered with the operational use of the new release of the system were reported as DRFs written against programs in the new release, rather than the base system.

It was first hypothesized that the parameter "number of problems" relates only to the size of the programs. A regression analysis of the equation

$$\text{No. of Problems} = a_o + a_1 \text{ (no. cards/1000)}$$

was performed, with the scatter shown in Fig. 6.12. Obviously the explanation was not very good. Examination of the data, however, showed that the points above the line corresponded to those programs with a significant number of design changes. It was therefore hypothesized that the number of problems reported, once a release of the system became operational, would be related to the size of the software (residual errors), the new requirements introduced with the release of the modified system, and the problems corrected within the base system by the new release.

It was hypothesized that each new requirement modified an average fraction of the program, and that each modification introduced some number of new problems proportional to the size of the modification. This was represented by combining the two factors into a single product, represented by the constant a_2 in the following equation. It was further hypothesized that the errors corrected in a base release would also modify an average amount of the program, and in turn, introduce new problems with numbers proportional to the size of the change. Again, the two considerations were combined as a product and represented by

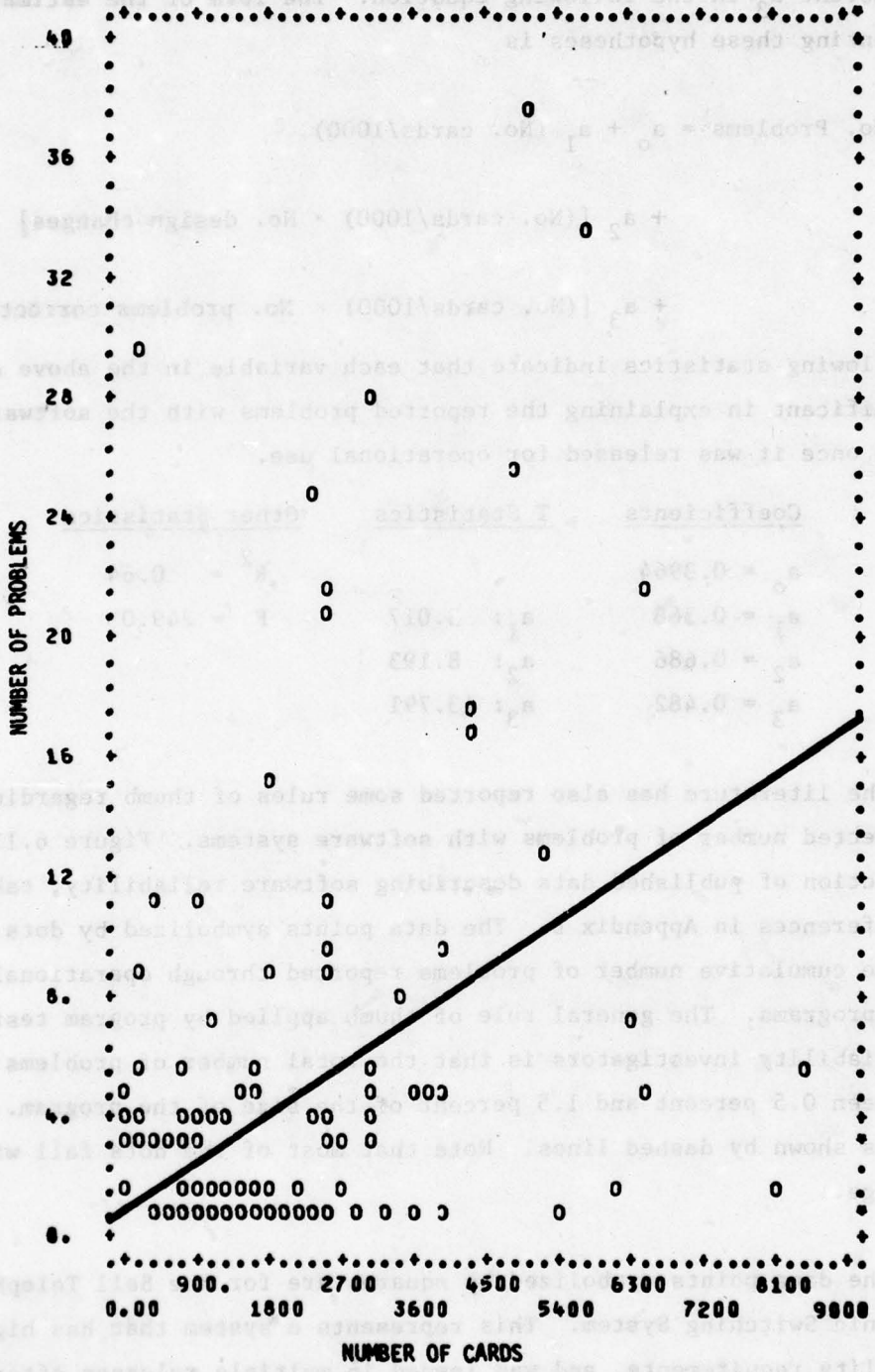


Figure 6.12. Number of Errors Vs Number of Cards

the constant a_3 in the following equation. The form of the estimator representing these hypotheses is

$$\text{No. Problems} = a_o + a_1 \text{ (No. cards/1000)} + a_2 \text{ [(No. cards/1000) } \times \text{ No. design changes]} + a_3 \text{ [(No. cards/1000) } \times \text{ No. problems corrected]}$$

The following statistics indicate that each variable in the above equation is significant in explaining the reported problems with the software version once it was released for operational use.

Coefficients	T Statistics	Other Statistics
$a_o = 0.3964$		$R^2 = 0.64$
$a_1 = 0.368$	a_1: 3.017	$F = 249.0$
$a_2 = 0.686$	a_2: 8.193	
$a_3 = 0.482$	a_3: 13.791	

The literature has also reported some rules of thumb regarding the expected number of problems with software systems. Figure 6.13 shows a collection of published data describing software reliability, taken from references in Appendix B. The data points symbolized by dots represent the cumulative number of problems reported through operational use of the programs. The general rule of thumb applied by program testing and reliability investigators is that the total number of problems will be between 0.5 percent and 1.5 percent of the size of the program. This range is shown by dashed lines. Note that most of the dots fall within the range.

The data points symbolized by squares are for the Bell Telephone Electronic Switching System. This represents a system that has high reliability requirements, and was issued in multiple releases after thorough testing. The problem reports are only for those problems

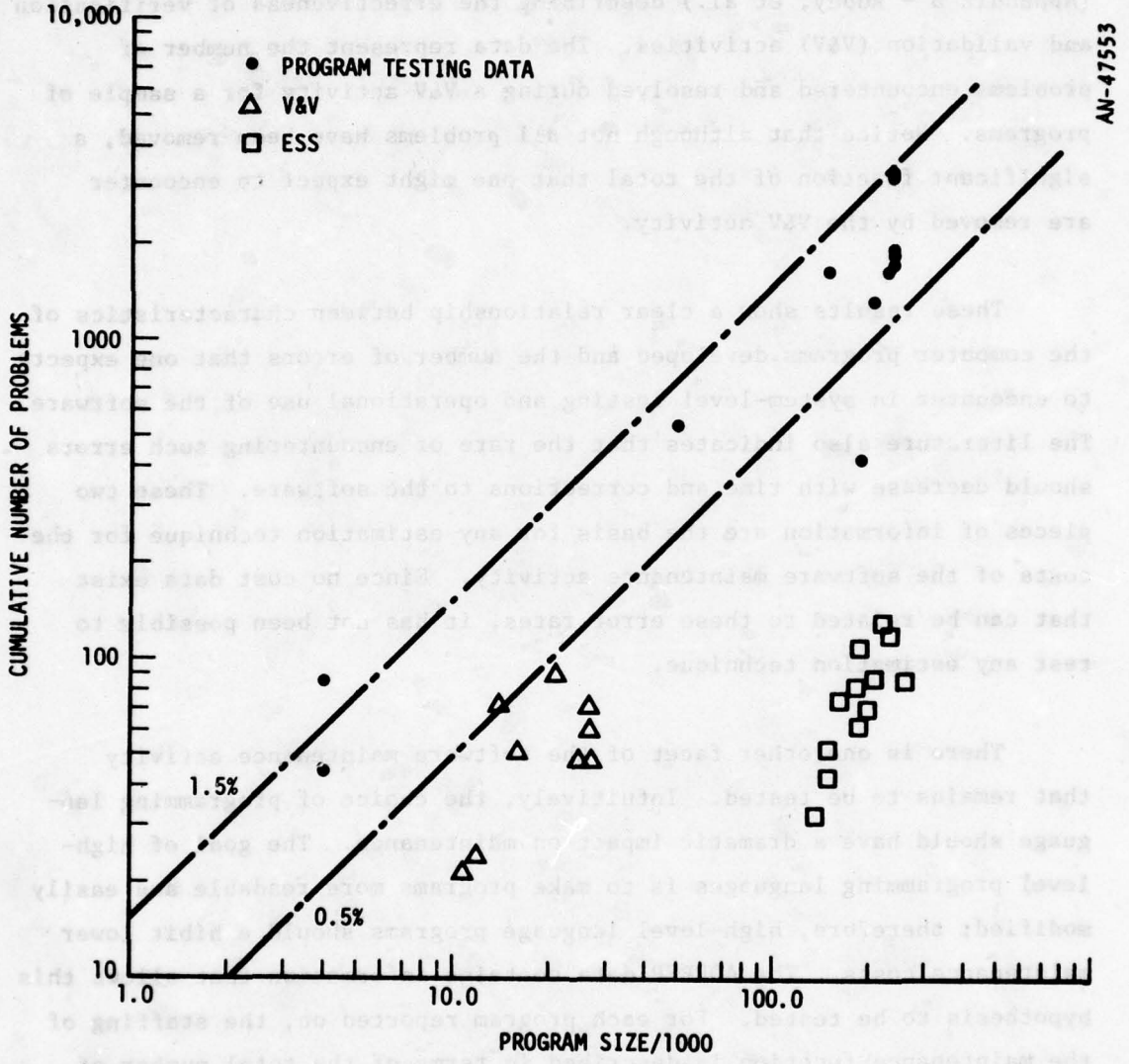


Figure 6.13. Software Reliability Data

encountered in operational use. Note that this data falls far below the trend for other programs, as might be expected. The remaining points, symbolized by triangles, represent data reported by Logicon (Appendix B - Rubey, et al.) describing the effectiveness of verification and validation (V&V) activities. The data represent the number of problems encountered and resolved during a V&V activity for a sample of programs. Notice that although not all problems have been removed, a significant fraction of the total that one might expect to encounter are removed by the V&V activity.

These results show a clear relationship between characteristics of the computer programs developed and the number of errors that one expects to encounter in system-level testing and operational use of the software. The literature also indicates that the rate of encountering such errors should decrease with time and corrections to the software. These two pieces of information are the basis for any estimation technique for the costs of the software maintenance activity. Since no cost data exist that can be related to these error rates, it has not been possible to test any estimation technique.

There is one other facet of the software maintenance activity that remains to be tested. Intuitively, the choice of programming language should have a dramatic impact on maintenance. The goal of high-level programming languages is to make programs more readable and easily modified; therefore, high-level language programs should exhibit lower maintenance costs. The ADPREP data contains information that allows this hypothesis to be tested. For each program reported on, the staffing of the maintenance function is described in terms of the total number of maintenance people assigned per month.

In Fig. 6.14, programs written in machine language are plotted with dots and programs written in a high-level language are plotted with squares. It is generally difficult to discern if the published

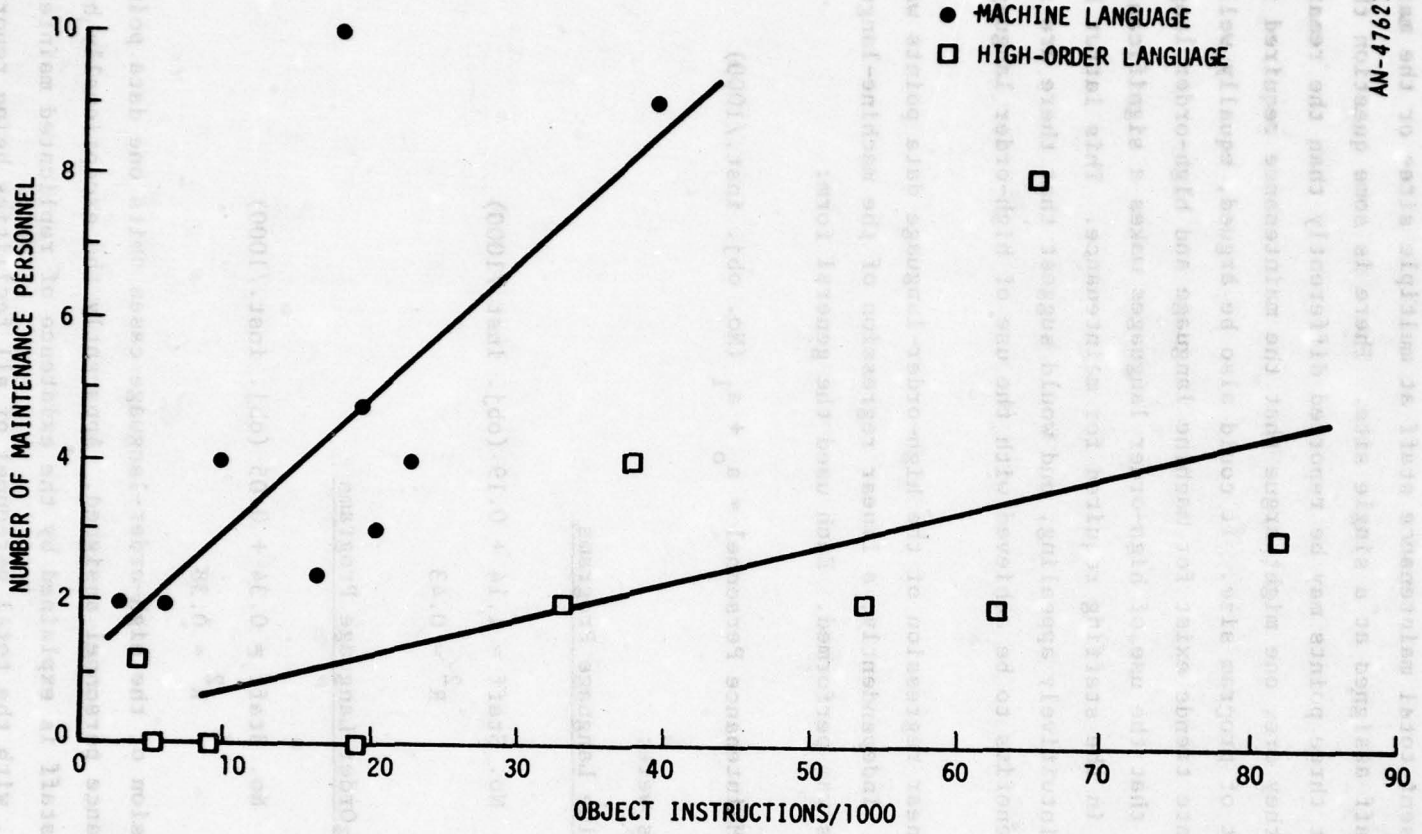


Figure 6.14. Maintenance Requirements for Machine-Language and High-Order-Language Programs

data represent total maintenance staff at multiple sites or the maintenance staff assigned at a single site. There is some question that the highest three points may be reported differently than the remaining data. If they are, one might argue that the maintenance required is independent of program size. It could also be argued, equally well, that separate trends exist for machine language and high-order language, indicating that the use of high-order languages makes a significant difference in the staffing required for maintenance. This latter hypothesis is intuitively appealing, and would suggest that there are substantial benefits to be achieved with the use of high-order languages.

A linear regression of the high-order-language data points was performed. Independently, a linear regression of the machine-language data points was performed. Each used the general form:

$$\text{No. Maintenance Personnel} = a_o + a_1 \text{ (No. obj. inst./1000)}$$

The results were:

Machine Language Programs

$$\text{No. Staff} = 1.14 + 0.19 \text{ (obj. inst./1000)}$$

$$R^2 = 0.43$$

High-Order Language Programs

$$\text{No. Staff} = 0.34 + 0.05 \text{ (obj. inst./1000)}$$

$$R^2 = 0.38$$

The regression of the high-order-language cases omits one data point with 31 maintenance personnel assigned. Apparently the exceptionally high number of staff is explained by the existence of replicated maintenance facilities, with the total personnel of all facilities being reported.

Note also that the R^2 statistics are not very good, and the relationships are tenuous.

In these two regressions, the maintenance staffing for machine-language programs is approximately four times as large. This number is consistent with language factors derived in Sec. 6.3.2 in this report that show assembly language requires 2 to 5 times the effort of high-level-language programs with the same number of object instructions. It also shows that assembly language is not used for larger programs, probably due to the loss of efficiency.

Based upon this initial examination of the maintenance activity, several guidelines about the maintenance process can be stated. The first is that the program-improvement aspect of maintenance is a regular, repeating process for programs that have a long useful lifetime. Because maintenance staffing appears to be somewhat arbitrary, the extent to which programs are improved and the schedules for this type of maintenance are probably based on the residual manpower available after error-correction activities for the currently operational version of a system.

The error-correction aspect of program maintenance appears to follow some general trends. First, the total number of errors is probably limited, with the limit related to program size and the number of modifications installed in the program during maintenance. Second, the MTBF for software will increase with time and corresponding error correction. Thus, for a static program, one should expect to be able to taper the maintenance effort with time. The AOES data support this conjecture. If each release of the system did not stabilize with age, then maintenance effort required should grow as the number of releases being concurrently supported. There is no evidence to indicate such growth.

7 STATE-OF-THE-ART TECHNIQUES FOR ESTIMATING SOFTWARE COSTS

Although this study did not result in the development of CERs for each of the life-cycle phases, we have evaluated a number of software resource relationships. However, the work described in the previous sections is not very useful for Program Offices unless it is incorporated into previously-developed software cost estimating techniques. This section briefly describes some of the more important techniques from the literature and elsewhere that were reviewed during this study. This information is intended to provide some insight into the current state-of-the-art in software cost estimating.

This was not a major focus of the study; we do not claim this section is a thorough review, evaluation, or assimilation of all software cost estimating techniques that have been developed or proposed.* Rather, it is a composite of our experience which incorporates results from other sources. No software cost estimating techniques have yet demonstrated much precision. The reader is advised to use the techniques very carefully, keeping in mind their acknowledged imprecision.

Two basic types of software cost estimating models are discussed here. The first is a disaggregated approach in which the costs for some of the major life-cycle phases are estimated separately and then combined to form an estimate of the total cost (Sec. 7.1). The second is an aggregated approach in which total costs for a software project are estimated and then allocated to the life-cycle phases (Sec. 7.2). The disaggregated model is based primarily on earlier GRC work;[22] the aggregated approach has been used for most software cost estimating to date.

---

*A number of reviews have previously been published.[1,8,9]

In Sec. 7.3 we discuss various special factors (language, hardware constraints, etc.) which significantly impact the cost of software and in Sec. 7.4 we discuss how to time-phase aggregated estimates.

The estimation of software maintenance costs is discussed in Sec. 7.5 and concluding remarks are presented in Sec. 7.6.

7.1 BASIC MODEL (DISAGGREGATED)

The most common approach to software cost estimating is to utilize a basic model to estimate total man-months, computer hours, documentation, etc. for the entire project, and then use these estimates to derive the total project cost. However, in Spetember 1975, under the sponsorship of the National Aeronautics and Space Administration, General Research Corporation developed a basic model which employs a disaggregated rather than an aggregated approach to software cost estimating.[22] Two basic models were developed: an "initial model" and a "refined model." The initial model was based upon selected factors and relationships from the literature, together with a number of data points and observations developed by GRC. The second model was based upon another set of data compiled by GRC. Each of these is described below.

7.1.1 Initial Software Costing Model

The steps required to develop an estimate using the initial software costing model are as follows:

1. Estimate Total Software Size. The first step is to estimate the number of object instructions to be developed for the entire system. In developing this estimate, the required functions are divided into subfunctions; the size of each subfunction is estimated, either by analogy to a previously implemented, similar subfunction, or by a constraints technique;[44] and the individual estimates are combined into an estimate of the total size of the software system.

2. Estimate Effective Software Size. The estimate developed in step (1) is modified to account for the word length of the target computer and the effects of design change activity. The model described in Ref. 22 assumes a word length of 32 bits; adjustments are made if the target computer has a different word length. Furthermore, it was estimated that design change activity increases the effective size of the software by 70 percent (e.g., 1700 instructions are developed in order to deliver a 1000-instruction program).[22]

3. Estimate Coding Labor. Coding labor is estimated from the effective software size estimated in step (2). Some preliminary data on coding productivity are presented in Table 7.1. The actual coding rate selected should be based on the complexity of the software, as well as its size. If different portions of the software are of different complexity, then different rates should be used for each portion.

TABLE 7.1

AN INTERIM CODING PRODUCTIVITY MODEL

(Ref. 22)

Project Size (No. Instructions) / Difficulty	Instructions Per Man-Month	
	Total Size <30,000	Total Size >30,000
. Easy Batch Few Interactions	4,000	2,000
. Medium Adaptation or Some Interactions	---	1,000
. Difficult Real-Time Many Interactions	---	300

4. Adjust Coding Labor. The coding labor estimated in step (3) is next adjusted to account for hardware constraint and programming language. It was estimated[22] that the impact of constrained hardware resources is approximated by:

$$\text{Adjusted Coding Labor} = \left(\frac{0.7}{1 - \sqrt{P - 0.5}}\right) \times \text{Coding Labor}$$

where

P = fraction of memory used.

It was estimated[22] that the use of a high-order language requires 25 percent more memory than the use of object language, but that it reduces the required coding labor to 20 percent of that required by the use of object language, for each object-language instruction.

5. Estimate Total Technical Labor Cost. The estimate of total technical labor cost is based on the estimate of coding labor developed in step (4). It was estimated[22] that the percentages of effort devoted to each of the major phases of software development are as follows:

- Analysis and Design (36%)
- Coding (21%)
- Integration and Testing (43%)*

Based on these estimates, the amount of labor required for analysis, design, integration, and testing may be estimated. Totaling the labor estimates for all three phases and multiplying by the estimated labor cost gives an estimate of the cost of all technical labor.

6. Estimate Computer Costs. Computer costs associated with coding, integrating, and testing the software are assumed to be proportional

*This initial model does not incorporate any tradeoffs between phases or program size. These refinements are introduced subsequently.

to the costs of labor. They are estimated to add 104 percent (in 1972) to the total technical labor cost estimated in step (5).

7. Estimate Management and Documentation Costs. The costs of project management and software documentation, like computer costs, are assumed to be proportional to the costs of labor. They are estimated to add 28 percent to the total technical labor cost estimated in step (5).

8. Sum Costs. The final step is to sum the cost estimates of steps (5), (6), and (7) to determine the total software cost estimate.

The main problem in this cost model is that small errors in the first four steps are multiplied in the later steps.

7.1.2 Refined Software Costing Model

A refined software model was developed concurrently with the model described above. In effect, the refined model replaced steps (3), (4), and (5) in the initial model with the following means of estimating total technical labor costs:

1. Estimate Software Analysis and Design Labor Cost. The following formula was used to estimate this cost:

$$\ln Y = -1.17 + 1.03 \ln X_1 + 1.73 X_2$$

where

Y = man-years of effort

X_1 = number of object instructions, thousands

$$X_2 = \begin{cases} 1, & \text{if software development is hardware-constrained} \\ 0, & \text{if not.} \end{cases}$$

2. Estimate Coding Labor Cost. The following formula was used to estimate this cost:

$$\ln Y = -2.3 + 1.13 \ln X_1 + 2.12 X_2$$

where

Y, X_1, and X_2 are defined as above.

3. Estimate Integration and Testing Labor Cost. The following formula was used to estimate this cost:

$$\ln Y = -0.86 + 0.934 \ln X_1 + 1.64 X_2$$

where

Y, X_1, and X_2 are defined above.

The development of these formulas was based on data from seventeen software development programs, twelve of which were proprietary. The programs included two NASA projects and a variety of military space, aircraft, and missile applications.

7.2 BASIC MODEL (AGGREGATED)

As was previously stated, most approaches to software costing employ a basic model which estimates total man-months, computer hours, documentation, etc., from which total project costs may be derived. Two types of models are common, those that are based on the size of the software to be developed, and those that are not.

7.2.1 Estimates Based on Size

Most of these software costing models use an estimate of the size of the software to be developed as a basis for all subsequent calculations. Since it has been demonstrated in the literature that cost is not a linear function of size,[9,20,27] especially for larger programs, estimating relationahips for larger programs should use nonlinear relationships.

The steps required to develop a cost estimate based on the expected size of the software are as follows:

1. Determine the Type of Software. The first step is to categorize the software by type. Wolverton[19] selected the following categories based on extensive experience:

- Control routines
- Input/output routines
- Pre- or post-algorithm processors
- Algorithms
- Data management routines
- Time-critical processors

Both Wolverton[19] and Aron[28] also recommend that software complexity be taken into account by estimating the percentage of software that fits into each of the following categories:

- Easy--very few interactions with other system elements (e.g., applications programs)
- Medium--some interactions (e.g., compilers, I/O packages, utilities, etc.)
- Hard--many interactions (e.g., operating systems)

2. Estimate the Size of the Software. Size is normally defined as (1) the number of operational instructions (i.e., instructions that will perform the basic functions for which the system was designed), (2) the number of delivered instructions (i.e., operational instructions plus supporting instructions such as hardware diagnostics and debugging aids), or (3) the number of developed instructions (i.e., all instructions written during the course of the project, whether delivered or not).

Several techniques (which are not addressed here) are currently available for estimating software size (e.g., quantitative, constraint, analog, etc.)[44] Often, different techniques are used for different portions of the software. In any case, however, the estimated size of the software should be partitioned according to whether it is (1) new

software, (2) existing software to be modified, or (3) existing software that may be used without modification.

Aron[28] presents a method for sizing software by partitioning it into modules whose average size can be estimated from previous experience. It is then simply a matter of multiplying the number of modules by the average module size to arrive at a total. An average module size of 400-1000 assembly-language instructions is common.

3. Estimate Required Labor. From the size, an estimate is formed of the amount of labor required to analyze, design, code, check out, integrate, and test the type and quantity of software identified in steps (1) and (2). There are two basic approaches to estimating required labor: productivity measures and parametric estimating.

Productivity measures are estimates of the output of software development personnel in terms of the number of lines of code per unit time that can be produced.* Table 7.2 presents an example of a productivity table developed by Aron.[28] Another source[29] developed the following estimates based on the type of software:**

Software Type	Productivity (instructions per man-month)
• Mathematical operations	166
• Report generation	125
• Logic operations	83
• Signal processing or data reduction	50
• Real-time or executive functions	25

*The term "line of code" refers to a fully checked-out, tested, and documented statement in the selected language.

**These estimates include all technical, support, and management labor for the project.

TABLE 7.2

PRODUCTIVITY TABLE[28]

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

Nanus and Farr[30] developed estimates of 255 instructions per man-month for operational programs and 311 for utility programs. Another source[29] estimated between 166 and 250 instructions per man-month for utility programs. Estimates presented in Sec. 6.3.1 were 33 instructions per man-month for commercial programs and 100 for defense-system programs. Figure 7.1 illustrates Boehm's[31] view of how productivity is increasing over time as a result of improved tools and techniques.

Parametric cost estimates relating to size are developed by plotting the number of man-months for software development versus the number of machine instructions, for a representative sample of past projects, and then fitting a curve to these points. Figure 7.2 is an example of such a curve.[32] In this case, the estimate of labor is derived through the use of the size estimate developed in step (2). If the type and complexity of software are to be considered when using this technique, then an appropriate <u>set</u> of curves must be developed.

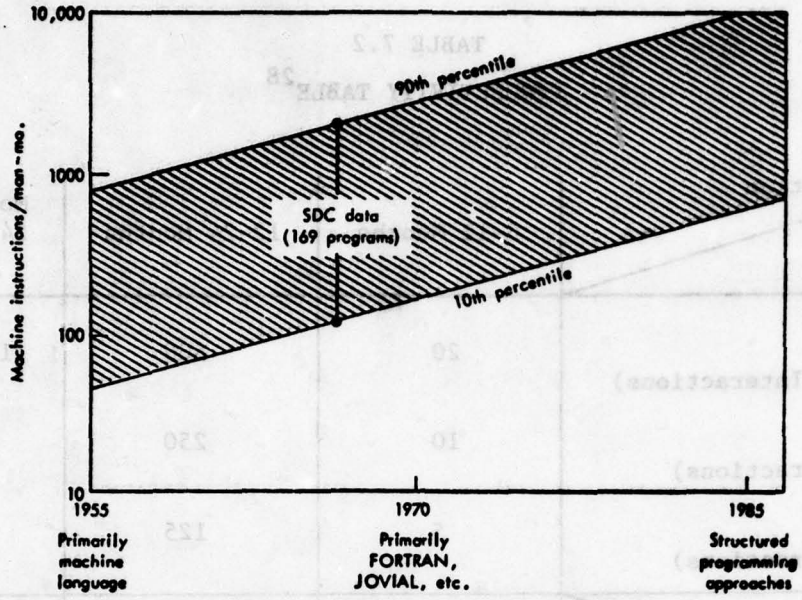


Figure 7.1. Technology Forecast: Software Productivity (Ref. 31)

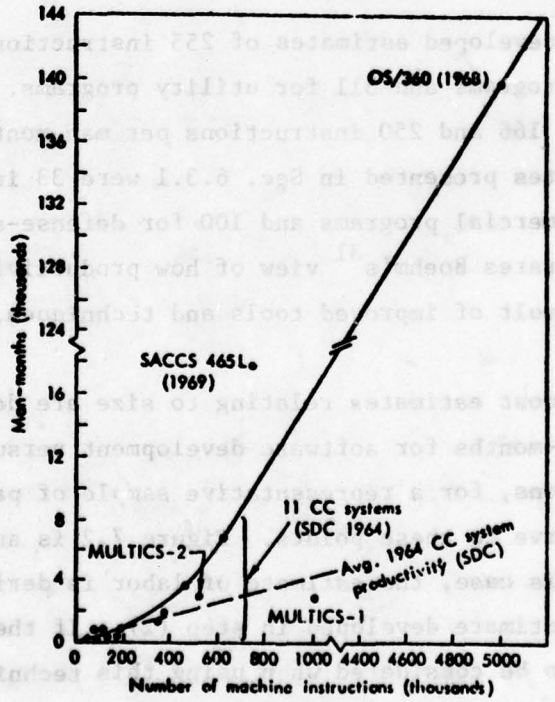


Figure 7.2. Man-Months vs Program Size (Ref. 32)

4. Adjust Estimates. The labor estimates developed in step (3) are adjusted to account for special conditions of the project: the use of a higher-order language, hardware constraints, personnel qualifications, etc. Some of the most common factors for these adjustments are discussed in Sec. 7.3.

5. Estimate Cost Distribution. The adjusted labor costs are then distributed over the system development life cycle to facilitate the estimation of all other software costs (i.e., costs other than for technical labor). Table 7.3 presents the actual percentage division between major life-cycle phases for several major projects, as well as estimates proposed by numerous experts in the field.

Table 7.4 presents various related rules of thumb that were discussed by Morin.[9] The reader should note that this information was taken directly from tables presented in Morin's paper without the accompanying text. Since the accompanying text presents many evaluations, restrictions, and caveats concerning these rules of thumb, the reader is cautioned not to make use of them without first reviewing Morin's paper.

6. Estimate Other Software Costs. The steps up till now have addressed the technical labor required to develop the software. The sixth step is to estimate the cost of associated computer time, documentation, and management.

Computer hours may be estimated as a function of the number of man-months for software development or as a function of the number of instructions. Figure 7.3 illustrates these two relationships.[30] In general, estimates of four hours per man-month and 20 hours per 1,000 instructions are common in the literature.

TABLE 7.3

DISTRIBUTION OF SOFTWARE EFFORT

(Refs. 22, 29, and 30)

Source	Analysis and Design	Coding and Checkout	Integration and Test
Projects:			
Apollo	31%	36%	33%
DAIS	38%	15%	47%
Gemini	36%	17%	47%
NTDS	30%	20%	50%
OS/360	33%	17%	50%
SAGE	39%	14%	47%
Saturn V	32%	24%	44%
SETS/BL	42%	18%	40%
Skylab	38%	17%	45%
Titan III	33%	28%	39%
X-15	36%	17%	47%
Authors/Companies:			
Aron	30%	20%	50%
Boehm	34%	18%	48%
Brandon	32%	28%	40%
Brooks	33%	17%	50%
Farr	35%	18%	47%
GRC	30%	20%	50%
Krauss	47%	16%	37%
Raytheon (Business)	44%	28%	28%
RCA	32%	21%	47%
TRW (C&C)	46%	20%	34%
TRW (Scientific)	44%	26%	30%
Wolverton	46%	20%	34%
Approximate Averages	37%	20%	43%

TABLE 7.4

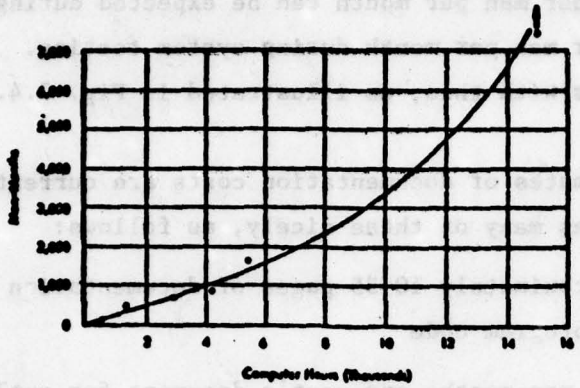
SOFTWARE DEVELOPMENT RULES OF THUMB

(as identified by Morin[9])

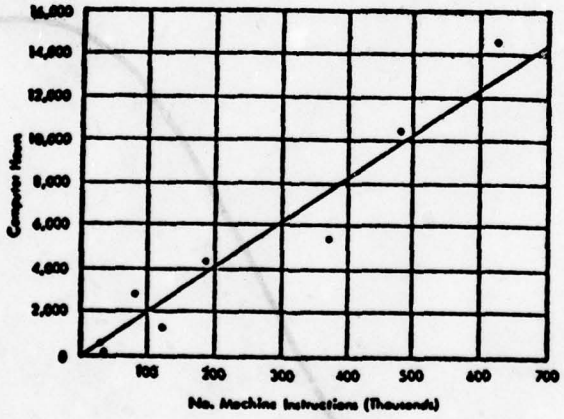
System Design Phase:	
Analyze Computer Program Production Requirements	3 to 9 weeks for a senior programmer
Analyze Similar and Interfacing Systems	2 to 10 man-weeks depending upon the nature of the project
Analyze Requests for System Change	Gross Estimate: 5 to 20% additional costs, 10 to 15% additional for schedules
Design the Total System	1 to 3 man-months depending upon conditions and the delays experienced
Design the Computer Program System	10% of the total man-months
Familiarize the User with the System Design	3 man-days per design document per agency contacted, plus allowances in elapsed time for travel
Indoctrinate Production Personnel	With analysts turning over the design to programmers, 1 month minimal formal training time per programmer. Without handover, training costs minimal, but hidden.
Coding Phase:	
Develop Program System Test Plans	1 man-month per 10,000 estimated machine instructions
Design Programs	1 man-month per 1000 - 2000 machine instructions. 1 man-month per 1000 instructions for large programs (over 30,000 instructions)
Design Program Files	1 man-month per 10,000 items
Establish System Files	1 man-month per 10,000 machine instructions for small projects. 2 man-months per 10,000 machine instructions for large projects (over 30,000 instructions)
Code the Programs	Gross Estimate: 1 man-month per 5,000 machine instructions

TABLE 7.4 (Continued)

Integration and Testing Phase:	
Learn Test Environment and Test Procedures	1 man-week per programmer
Test Individual Programs	Anticipate 1 error per 30 instructions About 20 percent of testing effort Approximately 1 error for each 125 instructions 3 to 10 trials for smaller programs, 100 distinct trials may be required for a larger program
Test Program Subsystem	Between 0 to 30 percent of total testing effort depending on number of subsystems
Test Program System	About 50 percent of total testing effort of 25 percent of the total effort
Implementation Phase:	
Outline User Documentation	Approximately two man-weeks per user's document, plus writing and editing costs of producing outlines and plans.
Conduct Demonstration Test	About one week elapsed time for a system of moderate size. About two man-weeks for a system of 10,000 to 20,000 instructions.
Analysis of Test Results	About one man-week for a system of 10,000 to 20,000 instructions.
Documentation of Test Results	Approximately two weeks for drafting test report.



Versus Man-Months



Versus Machine Instructions

Figure 7.3. Estimating Computer Hours (Ref. 30)

Aron[28] estimates that a rate of 7-8 hours per man per month over the last 50-70 percent of a project can be expected. He also estimates that 2-3 hours per man per month can be expected during implementation and 20 hours per man per month during system testing. Pietrasanta[27] generally agrees with this, as illustrated in Fig. 7.4.

Many estimates of documentation costs are currently available. Morin[9] summarizes many of these nicely, as follows:

- Approximately 10-35 pages of documentation per 100 lines of program code
- Two man-months per user's document for outlining
- Three to five pages (750 to 1250 words) per man-day drafting

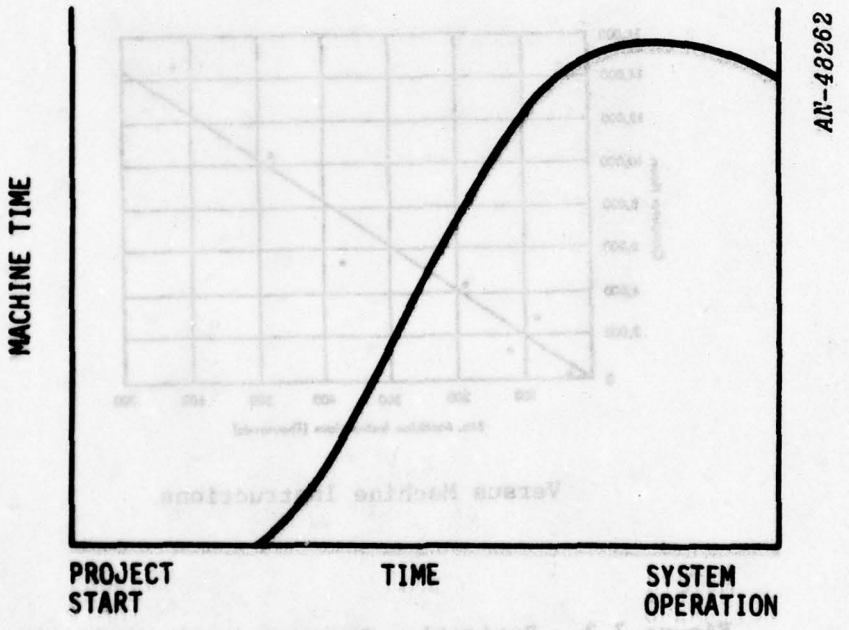


Figure 7.4. Pietrasanta's Machine-Time Requirements (Ref. 27)

- An average of 20 pages per man-day for technical review
- An average of 50 pages per man-day for editing
- An average of 15 to 20 pages per man-day for typing
- Two pages per man-day for illustrations (e.g., flow charts)
- An average of 10 pages per man-day to revise

In general, the cost per page of non-automated documentation is approximately $35-$50. It is estimated that this accounts for 10 percent of the total project cost.

Costs associated with the management of the project are estimated to account for 50% of the total project cost by Aron.[28] This is nearly the same as that previously estimated by GRC.[22]

7. Sum Costs. The seventh step is to sum the cost estimates developed to this point to determine the total software cost.

In conclusion, these seven steps are a generalized approach which attempt to incorporate the basic features of each model reviewed in the literature. In actuality, a particular model may include more steps and take them in a different order. The intent here is to provide an overview of the approach.

7.2.2 Estimates Based on Parameters Other Than Size

In general, software cost models which do not employ software size make use of steps similar to those just described. However, all estimates previously based on the number of instructions are instead based on such parameters as those shown in Table 7.5, which were developed in 1970 and were designated Automatic Data Processing Resource Estimating Procedures (ADPREP).[12*] This study was preceded in 1967 by the study

*The reader is reminded that the published data from this study was used extensively in Sec. 6.

TABLE 7.5

DEVELOPMENT AND INSTALLATION COST ESTIMATES[12]

Total Number of Input Type Transactions of ADPS input that normally are identified by a unique transaction code and/or unique input format.

Total Number of Different Output Formats of ADPS products.

Total Number of Record Types in Data Base where a logical record type is a set of logically related data fields independent of the physical manner of storage.

Total Number of Input Data Fields that are unique in content and/or format, e.g., if there is a data field for "name" on six different card formats, the number of unique data fields is one.

Average Number of Transactions per Month of Input, in thousands, originating outside the ADPS.

Average Number of Input Cycles per Month for processing input data.

Average Number of Characters, in Millions, in Data Base where the data base is a collection of files that contain unique information, are accessible to the ADPS, and are normally referenced or updated. Intermediate files are not included.

Net Growth per Month in the Size of the Data Base, in millions of characters.

Average Number of Characters per Month of Output, in millions, from the ADPS destined to users. Intermediate output of the ADPS are not included. Only nonblank characters are counted.

The Inclusion (or Exclusion) of On-Line Query Processing in the ADPS Design Coded as follows: 1 = inclusion; 0 = exclusion.

The Average Number of Records, in Millions, Contained in the Data Base Files: Intermediate files are not included.

Average Number of Output Cycles per Month for production generated by the ADPS for the user.

reported in Ref. 6, which also did not use number of instructions as a key input parameter.

The latest study which resulted in the development of estimating relationships not strictly employing software size was conducted in 1974 by Frederic.[10] This study resulted in the software estimating relationships presented in Table 7.6.

It is important to note that the salient feature of each of these methods of estimating costs is simplicity. A few simple parameters are estimated and then used in a single formula to calculate total software cost.

The inherent problem with these types of models is that it is difficult to impose adjustments for special factors. Also their accuracy is very poor, perhaps as a consequence of their simplicity.

7.3 ADJUSTMENTS

Past studies have identified several factors which strongly affect the cost of software and should therefore be accounted for by adjustments to a cost estimate. Several of the most significant factors are addressed below.

7.3.1 Complexity

Complexity is a function of the machine, the language, the type of application, the size of the project, etc. Aron[28] assessed complexity as easy, medium, or hard as defined in Sec. 7.2.1, step (1). Furthermore, he estimates that "medium" is approximately two times more difficult than "easy," and that "hard" is four times more difficult.

7.3.2 Familiarity

The extent to which assigned personnel are familiar with a particular application can have a significant impact on the amount of labor

TABLE 7.6

SUMMARY OF PROVISIONAL SOFTWARE ESTIMATING RELATIONSHIPS[10]

Inputs / Outputs	M, Total Man-Months Labor	D, Total Delivered Instructions (Thousands)	O, Total Operating Instructions (Thousands)	Air Threats (B)		Sea Threats (C)	
				S, Total Words Fast Storage (Thousands)	T, Targets Terminal-Tracked	S, Total Words Fast Storage (Thousands)	T, Targets Terminal-Tracked
Total Development Cost FY 73 $M	$0.0043(M)$	$0.01(D)^{1.18}$	$0.01(O)^{1.24}$	$0.0026(S)^{1.79}$	$0.30(T)^{1.88}$	$0.0043(S)^{1.79}$	$0.19(T)^{1.88}$
Total Man-Months Labor		$2.43(D)^{1.18}$	$2.52(O)^{1.24}$	$0.59(S)^{1.79}$	$69(T)^{1.88}$	$1.01(S)^{1.79}$	$45(T)^{1.88}$
Total Delivered Instructions (Thousands)			$1.03(O)^{1.05}$	$0.30(S)^{1.51}$	$17(T)^{1.59}$	$0.48(S)^{1.51}$	$12(T)^{1.59}$
Total Operating Instructions (Thousands)				$0.31(S)^{1.44}$	$14(T)^{1.51}$	$0.48(S)^{1.44}$	$10(T)^{1.51}$
Total Words Fast Storage (Thousands)					$14.30(T)^{1.05}$		$8.30(T)^{1.05}$

NOTES: (A) Costs assume $3,930/mm for labor, $77/computer hour, and 4.23 computer hours/mm.

(B) Use Air Threat column if maximum threat approach speed is in the 250-700 m/sec range.

(C) Use Sea Threat column if maximum threat approach speed is 50 m/sec or less.

(Ref. 10)

required. One large software house[29] estimates that unfamiliarity can decrease the normal coding rate by as much as 50 percent; or familiarity can increase it by as much as 30 percent.

7.3.3 Hardware Constraints

Hardware constraints were previously identified in Sec. 6.2.3 and measures were made of their impact. Figure 7.5 is another illustration of the effect of hardware speed and memory size constraints on the relative cost of software.[31] Our results (Sec. 6.3.3) showed a five to one growth in relative man-hour requirements, which compares reasonably well with Fig. 7.5, considering that utilization was approximately 98 percent.

7.3.4 Interactive Coding and Debugging

The use of interactive terminals to code and subsequently debug software can significantly reduce the effort required during the coding and checkout phase. It is estimated that productivity can be increased by as much as 20 percent through their use.[33]

7.3.5 Language

The use of high-order languages such as FORTRAN and JOVIAL has distinct advantages over conventional assembly-language coding. Hahn and Stone[34] developed the following estimates of software production rates (instructions per man-day) for several high-order languages and for machine language:

	Low	Average	High
FORTRAN	3.3	4.5	5.7
COBOL	4.5	5.8	7.2
JOVIAL	4.0	5.7	9.8
Machine Language	6.6	8.1	9.7

Each statement in a high-order language takes more man-hours to produce than a statement in machine language, but it results in the generation of several object-language instructions. The ratio of the

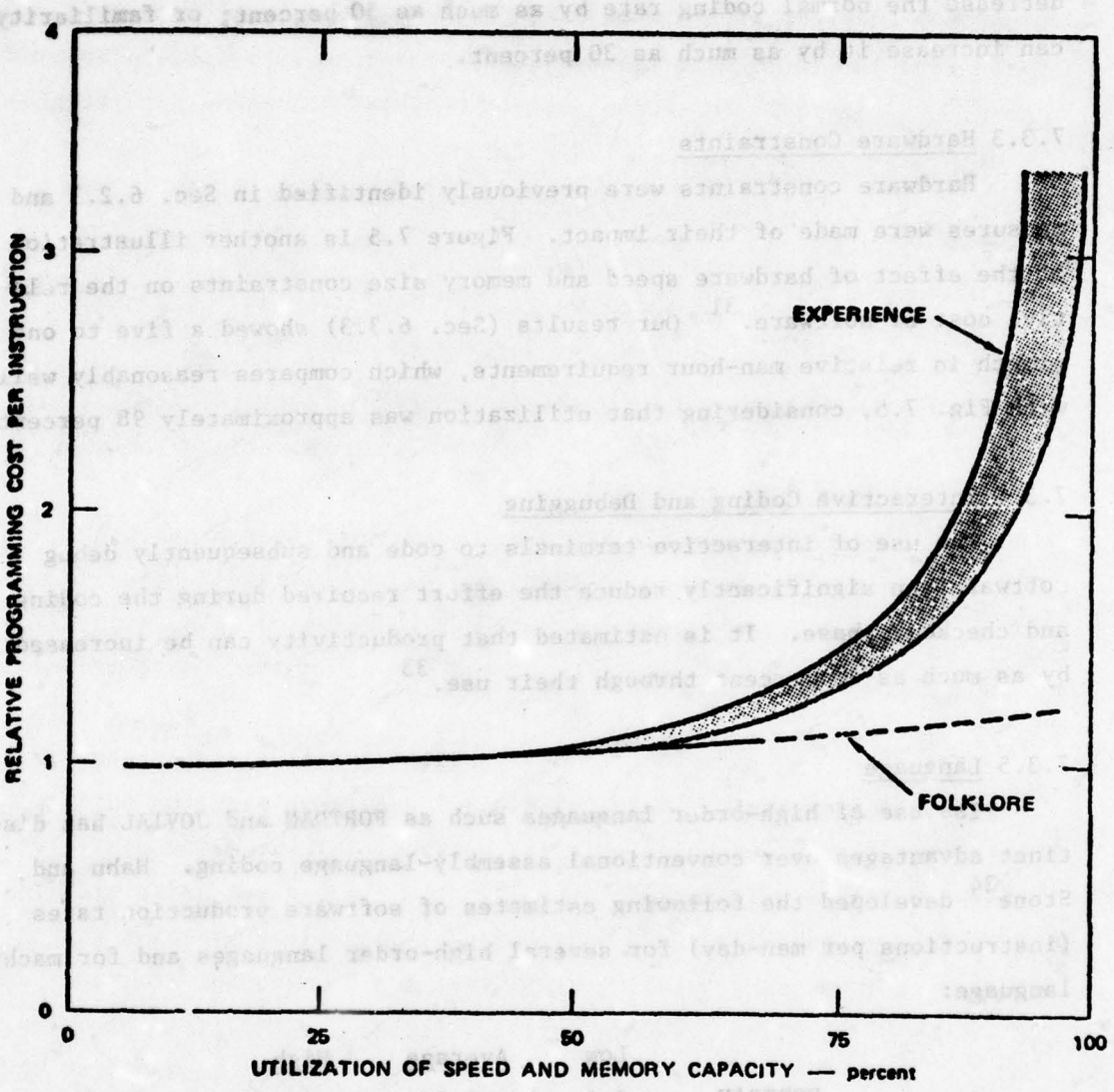


Figure 7.5. Hardware Constraints Cause Major Software Impact (Ref. 31)

numbers of object-code to source-code instructions is referred to as an expansion ratio. In Sec. 6.4, we showed that JOVIAL (on a CDC 3800) has an expansion ratio of 2.4. Other language/machine combinations are given in Table 6.2. Thus, use of a high-order language will reduce the cost per object-code instruction, even though the cost per source-code instruction is greater. For example, the object-language production rates for JOVIAL in the table above would be 9.6, 13.7, and 23.5.

This explains the findings of those who participated in the Government/Industry Software Costing and Sizing Workshop.[35] They estimated that the use of a high-order language could halve the cost per instruction. In addition, a major software house[29] estimates that the cost of software design, coding, and checkout can be reduced by as much as 80 percent through the use of high-order languages. They also estimate that the use of macros when coding in assembly language can result in a saving of up to 10 percent.

7.3.6 Personnel Qualifications

The qualifications (as well as the quality) of the persons assigned to a project affect the amount of labor required. Participants in the Costing and Sizing Workshop[35] estimated that there is approximately a 5:1 variability in labor requirements depending upon a person's skill. However, this is a very difficult parameter to determine early in the development cycle.

7.3.7 Personnel Utilization

The amount of time actually directed towards the completion of a project is not 100 percent of time assigned (i.e., utilization is less than 100 percent). Actual utilization is affected by such things as turnovern, computer down-time, coffee breaks, etc. We estimate that a 55 to 85 percent utilization rate is not unrealistic for most cases. However, when a new computer system, operating system, etc. is involved in a software development project, the utilization rate can drop below 50 percent.

7.3.8 Staff Size

Brooks[20] contends that staff size is a highly critical consideration because of the required interactions between personnel. He goes on to quantify the amount of required communications as follows:

$$\text{Communications Time Factor} = \frac{n(n-1)}{2}$$

where

n = number of personnel

This formula indicates that, for example, four people require twice as much communication as three.

Another source[29] estimates that a staff consisting of 6-10 people can be very effective, but that a staff of more than 20 people can require as much as three times the time and effort because of the communication problem.

7.3.9 New Versus Existing Software

The software developed for a particular project normally consists of (1) new software, (2) modified existing software, and (3) existing software which may be used without modification. The costs of developing new software have been discussed throughout this report. It is obvious that the cost of using existing software that requires no modification is small. The cost of modifying existing software, however, is not well known.

The use of existing software, whether modified or not, significantly impacts coding, integration, and testing, and particularly analysis and design. One source[29] estimates that it reduces the analysis and design effort by as much as 80 percent, and the coding, integration, and test effort by as much as 20 percent.

7.3.10 Requirement Changes During Development

Requirement changes are an important part of software cost variation which to date had not been addressed quantitatively with any success. In Section 6.3.4, we used the ADPREP[12] data to develop a relationship between man-hours, lines of code, and number of changes. The relationship had an R^2 of .65 and is shown below.

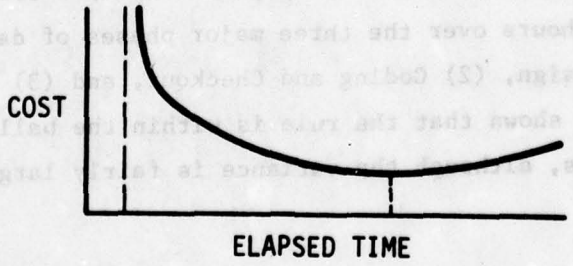
$$MM_{Development} = 6.6 + .678\ (\text{object instructions}/1000) + .110\ (\text{object instructions}/1000 \times \text{no. changes})$$

It is recognized that the timing of a change is important in determing its impact. However, data did not permit exploring that dimension. The preceding equation estimates an average penalty (increaed man-months) for each change.

7.4 TIME-PHASING THE ESTIMATE

An estimate of the total cost of the software by itself is usually not sufficient for use by the Program Office. Costs must be spread over the entire development life cycle to facilitate the preparation of a time-phased budget. As a result, an estimate of the total time allowed to do the job must be made.

Although specific methods of calculating the ideal time for a software effort are scarce, the literature does agree on the general shape of the curve which plots software cost versus elapsed time. Meyer[36] developed the following curve:



This curve shows that there is an optimum project time span that minimizes costs. As the time decreases below this optimum time, costs increase and eventually rise to infinity. As the time increases above the optimum, costs increase due to the extended costs of a minimum level of personnel, delays, ECPs, etc. The problem is that it is extremely difficult to estimate this optimum time span.

Putnam[37] notes that manpower costs have been inversely proportional to the square of time for those projects which he has analyzed. However, it appears that he has concentrated on projects which were time-constrained from the start and has, therefore, not seen the increasing part of the curve.

Meyers' estimating technique[36] assumes a project duration in the range of 17-30 months which is further divided as follows:

- System Design (5-9 months)
- Programming (7-12 months)
- Integration and Test (5-9 months)

Wolverton[19] also addresses scheduling, but in a fashion that derives the project time span after the project costs and man-months have been spread over the development life cycle.

Once total development time has been defined, several rules can be used to spread the cost. Putnam[37] has studied the distribution of resource consumption, defining the distribution of man-hours over time, as well as the different phases. The popular 40/20/40 rule can also be used to spread man-hours over the three major phases of development: (1) Analysis and Design, (2) Coding and Checkout, and (3) Test and Ingegration. Table 7.3 shows that the rule is within the ball park of many recent projects, although the variance is fairly large.

Furthermore, in Sec. 5, we show evidence that the ratio of testing to coding decreases with program size. Relationships using the NASA data showed that as the program size increased to 25,000 lines of source code, analysis and design remained around 41 percent while coding increased from 17 to 21 percent and testing decreased from 42 to 38 percent.

If the relationships between the phases are developed, then they can be used to spread man-hours over the development life cycle, using the time-phasing of activities defined by Putnam[37]. In this case, changes in distribution as program size increases are automatically incorporated, and the use of the 40/20/40 rule is avoided.

7.5 SOFTWARE OPERATIONS AND SUPPORT

While a significant amount of research in the area of software cost estimating has been devoted to the software development process, much less attention has been given to the costs associated with software operation and support. Cost data for defense projects indicates that operations and support costs can significantly impact overall system costs. Brooks[20] estimates that the cost of maintaining software is approximately 20-40 percent of the cost of the initial software development. In addition, cost of making changes per object instruction (of the altered code) can run from 10-100 times the original cost of producing the line of code.[38]

As discussed at some length in Sec. 6.5, much confusion revolves around what one calls "maintenance." Error correction, the only portion of maintenance that can be attributed to the original code, accounts for less than 1/3 of the technical effort of software maintenance projects that were reviewed. The remaining time is devoted to product improvement.

Some models have been developed to predict staffing requirements for maintenance. For example, Planning Research Corporation[12]

developed such a model in 1966 based upon ADP data. Various graphs presented in their report provide estimates of (1) the required number of maintenance personnel, (2) the required number of operations personnel, and (3) the dollars per month for hardware costs of program maintenance. The inputs required to determine these estimates are (1) characters per month of input volume, (2) number of input fields, (3) characters per month of output volume, (4) number of output fields, and (5) characters in the data base.

In Sec. 6.5, we presented estimating relationships developed from the ADPREP[12] data base which suggested much lower costs for maintaining programs written in high-order languages. However, the R^2 values are low, so the relationships should be used cautiously.

The trouble with both these results is that they present staffing models which include product improvement. It is our conjecture that productive time not spent correcting errors will be used in defining product improvements. Thus, requirements for original staffing tend to become self-fulfilling. As a result, we have concentrated our efforts on predicting requirements for error correction only. A model has been developed in Sec. 6.5 which relates the cumulative number of problems to the number of cards, number of design changes (ECPs), and the number of previous errors corrected. In effect, we assert that number of errors would diminish over time if there were no design changes. The resulting equation had an R^2 of .64 and is given below:

$$\begin{aligned}\text{Cumulative No. of Problems} &= .396 + .368\ (\text{No. of source cards}/1000) \\ &+ .686\ (\text{No. of source cards}/1000 \times \text{No. of Design Changes}) \\ &+ .482\ (\text{No. of source cards}/100 \times \text{No. of Previous Problems Corrected})\end{aligned}$$

Also, we examined the impact of V&V upon cumulative errors after acceptance testing. Data presented in Fig. 6.13 supports the thesis that V&V discovers 25%-50% of the potential errors.

7.6 CONCLUDING REMARKS

In this section we have identified and briefly described some of the more important techniques from the literature. We have also incorporated the more useful results from this study. We hope this information will be useful to Program Offices, although we again remind the reader that there is little precision in the estimating techniques developed to date. Therefore, each technique must be used with caution.

The inability of the cost estimating community to generate more precise techniques has been largely due to the difficulty in obtaining a large set of consistently defined data. As a consequence, trade-offs within developments have not been examined. If a cost reporting system is implemented by the Air Force, such trade-offs will be more visible. Elements of such a reporting system are described in the next section.

It has been recommended that a consistent estimating technique be used so that departures from the estimate can be recorded and the estimating techniques improved. In particular, Devenny[1] recommends the use of the Price model, for which a software component will soon be available. It is our opinion that a consistent estimating technique alone is not what is necessary to develop better cost estimating techniques. What is even more important is the consistent gathering of actual data, and a sound, explicitly analytic approach to software cost estimating that can be adapted to new procedures and technological developments.

Basically, not enough is now known to standardize on one technique. However, if standardization is required we recommend that it should be a technique that is published and can be evaluated by the

software community. Since the Price model is proprietary and therefore not published, we believe that it should not be selected as the standard. It can, of course, be of value as one of several estimates used to check for reasonableness.

8 REPORTING SYSTEM ELEMENTS

8.1 INTRODUCTION

The second major goal of the study was to develop a recommended set of software reporting system elements around which a uniform data collection system can be designed. Software resource-consumption data can then be collected from various Air Force developments consistently and unambiguously. This data can ultimately be used for developing better software CERs.

GRC was asked to identify only the elements themselves. The actual reporting system (forms, formats, information flows, etc.) was beyond the scope of our study. Hence, this section only addresses the definition of the elements and the frequency of data collection. However, before discussing these elements, it is useful to restate the goals of the reporting system and the resulting principles which have guided the selection of elements.

The four primary goals of the reporting system are as follows:

1. Provide for Cost Control by Program Offices. The main short-run benefit of the reporting system will be the visibility it affords the Program Offices for controlling software cost. Currently, a Program Office receives cost information through the Cost Performance Report, a deliverable of the Contract Data Requirements List. The information reported is very aggregated. Some projects have only one line for software cost data; others have more, provided that more than one Computer Program Configuration Item (CPCI) has been defined and provided that the level of cost reporting extends down to CPCIs. Also, reported costs often do not include software engineering costs during analysis and design, or formal testing costs. This inconsistency in definition is a major contributor to apparent cost variability.

2. Provide a Data Base for Better Cost Estimates. The long-run benefit of the reporting system will be the development of a data base on software costs with consistent definitions. The usefulness of such a data base for CER development is evident. It should be noted that several previous studies (as early as 1966) have defined such a software cost reporting system.[2,3,7] If some form of these systems had been implemented, we would have a fairly good data base today. Therefore, the exact form of the data base to be developed is probably not as important as the implementation of a uniform data collection effort.

3. Have a Reasonable Chance of Wide Acceptance by Contractors. Perhaps the earlier efforts were not implemented because they asked for too much detail. For example, the referenced studies require that redesign and recoding (in response to a discovered anomaly) be recorded separately as additional design and coding, respectively. This is very difficult to do and leads to an arbitrary allocation of time.

There is always a tradeoff between (1) the scope of the detailed data collection effort required for future cost estimation, (2) the aggregated information necessary for cost control and, (3) the even more aggregated information that contractors will readily provide. The wider the difference between (1) and (3), the more likely that contractors will resist providing the information.

4. Be Based on Measurable Items. Detailed costs should be gathered around typical contractor work packages and not artificial groupings of costs. Furthermore, these work packages should be standardized so that comparisons can be made between programs. However, the time of managers and other indirect personnel, who are not assigned to specific work packages, should not be artificially allocated to these work packages. Data should be collected in aggregate and allocations should be made later if desirable for some analysis.

Using these goals, we developed the following three principles which have guided our selection of elements:

1. Resource consumption should be related to progress towards the completion of software. This is the single most important item for cost control. Currently, it is assumed that if 30% of the hours have been spent, then 30% of the work has been done. Cost overruns attest to the unsuitability of this idea.*

2. The level of detail should not be greater than what is specifically needed for cost control and the prediction of future resource consumption. This still may be too detailed for contractor acceptance, and tradeoffs have been made in element definition which sacrifice some information for this reason.

3. Data requested should relate to the actual development process and not require artificial allocations. This is self-evident and a central focus of the definition of a Work Breakdown Structure (WBS).

The reporting system elements were defined using these principles as a guide. Conceptually, cost elements have three dimensions (Fig. 8.1): the software end items being developed, the kind of resources being consumed, and the phase of the life cycle in which the resource consumption takes place. All three dimensions are important for cost control and cost estimation.

---

*Reference 1 plots cost histories for several ESD software developments against original estimates. All show cost growth, some by more than 100%.

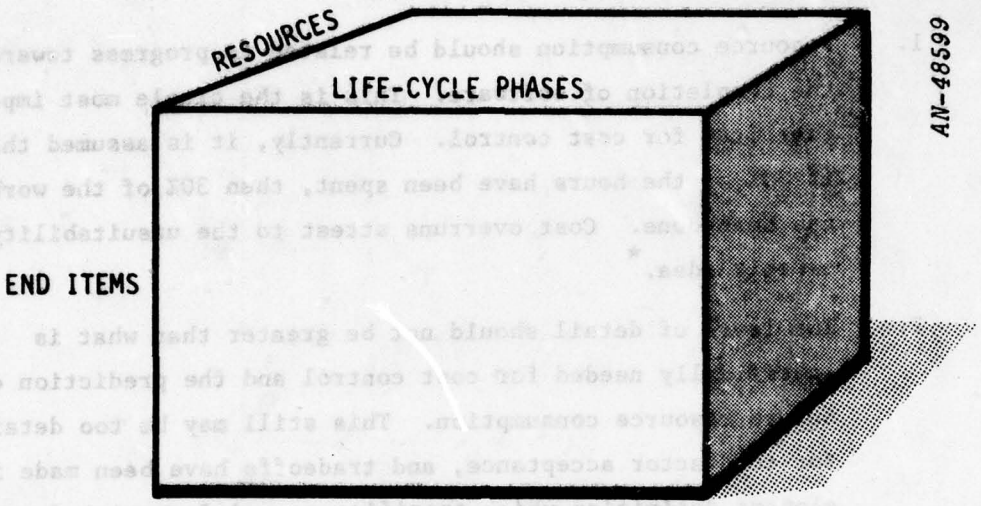


Figure 8.1. Cost Element Dimensions

In this section, each of these dimensions will be defined. First, we will define the phases of the life cycle (Sec. 8.2). Because of its importance, the measurement of progress towards completion of the development phase will then be discussed, and a set of data elements describing the product and its relation to the phases of the life cycle will be specified (Sec. 8.3).

This will be followed by our definition of a standardized set of software end items, the second dimension of Fig. 8.1 (Sec. 8.4). Next, we will define the resources consumed and a set of resource-consumption data elements (Sec. 8.5); the personnel and computer resources are discussed more fully in Secs. 8.6 and 8.7 respectively. Section 8.8 will consider the feasibility of implementing this reporting system.

AD-A053 020 GENERAL RESEARCH CORP SANTA BARBARA CALIF F/G 9/2
COST REPORTING ELEMENTS AND ACTIVITY COST TRADEOFFS FOR DEFENSE--ETC(U)
MAY 77 C A GRAVER, W M CARRIERE F19628-76-C-0180
UNCLASSIFIED CR-1-721-VOL-1 ESD-TR-77-262-VOL-1 NL

3 OF 4
AD
A053020

The section will be concluded with a discussion of two special topics: in Sec. 8.9, how to handle changes in requirements (ECPs); and in Sec. 8.10, contract work that precedes full-scale development.

Throughout, the reader should keep in mind that the development of software for a weapon system is not easily separated from the development of the rest of the weapon system. Occasionally, software is the end item itself, or a separate subcontractor is responsible only for software. In these cases, software cost definition and data collection is relatively easy. However, in general, software will be only a portion of the development contractor's work.

8.2 LIFE-CYCLE PHASES

The first dimension, that of the life-cycle phases, is the same as the milestone definitions discussed in Section 2.1: analysis, design, code and checkout, internal test and integration, qualification test, installation, and maintenance (operations and support).* A simplified version of the software life cycle is presented in Fig. 8.2, showing the relationships between phases, the relationship of phases and milestones, and finally, the level of information available at each phase. Fig. 8.2 is a simplified version of Fig. 2.3, specifically excluding changes in requirements (ECPs) during software development.

*The specific tasks that make up each phase are identified and assigned to specific reporting-system elements in Sec. 8.6. These reporting elements will sometimes bear the same title as a life-cycle phase. When this happens, the activity definition rather than the milestone definition is intended. A table is given (Table 8.14) which clarifies the two different uses of these words.

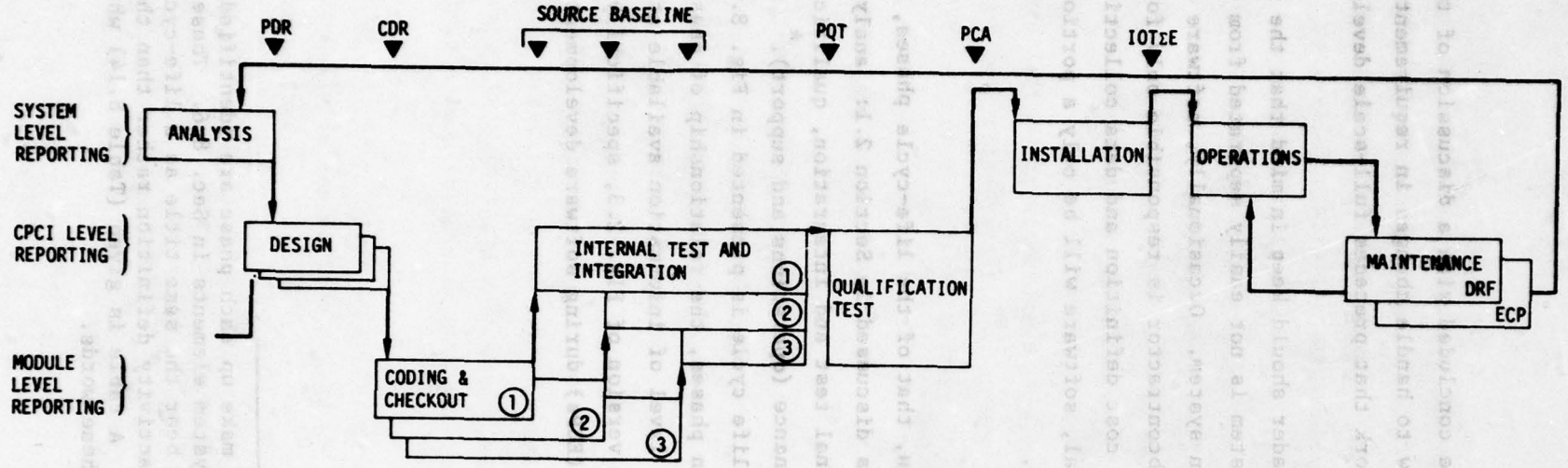


Figure 8.2. Life-Cycle Phases and Level of Reporting

To review: the software development life cycle begins with the Analysis phase, during which the software requirements are analyzed and refined, functions are defined and allocated to specific software end items, and Computer Program Configuration Items (CPCIs) are defined. Each CPCI is specified in detail, including test requirements, and these requirements are documented and baselined in the Part I (Development) Specification document which is reviewed and approved at (or preferably before) PDR.

Each CPCI is further disaggregated during the Design phase, with functions being assigned to Computer Program Components (CPC), modules, and subroutines, and detailed flow diagrams being developed which describe their relation. Part II (Product) Specifications document this work, which is reviewed at CDR. Part II Specifications are not baselined at this time; they are updated periodically through the development, and finally baselined at the Physical Configuration Audit (PCA).

Coding and Checkout of source code then proceeds with individual modules, routines, etc., being assembled, compiled, and unit-tested at different times. As source decks are completed, they are integrated and tested with other modules during the Internal Test and Integration phase. Redesign and recoding are accomplished as required until the entire CPCI is ready for the Qualification Testing phase, which completes the development. Software is then reproduced, installed, tested, and modified (if required) at operational sites during the Installation phase. The software is maintained with error correction in response to error reports (DRFs), and changing requirements (incorporated through ECPs) which form a new software development.

From this description it is easy to see that different levels of information are available throughout the life cycle. During analysis and again during installation and operations, information is generally available only on the software as a whole, and is difficult to separate

from information on the system hardware. During the other phases, the software can be divided into end items (CPCIs) as specified by the Part I Specifications, and resource consumption can be tracked at this level. Further detail is available with the definition of Computer Program Components (CPCs) and modules in the Part II Specifications. Although some costs can theoretically be tracked at the module level during coding and debugging, they become less trackable during Internal Test and Integration and Qualification Testing. In addition, some costs can be tracked to the module level during maintenance.

Phases are defined by milestones: Preliminary Design Review (PDR) marks the end of Analysis; Critical Design Review (CDR) ends Design; the start of Preliminary Qualification Test (PQT) completes Internal Test and Integration; the completion of the Physical Configuration Audit (PCA) marks the end of the development process.

It would greatly simplify reporting if the phase boundaries marked the end of activities (e.g., if CDR ended the design activities). In fact, however, activities cross these boundaries. While it may be possible to capture some of this overlap (e.g., continuation of initial design until it is first baselined), attempts to do so in detail may cause reporting difficulty and result in either the reporting of estimated times or rejection of the whole reporting scheme by contractors. Separating redesign from recoding when correcting a bug in Internal Test and Integration is a good example of this problem.

It probably will be possible to capture planned overlapping of activities, especially in the earlier phases of the life cycle. Accordingly, we recommend that:

1. Specific analyses which have not been completed by PDR but are specifically approved at PDR for later completion should be assigned to the Analysis activity.*

2. Initial design activities which have not been completed by CDR but are specifically approved at CDR for later completion should be assigned to the Design activity.*

3. Coding that is initiated before CDR should be assigned to the Coding and Checkout activity,* provided that it was specifically authorized at PDR.

Capturing this information along with the PDR and CDR dates will show the cost consequences of planned overlaps. It is recognized that, by definition,[11] PDR should end the analysis activities, CDR should end design activities, and coding activities should not begin before CDR. However, the real world does not work this way. It is better to have approved overlap and know what is happening, than unapproved overlap due to regulations. It is probable that some overlap is cost-effective.

Separating redesign and recoding activities in later phases of the life cycle is another story. It is very difficult to make this separation, and we recommend that no attempt be made. This is a primary difference from previously recommended reporting systems.[2,3] We speculate that one reason that these systems were not implemented could have been the requirement to separately identify redesign and recoding in these later phases.

---

*When capturing overlap, reporting elements will be used. These reporting elements may use the same terms as the life-cycle phases, but relate to activities rather than milestones. The terms have been used interchangeably in the literature. In Sec. 8.6, the distinction between the two uses is clarified. New terms should perhaps be invented for one set of these definitions.

We also recommend the establishment of a new milestone. This milestone will occur whenever a module has been coded and debugged, and its source deck is baselined for the first time. In effect, this becomes a "moving milestone" and will provide an almost continuous measure of product completion. The establishment of the source-deck baseline is enhanced if a Program Support Library is established for the project.* Costs and man-hours expended on a module will be reported against the coding and checkout phase until the module is baselined, and thereafter will be reported against Internal Test and Integration. Contractors have reacted favorably to this idea.

We feel that reporting costs and man-hours with respect to these life-cycle phases will not require artificial allocations.

8.3 SOFTWARE PRODUCT INFORMATION AND PRODUCT COMPLETION STATUS

A clear, concise, and accurate description of the software product to be developed is required for any project on a historical data base. Too often, pertinent information about the software product is lost or ill defined. Attempts at relating resource consumption to ill-defined input variables are doomed to failure. Lack of precision with respect to these input variables is often the result of depending upon human memory of what occurred during the development, years afterward. Collecting quantitative, descriptive data on the software development can alleviate this uncertainty.

An equally important problem is to relate resource consumption to software completion status. We believe this is the single most important resource reporting requirement for cost control. Currently, the Program Offices must rely on reports of the percentage of man-hours expended to ascertain the progress of development. However, the

*Ref. RADC TR 74-300, Structured Programming Series

examples in which percentage of man-hours expended meant far less than the equivalent percentage of software developed are too numerous to place much faith in this method. Yet the Program Offices currently have no other measure of product status between CDR and the beginning of Qualification Test.

We recommend that software description and status information be collected during the software life cycle as shown in Table 8.1. Specific information to be requested is shown in Tables 8.2 and 8.3.

During the early part of the software development, before CDR, product descriptions will be estimates. The level of product definition will continue to be refined, but no actual code will have been completed.

At the beginning of the contract, we suggest recording the original schedule, purpose, and target hardware in a permanent data base. These data will be reported for the system (segment) as a whole, since CPCI definitions should not have been baselined this early in the system-acquisition cycle.

At PDR, CPCIs will have been selected and baselined along with the Part I Specifications. We recommend that these CPCIs be picked from (or consistent with) a standardized list of software end items called Computer Program Components (CPCs).* It is important to gather resource and product information from a standardized list of end items so that CERs can eventually be based on the CPCs, much as a hardware cost estimate is now made for the component parts.

*A suggested list is presented in Sec. 8.4.

TABLE 8.1

SOFTWARE PRODUCT DESCRIPTION AND STATUS

Report	Data to be Included	Type of Data		Reporting Level		Report Timing
		Est.	Act.	CPC	Module	
Computer Program Development Plan	A,B*	X		(Total)		At contract award
CPC Definition and CPCI Update	C,D,E,F*	X		X		At PDR
CPC Update	C,D,E,F*	X		X		At CDR
Module Definition (Date of baseline and man-hours for coding)	F*	X			X	At CDR
Baseline Source Deck	F,H*		X		X	When module is baselined; monthly during Coding and Internal Test and Integration.
Start of Qualification Test	C-H*		X	X		At PQT (first)
Product Change	Table 8.3		X		X	Monthly during Qualification Testing
Completed Product (PCA)	Table 8.2		X	X		At PCA
Installation	Changes to 8.2 for specific site		X	X		At IOT&E
Maintenance Changes	Table 8.3		X	X		Monthly during maintenance
Model Update	Table 8.2		X	X		Whenever a new model (ECP) is incorporated during maintenance.

* Table 8.2, paragraphs so lettered; items marked (I) only.

TABLE 8.2

SOFTWARE PRODUCT DESCRIPTION

(I) marks data to be included in interim reports
(reports 1-5 of Table 8.1)

A. Descriptive Information for the Entire Software Development

(I) Purpose

Definition of CPCIs

Definition of CPCs

(I) Description of Target Hardware

B. Milestone Information for the Entire Software Development (I)

For each of the following, the date estimated at contract award and the actual date:

PDR

CDR

Part I Specifications baselined

Part II Specifications baselined

First coding

Initial design completed

Initiation of qualification testing

PCA

C. Descriptive Information for each CPC

(I) Name

(I) Brief statement of purpose

(I) Parent CPCI (if CPC is not a CPCI)

Names of programs required for testing

Any automated V&V tools utilized during development

Target hardware (if different from that of other CPCs)

TABLE 8.2 (Continued)

D. Development Method for Each CPC

(I) Programming language or languages

Host assembler and/or compiler if other than target machine

Host computer if other than target machine

Structured programming technique (if any)

Separate V&V contract (if any)

Size of object-code space constraint (if any)

Execution-time constraint (if any)

Delivery-date constraint (if any)

Budget constraint (if any)

Percentage of the implementation based on requirements adapted from existing system (if any); identification of system

E. Milestone Information for Each CPC (I)

For each of the following, the date estimated at PDR (CDR for † items) and the actual date (where not the same as in B):

PDR

CDR

First module baselined †

Last module baselined †

Last assembly and/or compilation prior to Qualification Testing †

Last assembly and/or compilation prior to PCA †

TABLE 8.2 (Continued)

Part I Specifications baselined

Part II Specifications baselined

First coding

Initial design completed

F. Code Size Measure for Each CPC (I)

For each of the following, the PDR estimate, the CDR estimate, the actual at first baselining, and the actual at PCA:

Source Code - Total lines (by programming language)

New source code

Adapted from existing code

Existing code (unadapted)

Object Code - Total lines

Compiled from source code

New object code

Adapted object code

Existing object code (unadapted)

G. Code Changes Since Baseline for Each CPC

(I) Number of compilations prior to first qualification testing

(I) Number of assemblies prior to first qualification testing

TABLE 8.2 (Continued)

Number of compilations prior to PCA

Number of assemblies prior to PCA

Note: This information could be expensive to track. Require only if an automated data retrieval system (library) is in use.*

H. Code Structure for Each CPC (I)

For each of the following, record the actual number of items at first baselining and at PCA:

Modules

Non-overlapping fields in the data base

Input formats

Output formats

Unconditional branches

Conditional branches

Interfaces other than data I/O

Note: The ultimate purpose of these measures is to define the program complexity, and then relate it to cost. The above represents our best judgement of the minimal list required for this purpose. Much has been written about code complexity, and a more exhaustive list has been compiled by IBM.[7] The composition of this portion of the required data should be the focus of continued study.

I. Other Size Measures for Each CPC

Number of pages in Part I Specifications

Number of pages in Part II Specifications initially approved

---

*Program support library--see Structural Programming Series, RADC TR 74-300

TABLE 8.2 (Continued)

Number of pages in Part II Specifications at PCA

Number of test procedures in qualification test procedures initially approved

Number of test procedures actually executed during PQT and FQT

Execution time (as compared to constraint)

TABLE 8.3

REPORTING ITEMS FOR CHANGES TO CODE

Number of errors detected

Number of errors corrected

Number of compilations

Number of assemblies

Total lines of source code added

Total lines of source code modified

Total lines of source code deleted

Change to object-code size due to source-code changes

Total lines of object code added with octal correctors

Total lines of object code dropped (without source-code change)

Reason for changes: coding error, design change, new requirement, caused by another software fix.

The relationship between CPCIs and CPCs is discussed in the next section. Suffice it to say now that CPCIs are generally (but not always) more aggregated than CPCs and that their relationship should be known at PDR.

At PDR, the CPCs should be defined (at least their interfaces and performance requirements). This definition includes the following from Table 8.2:

Section C: CPC name, purpose, and relationship to CPCI.

Section D: Programming language, and constraints on size, speed, delivery date, and budget.

Section E: Milestone estimates for CPC development (excluding † items)

Section F: Code size measures

By CDR, the product definition will have been expanded so that modules are defined for each CPC. At this point, the PDR estimates should be updated for each CPC, including milestone estimates for (†) items in Section E of Table 8.2. In addition, for each module we should have a detailed estimate of its size, Section F information from Table 8.2, and an estimated date for baselining the source deck (i.e., when source deck is coded and checked out for the first time), together with an estimate of man-hours to develop the baselined source code. This will provide the information necessary for charting software development progress through the Coding and Checkout phase.

Progress towards initial completion of the source code should be continually assessed. This can be accomplished by using the "moving milestone". Each module source deck should be baselined, and the completion data should be reported along with the size of the module (Sec. F, Table 8.2). It would also be useful to record the composition of the code (Sec. H, Table 8.2).

The continued reporting of source-code completion will provide an almost continuous measure of product completion (e.g., 30% of the source code has now been written and debugged). Furthermore, actuals can be checked against estimates to see if man-hour consumption is in line with estimates, product size is within projections, and coding is on schedule.

In practice, this reporting requirement may be deemed excessive by the contractor. However, requiring delivery of the source code as it is completed can hardly be objected to and will provide all required information. Air Force project monitors can analyze each source deck, extract the information, and verify its completion. Furthermore, making this a requirement will provide great incentive to the contractor to stay on schedule.

Although not appearing in Table 8.1, changes to the baselined source code for each module ideally should be tracked continually, beginning with baselining and continuing until the entire CPCI is ready for FQT. Reported information would include for each module the number of errors detected, number of times code changes, total lines of code added and deleted, and current module size. Such information would allow the Program Office to spot problems quickly.

Tracking changes is technically feasible, if the library concept is being used, but would probably be considered excessive by contractors. We therefore do not recommend the regular reporting of this information. However, the capability and right to sample library records periodically would accomplish the same control objectives. Properly done, it could also provide enough information for cost prediction.

Prior to Qualification Testing, a complete description of each CPCI should be prepared. This would include information regarding the actual code developed (items marked (I) in Secs. C-H of Table 8.2) and

previously reported as estimates, together with the number of compilations and assemblies performed during Internal Test and Integration.

During Qualification Testing, the Government can capture changes. Accordingly, we recommend that the data of Table 8.3 be captured for each CPC on a monthly basis. This will provide a great deal of visibility during this critical period.

At PCA, a detailed set of measurements describing the product should be completed and included in the project history. We recommend that the product-description data elements identified in Table 8.2 be collected. This need not be a contractor requirement since the analysis could be performed by Air Force personnel on the delivered product.

If changes to the code are required during installation, these changes ought to be noted by updating the CPCI description data for the affected site.

Finally, during Maintenance, changes to the code in response to errors should be recorded monthly. We recommend that the types of data identified in Table 8.3 be reported. New data can be prepared which summarizes the monthly changes by CPC whenever a new version (in response to an ECP) is incorporated into the system.

Accumulating this product information should not only provide a significant improvement in control, but will also provide the product-description information necessary to develop good CERs.

8.4 END ITEMS

The second cost-element dimension (Fig. 8.1) is the end item. As previously mentioned, it is important to establish a standardized set of software end items which can be used to define the software development. Then, if resource consumption and product information are

collected against this list for a number of developments, sufficient information will be accumulated to develop CERs for the software end items. Ultimately, software can be described by its parts, much like hardware, and estimates of the total package may be based upon estimates of the individual parts. Also, such an end-item list will give the Program Offices greater visibility into the software development. One budget line item for software is simply not enough visibility for cost control.

The end items should be functionally oriented so that they can realistically form contractor work packages. New requirements may require additions to this end-item list, but at least the standardized list will provide a basis for gathering reasonably comparable information across development programs.*

The end items have been designated Computer Program Components (CPCs) and it is anticipated that CPCIs will be made up of groups of CPCs. Standardizing a CPCI list, which is conceptually similar, was viewed as too restrictive. CPCIs should be based upon the level of control identified in the Part I Specifications and specific test plans. Occasionally, for high-risk items, a CPCI could be a part of a CPC. For example, a particular JOVIAL compiler may be both high-risk and critical to the development, and it may be desired to make it a separate CPCI. In this case, if "all compilers" is a CPC, as we are suggesting, then at least two CPCIs should be defined; one for the JOVIAL compiler and one for the remaining compilers.

Our suggested CPC list is given in Table 8.4, with definitions of terms.

---

*If a particular end item is being developed on more than one computer system, then it should be split into separate end items and tracked separately.

TABLE 8.4

COMPUTER PROGRAM COMPONENT LIST

1 SUPPORT SOFTWARE

Supports the development, testing, operation, and maintenance of the applications software.

1.1 EXECUTIVE

Coordinates the operation of the computer hardware for a particular application, often referred to as the "operating system."

1.1.1 Computer Resource Manager

Performs the overall control function for a computer system in a supervisory mode. Includes such features as an I/O supervisor, a permanent file manager, a CPU scheduler/memory manager, etc.

1.1.2 Computer/Peripheral Interface

Coordinates and controls the internal transfer of data between the CPU and peripherals: disks, tape drives, card readers, printers, etc.

1.1.3 Computer/Operator Interface

Coordinates and controls the transfer of data between the CPU and data display/entry devices which constitute the man-machine interface.

1.1.4 Computer/Computer Interface

Coordinates and controls the internal transfer of data between CPUs within the system.

1.1.5 Computer/Special Device Interface

Coordinates and controls the internal transfer of data between the CPU and such special devices as sensors, communications multiplexers, etc.

1.1.6 System External Interface

Coordinates and controls the transfer of data between the system and other systems external to the system.

1.1.7 System Failover and Recovery

Coordinates and controls all internal error handling, system failure reconfiguration, and recovery procedures required to continue system operation, even if in a degraded mode.

1.1.8 Performance Monitoring

Collects appropriate system performance data (on-line) for subsequent reduction and analysis (off-line).

1.2 UTILITIES

Perform auxiliary functions (off-line) such as reading cards, printing, transferring files from device to device, performing housekeeping functions required for file residence on mass storage devices, etc.

TABLE 8.4 (Contd.)

1.3 LANGUAGE PROCESSOR

Converts programs in one language to those in another.

1.3.1 Assembler

Operates upon a symbolic-language program to produce a machine-language program. The symbolic instructions generally correspond one-to-one with machine instructions. An assembler does not make use of the logical structure of the program.

1.3.2 Compiler

Operates upon a symbolic-language program to produce a machine-language program, using extensive syntactic analysis. The compiler makes use of the logical structure of the program and generates more than one machine instruction for each symbolic instruction.

1.3.3 Interpreter

Translates and executes each source-language instruction before translating and executing the next one. In the case of looping, the translation is repeated for each instruction in the loop each time control passes through the loop.

1.3.4 Translator

Translates statements in one symbolic langauge to equivalent statements in another symbolic language. The translation may be many-to-one, one-to-many, or one-to-one. Macro-processors and pre-compilers are included in this group.

1.4 LOADER

Controls the reading of programs and data for input to a computer, either for storage or for immediate use.

1.4.1 Bootstrap Loader

Initiate the reading of essential support software under "cold-start" conditions.

1.4.2 Linkage Editor/Loader

Produces a binary form of a program in which all symbolic references between different segments have been replaced with binary addresses, and loads the binary program into core memory for execution.

1.5 HARDWARE DIAGNOSTICS

Detects and isolates hardware malfunctions and demonstrates the proper functioning of the main processor, peripherals, displays, and data entry devices, as well as the interfaces between the system, subsystems, and external systems.

TABLE 8.4 (Contd.)

1.6 SYSTEM SIMULATIONS, TESTS, AND EXERCISES

Identifies potential areas for software improvement, detects and isolates software malfunctions, and demonstrates the proper functioning of the software.

1.6.1 Operational Scenario

Generates test situations by simulating "real-world" inputs to the system.

1.6.2 Control and Sequencing

Coordinates the generation and presentation of simulated input data to the system.

1.6.3 Data Collection

Collects and records data regarding the operation of the system.

1.6.4 Data Reduction and Analysis

Reduces the collected data and performs various types of mathematical analyses upon which to base an assessement of the operational efficiency, effectiveness, etc. of the system.

1.7 SOFTWARE DEVELOPMENT AIDS

Supports the development of applications software, but is not part of that software (e.g., debugging aids, data base analyzers, timing analyzers, compiler writing systems, etc.).

1.8 PORJECT MANAGEMENT AIDS

Supports project management.

1.8.1 Schedule Maintenance

Prepares, updates, and projects event, milestone, and overall project schedule to assist management in controlling the effort.

1.8.2 Financial Accounting

Maintains detailed accounting records on manpower, computer resources, overhead, etc.

2 APPLICATIONS SOFTWARE

Performs the specific tasks and functions for which the system was developed.

2.1 AVIONICS SOFTWARE

Controls, both automatically and in conjunction with the pilot, all aspects of the operations of an aircraft performed by computer.

TABLE 8.4 (Contd.)

2.1.1 Mission Planning

Coordinates the operation of all avionics software to ensure that the mission is accomplished in an effective manner. May provide any or all of the following functions:

- Preplanned Mission Evaluation
- Real-Time Mission Modification
- Steering Coordination
- Weapon Delivery Coordination
- Waypoint Sequencing and Mode Selection

2.1.2 Navigation

Maintains an awareness of the position, course, and distance traveled. Deals with any or all of the following navigational techniques:

- TACAN
- LORAN
- Doppler Radar
- Inertial Reference Unit
- Auxiliary Attitude Reference
- Air Data Computations
- Kalman Filter

2.1.3 Aircraft Steering

Coordinates the flight control software so that the aircraft may be steered either automatically (autopilot) or by the pilot (responding to displayed data). May provide for any or all of the following steering modes:

- Course Select
- Manual Course
- Instrument Landing System
- Airborne Instrument Landing and Approach
- Data Link
- TACAN

2.1.4 Flight Controls

Responds to the aircraft steering inputs to control attitude, speed, accelerations, etc. Manages the following types of controls:

- Roll, Pitch, and Yaw Controls
- Velocity/Acceleration Control
- Air Induction Control
- Energy Management and Control
- Cockpit Environment Control
- Crew Flight Input

TABLE 8.4 (Contd.)

2.1.5 Weapon Delivery

Controls the delivery of ballistic weapons, unpowered or powered terminal guided weapons, and powered mid-course guided weapons. May provide any or all of the following functions.

- Continuous Computation of Impact Point
- Weapon Miss Distance Computation
- Automatic Release Control
- Visual Release Control
- Stores Management and Control

2.1.6 Sighting, Designation, and Fixtaking

Computes ranges and angles to targets, destinations, and other points; accepts tracking-handle inputs which generate position, velocity, and heading error data for use by the navigation software; computes coordinates of terrain features; and calibrates system altitude and height above target. May provide for any or all of the following techniques.

- Forward Looking Radar
- Astro Tracker
- Laser Spot Seeker
- Low Light Level Television
- Forward Looking Infrared
- Low Altitude Radar Altimeter
- Tracking Handle
- Electro-Optical Sighting
- Visual Sighting
- Fixtaking

2.1.7 Display Control

Supports the presentation of data to the pilot. May provide for any or all of the following types of displays:

- Heads-Up Display
- Navigation Display
- Sensor Display
- Data Display

2.1.8 Data Entry/Retrieval

Supports the entry and retrieval of data by the pilot. May provide any or all of the following functions:

- Mission Entry
- Aircrew Panel Control
- Data Base Access

2.1.9 Communications

Controls all voice, digital, and video communications to and from the aircraft.

2.1.10 Electronic Countermeasures

Controls the equipment that reduces the effectiveness of enemy equipment and tactics employing electromagnetic radiation. May provide for any or all of the following countermeasure techniques:

Threat Warning
Electronic Warfare
Penetration Aids

2.2 COMMUNICATIONS, COMMAND, AND CONTROL SOFTWARE

Acquires relevant data, processes these data, presents the results to an operator for timely decision-making, and generates appropriate response based on the decision.

2.2.1 Data Acquisition

Controls the collection and initial processing of sensor data and the transmission of these data to the main processor. Performs the following functions:

Sensor Control
Signal Processing
Data Transfer

2.2.2 Data Processing

Receives, identifies, reduces, and combines input data for display. Performs the following functions:

Mission Control
Data Identification
Data Reduction
Data Manipulation

2.2.3 Operator Analysis and Decision Making

Forms the man-machine interface and provides for control of the mission by displaying, monitoring, and accepting data at operator consoles. Performs the following functions:

Mission Management
Data Display and Monitoring
Operator Data-Entry and Control

2.2.4 Response Generation

Controls the actions of the system in response to human decisions or as a result of the automatic assessment of a situation. Performs the following functions:

Communications Switching
Message Processing
Report Generation

TABLE 8.4 (Contd.)

2.2.5 Data Storage and Retrieval

Coordinates and controls the on-line storage, retrieval, and display of data contained in the system's files, tables and indexes.

3. DATA BASE

Contains all files, indexes, tables, and libraries required to store data to be used in the operation of the system, as well as the software required to maintain this data base.

3.1 FILES

Collections of related records, organized to meet a specific purpose, stored on magnetic tape, disk, etc., or in some cases, directly in core memory.

3.1.1 On-Line Updatable Files

Files which are processed on-line by console operators and are displayed and updated through the use of keyboard data entry.

3.1.2 Internal Files

Files which are used internally by the application software. This type of file is normally a read-only file which is transparent to the user.

3.1.3 System-Generated Files

Files which are generated by the system as a result of normal operations (i.e., are not specifically updated by console operators). Historical data are normally captured in this type of file.

3.1.4 Remote Data Base Files

Files which are utilized by the system and the operational personnel, but are not physically located with the system and are not directly under its control.

3.2 INDEXES

Ordered lists of references to the contents of a larger body of data, such as a file or record, together with keys or reference notations for identifying, locating, searching, or retrieving the contents.

3.3 TABLES

Organized collections of data, usually arranged in an array in which each item is uniquely identifiable by some label or by its relative position.

3.4 Program Support Library

An organized collection of data associated with a program or a group of programs.

3.4.1 Program Library

A collection of proven computer programs, routines, and subroutines which can be combined with other programs or inserted into them by various methods to solve problems or parts of problems.

TABLE 8.4 (Contd.)

3.4.2 Macro Library

A library that consists of sets of instructions which are represented by a single macro-instruction. Using this library, a programmer can code software such that the computer automatically generates the appropriate set of instructions represented by each coded macro-instruction.

3.5 MAINTENANCE ROUTINES

Supports the off-line maintenance of the system data base (initialization routines, data entry routines, restructuring routines, updating routines, formatting routines, etc.).

Table 8.4 lists CPCs under three headings: Support Software, Applications Software, and Data Base. The section on applications software covers two areas specifically of interest in this contract: avionics, and command, control, and communications (C^3). A slightly different approach was used for these two types of software. The C^3 software is divided into functions which are common to all C^3 systems. To the extent that this list is exhaustive, there should be no need for adding items. For avionics, however, applications are mission-oriented and are more akin to a shopping list from which CPCs will be selected. This list is likely to grow with technology.

An initial version of the CPC list was based on (1) previous experience in software design; (2) review and analysis of the Work Breakdown Structure (WBS) presented in the Request for Proposal for this study contract; (3) review and analysis of various other existing WBSs; (4) a survey of literature relating to the development of WBSs; and (5) reference to literature describing the essential functions of both avionics and communications, command, and control systems. Of particular importance in the avionics area was work done by General Dynamics[39] and an article by Lynn Trainor.[40]

Once the initial version has been developed, entries were compared with a list of CPCIs from recent ESD software developments, received from the ESD project officer for this study. The review consisted of determining whether each of the existing CPCIs could be categorized according to the entries in the CPC list. An iterative process ensued that refined the initial version to that presented in Table 8.4.

At the same time, Mr. John Glore of MITRE was preparing a similar list as part of his effort in developing a Statement of Work preparation guidebook.[41] Mr. Glore has concentrated more on the non-applications software. His efforts were combined with ours in preparing Table 8.4.

The CPC list is our "best effort" to identify the elements of a software system. However, it is inevitable that additional entries will be identified as the list is reviewed by people at other Program Offices and as a result of the development of new projects.

8.5 RESOURCES

We now turn to the third dimension of software cost elements (Fig. 8.1), namely, the resources. These resources fall into the categories shown in Table 8.5. Some of these resources are contractor-furnished while other are government-furnished; the first three elements of contractor-furnished resources are the most important to software and will be covered in more detail in the reporting system.

Direct Technical Labor and Support Personnel is the most significant category of the three. Man-hours, as well as costs, should be

TABLE 8.5

RESOURCE CATEGORIES

Contractor

- Direct Technical Labor and Support Personnel
- Computer Resources
- Software Unique Facilities and Equipment
- Overhead
  - Support Personnel
  - Materials Consumption
  - Travel
- G&A and Fee

Government

- Air Force Labor
  - Military
  - Civilian
- Technical Assistance Contractor Labor
- Government-Furnished Equipment and Facilities

collected. Important subcategories include Direct Software Development, Configuration Control, Project Management, Documentation, and Training. Details are discussed in Sec. 8.6.

Computer Resources, the next most important category, includes acquisition, rental, and operation of the computer systems used in software development and maintenance. Specifics are given in Sec. 8.7.

The third category, Software Unique Facilities and Equipment, often is important in large software projects. These include a facility for the software development and a facility (perhaps the same) for centralized software maintenance. Specific costs include acquisition (construction) or rental. Operation of the facilities should also be charged to this category, although this tends to be an overhead account and we therefore suggest handling it as a rental charge. This would allow accountability for parts of facilities.

The remaining contractor items are very difficult to assign directly to software. They include non-technical support personnel, material consumption, travel, G&A, and fee. These items have less to do with software itself than with the way the contractor does business. A sufficiently elaborate accounting system could keep track of the number of support personnel, specific travel, consumption of paper, etc. However, we do not believe the collection of this type of information would be cost-effective; requiring its separation and collection is bound to lead to contractor resistance.

It is our recommendation that these costs be accumulated for the contractor portion of the defense system as a whole, and a burden rate be calculated. This burden rate could be compared among contractors, if desired, to assure competitiveness. It could also be used to allocate indirect costs if a total software cost for the contractor is

desired. Total contractor costs for a particular software development would then be given by:

$$C = (C_{DL} + C_C + C_{F\&E})\ (1 + R)$$

where

C = total cost

C_{DL} = cost of direct technical labor and direct support personnel

C_C = cost of computer resources

$C_{F\&E}$ = cost of software unique facilities and equipment

R = burden rate covering overhead, G&A, and fee.

Government resource consumption is also an important part of software resources. The role of the government manager changes during the life cycle from the Program Office staff, overseeing the software development, to Air Force Logistics Command system manager, providing or overseeing maintenance of the software. Costs include those of military and civilian government employees, and Technical Assistance Contractor(s). The reporting of these costs has been outside the scope of our study, but we speculate that they are a large and therefore important part of the total cost of software development and maintenance.

Similarly, it is important to record government-furnished equipment, facilities, and computer hardware as resources utilized for software development and maintenance. This is particularly important when comparing costs of software developments. Although gathering data on this cost is beyond our scope, we feel that all government-furnished software-related items must at least be specified in any software cost reporting system. Thus, we recommend a resource category for government-furnished equipment and facilities.

We now turn to definitions of the two main cost categories. The elements will then be summarized in Sec. 8.8.

8.6 DIRECT TECHNICAL AND SUPPORT LABOR

The primary resource consumed is direct technical and support personnel man-hours. Basically, the cost of the item is equal to the number of man-hours consumed times cost per man-hour. Of course, cost per man-hour varies with skill categories and the mix of skill categories changes over the life-cycle phases. Thus, man-hour data is not enough information to arrive at a good cost estimate. Therefore, we recommend that both man-hour and cost data be captured in the reporting system.

The ideal reporting system would report cost and man-hours by skill category for every item in the contractor's work breakdown structure. While this level of detail is technically feasible and internally exists to almost this level for some contractors, there is little or no chance that a contractor would allow the government such visibility into its operations. The excuse would be that it costs too much, and indeed the contractor would be at least partly right. This level would certainly not be cost-effective for the government. Furthermore, it is not really needed for cost control or for future cost estimation.

In this section, the three dimensions of the reporting system elements are consolidated into one set of reporting elements, at a level sufficiently detailed to give adequate visibility into the development process for cost control and direct-labor cost estimation. Both cost and man-hour information are required. However, reporting on a regular basis to skill category level, which has been suggested by some, is not required. It is far too much detail for a regular reporting system, and information on skill mix can be obtained from contractor estimates.

In this section, then, the cost reporting system elements are first defined and then related to typical software activities in the different life-cycle phases. Cost and man-hour data are to be reported on a regular basis against this structure. Supplemental information, based on contractor estimates, is then defined. These estimates are made at key milestones and include information on skill-level categories. Standard skill levels are also defined.

8.6.1 Suggested Reporting System Elements

Suggested reporting system elements for man-hours and costs are shown in Table 8.6. They are broken down by level of end-item reporting (CPC level, CPCI level, and total defense system or contract level).* All data are for direct technical labor, with the exception of support personnel such as managers, secretaries, and key punchers who are totally occupied by contract software tasks. Support specifically does not include personnel required to run the computer; these are included in computer resources (Sec. 8.7).

Of utmost importance to future cost estimation is the collection of resource requirements using a standard set of end items, which we have called CPCs. These were defined in Sec. 8.4. However, this goal should not require artificial allocations of time to these categories. Accordingly, we recommend that only the following activities be recorded to the CPC level, and identified with the activity label specified:* (1) Detailed design of the CPC functions should be recorded against Design, while coding and initial checkout of the CPCs should be assigned to Coding; (2) redesign and recoding of a CPC in response to test error detection should be recorded under Integration;**

---

*Reporting at different levels within the same extended work breakdown structure was an expressed concern. However, in conversation with Greg Maust, ESD Comptroller, it became clear that there is no restriction on reporting at different WBS levels in the Cost Performance Report.

**Note that no attempt should be made to separate redesign and recoding.

TABLE 8.6

REPORTING SYSTEM ELEMENTS

Activity	Definition
	CPC Level: All man-hours for particular activities that can be identified to the CPC level
Design	Detailed CPC design
Coding	Coding and initial checkout of CPCs
Integration	Redesign and recoding in response to test error detection
Installation	Modification of code for site-specific application
Maintenance	Coding and integration in response to error reports (DRFs).
	CPCI Level: All resources for particular activities related to the CPCI and not assignable to CPCs.
Design	CPCI design, interfaces, and allocation of functions to modules
Testing	Defining and carrying out CPCI-level software tests.
Independent V&V	Independent verification and validation of CPCIs
	Contract or System Level: All resources for particular activities related to the contract or system and not assignable to CPCs or CPCIs
Analysis*	Studies to resolve conceptual problems and demonstrate capability to meet requirements, including algorithm development, allocation of functions to CPCIs, etc.
Testing*	Testing of software functions at system level, including hardware interface
Independent V&V	Independent verification and validation at system level

TABLE 8.6 (Continued)

Activity	Definition
Management	
Engineering	All management personnel assigned to oversee the technical quality of the software product, including directing software development, directing the preparation and review of software portions of technical documents, preparation for and attendance at technical review meetings, on-site support during operations, fault isolation, etc.
Configuration	All management personnel assigned to oversee the maintenance of baselined software and software specifications and to document and incorporate approved changes to the baseline.
Project*	Other management activities directly associated with the software product, including contract management, cost and schedule management, business and administration planning, directing and controlling the project
Training	Preparation of course material, demonstration and conduct of training courses if contracted, etc.
Documentation	Writing, editing, publishing, reproduction, and dissemination of software documents required by the CDRL (could be reported at CPCI level if more detail is needed).
Technical	Part I and Part II Specifications, Test Plans, User's Manuals, specified technical reports, etc.
Configuration	Version descriptions, configuration index, change status reports, specification change notices, etc.
Program Management*	Software development plan, cost performance reports, program schedules, program milestones, etc.
Support*	Non-technical personnel totally assigned to the software portion of the contract such as key punchers, secretaries, etc.

*These items are sometimes difficult to separate from hardware.

and (3) modification of CPC code for specific site requirements should be reported under Installation, while coding and testing of corrections for bugs discovered during operation should be reported under Maintenance.

Direct labor at the CPCI reporting level includes those activities that cannot easily be separated into CPCs but can be tracked naturally to the CPCI level. This includes design activities concerned with the allocation of functions to CPCs and modules, as well as interfaces between CPCs. Defining and conducting CPCI-level tests in response to Part I specification requirements should be reported against Testing at this level. Also, any independent V&V activities which are related to specific CPCIs should be identified at this level.

Direct labor reported at the system (or contract) level can be divided into product and support activities. Product activities include all the special studies performed under Analysis, including functional allocation to CPCIs. System-level testing and independent V&V activities relating to these tests should also be reported at this level.

The support tasks include management, training, and documentation. They should also be reported at the system level with management divided into engineering, configuration, and project. Documentation is divided into technical, configuration, and program management.

The above elements are for technical and management man-hours only. Other support man-hours are reported in aggregate against the Support element at the bottom of Table 8.6. Note that at the system level it will be difficult to separate software from hardware or system costs. As a result it may be advisable to report some of these costs

system-wide and not try to allocate them to software. For completeness, we have included them in Table 8.6 because they would be present in a software-only development. Therefore, for comparability they should at least be estimated. More about this in Sec. 8.8 when the proposed elements are correlated with an existing reporting system.

8.6.2 Relationship of Reporting Elements to Life-Cycle Phases

These elements were selected after considering each of the software activities under each of the life-cycle phases and trying to preserve as much information as possible without requiring too much detail or manual allocations. In this subsection, the cost elements are further defined by typical activities occuring in the life-cycle phases (identified in Sec. 8.2). Tables 8.7 through 8.13 map the tasks typically performed in the phases into the proposed reporting elements. A summary is shown in Table 8.14. Note that data is typically available through monthly reports by phase. The tables show how the data is to be assigned to the reporting elements.

The reader is again reminded that some of the reporting elements bear the same name as the life-cycle phases. In effect, Table 8.14 is a mapping of life-cycle phase names into activity definitions (reporting elements) using the same terms. Also note that we in effect will have both a milestone and an activity reporting system (columns vs rows). By having both, and being able to track between the two (through Table 8.14), the advantages of both methods will be attained. This was done to show where difficulties lie between the activity and milestone interpretation of these terms. Where activities cannot be separated into the classic definitions, new terms are defined. Thus, Integration includes redesign and recoding tasks which are inseparable in the integration and testing life-cycle phases.

TABLE 8.7

REPORTING ELEMENTS: ANALYSIS PHASE (Milestone Definition)

(Contract Award to PDR)

	Reporting Element (Activity)	Item	Description
CPC Level	Design	See Table 8.8	If early start approved for some critical items.
	Coding	See Table 8.9	If early start approved for some critical items.
CPCI Level	Design	See Table 8.8	If early start approved for some critical items.
System Level	Analysis	Special Studies	Includes system specification requirement review, algorithm development, and evaluation; CPCI and CPC definition; functional allocations to CPCIs and CPCs; simulations; throughput and timing analyses; etc.
	Testing	Test Plan	Development of testing objectives.
	V&V	Requirements Validation	Identify V&V requirements. Validate choice of alternatives.
	Management	Engineering	As in Table 8.6 including preparation and conduct of PDR and supervision of Part I Specifications preparation.
		Project	As in Table 8.6
	Documentation	Technical	Technical documentation including software analysis reports, software Test Plan, and Part I Specifications.
		Program Management	As in Table 8.6.
	Support		As in Table 8.6.

TABLE 8.8

REPORTING ELEMENTS: DESIGN PHASE (Milestone Definition)

(PDR to CDR)

	Reporting Element (Activity)	Item	Description
CPC Level	Design	CPC Flowcharts	Detailed flowcharting of CPC functions in preparation for coding.
	Coding	See Table 8.9	Early coding on critical CPCs approved at PDR.
CPCI Level	Design	Function Allocation	Functional allocation to modules, including software design trade-off analysis. Includes design for data base, support software, and test software not assignable to CPCs.
		Interface Design	Design of interfaces between CPCs, including flowcharting.
	Testing	Test Procedures	Specify test procedures for each CPCI.
System Level	Analysis	See Table 8.7	Any special studies not completed at PDR and authorized at PDR to be continued to a specific date.
	Testing	System Test Procedures	Specify test procedures for exercising software within the system testing.
	V&V	V&V Plan	Define independent V&V Plan.
	Management	Engineering	As in Table 8.6, including CDR preparation and conduct, review of Part II Specifications, user's manuals, etc.
		Configuration	Input to configuration control plan, and establishment of configuration control mechanism.
		Project	As in Table 8.6
	Training	Training Plan	Training plan specification.

TABLE 8.8 (Continued)

Reporting Element (Activity)	Item	Description
Documentation	Technical	Technical documentation, including Part II Specifications, detailed test and evaluation procedures, user's manuals, etc.
	Configuration	Configuration management plan.
	Program Management	As in Table 8.6, including updated Computer Program Development Plan, training plan (draft), etc.
Support		As in Table 8.6.

TABLE 8.9

REPORTING ELEMENTS: CODING AND CHECKOUT PHASE (Milestone Definition)

(CDR to Source Baseline)

	Reporting Element (Activity)	Item	Description
CPC Level	Design	See Table 8.8	Specific detailed flowcharting for non-critical CPCs, OKd at CDR.
	Coding	Coding	All coding and debugging associated with developing initial source code for module.
CPCI Level	None		
System Level	V&V	V&V Procedures	Definition of independent V&V procedures.
	Management	Technical	As in Table 8.6.
		Configuration	The initial baselining of source decks (as they are completed) and inclusion in the software library.
		Project	As in Table 8.6.
	Documentation	Technical	Initial preparation of code documentation.
		Configuration	As in Table 8.6, including periodic reporting of current CPC status (baselined, in test, error response, etc.).
		Program Management	As in Table 8.6.
	Support		As in Table 8.6

TABLE 8.10

REPORTING ELEMENTS: INTERNAL TEST AND INTEGRATION PHASE (Milestone Definition)

(Source Baseline to Start of PQT)

	Reporting Element (Activity)	Item	Description
CPC Level	Integration	Error Response	Redesign and recoding in response to detected errors.
CPCI Level	Testing	CPCI Tests	Testing programs against CPCI Part I Specifications utilizing approved test plans.
System Level	Testing	None	
	Management	Engineering	As in Table 8.6, with emphasis on carrying out test plans and correcting errors.
		Configuration	As in Table 8.6, with emphasis on maintaining approved versions, record of changes, and test result data.
		Project	As in Table 8.6.
	Training	Training Manuals	Develop training materials.
	Documentation*	Technical	As in Table 8.6, with emphasis on test reports, preliminary program documentation, flowchart refinement, etc.
		Configuration	Reporting as required of CPC status, number of errors, etc. (Table 8.3).
		Program Management	As in Table 8.6.
	Support		As in Table 8.6.

*Could be reported to CPCI level if desired.

TABLE 8.11

REPORTING ELEMENTS: QUALIFICATION TEST PHASE (Milestone Definition)

(PQT to PCA)

	Reporting Element (Activity)	Item	Description
CPC Level	Integration	Error Response	Redesign and recoding in response to detected errors.
CPCI Level	Testing	CPCI Tests	Qualification Test of programs against Part I Specifications.
	V&V	V&V of CPCIs	Independent V&V of programs against Part I Specifications and V&V approved plans.
System Level	Testing	System Tests	System level software tests.
	V&V	V&V Check*	Independent V&V of system-level software performance.
	Management	Engineering	As in Table 8.6, with emphasis on Qualification Test, error correction, and final technical documentation.
		Configuration	As in Table 8.10
		Project	As in Table 8.6.
	Documentation**	Technical	Final version of all technical reports, including programs, baselined Part II Specifications, test results, etc.
		Configuration	Periodic reports of CPC status, number of errors, etc. (Table 8.3).
		Program Management	As in Table 8.6.
	Support		As in Table 8.6.

*Usually part of Installation. Exception can occur for Avionics applications.

**Could be reported to CPCI level if more visibility desired.

TABLE 8.12

REPORTING ELEMENTS: INSTALLATION PHASE (Milestone Definition)

(PCA to IOT&E)

	Reporting Element (Activity)	Item	Description
CPC Level	Installation	Site-Specific Changes	Changes to software code or data base to adapt approved software to specific site.
CPCI Level	None		
System Level	Testing	Contractor Test of Required Changes	Off-site testing of required changes.
	Independent V&V	Operational Test of Changes	Specific tests of software to accept site-specific changes.
	Management	Engineering	Technical management of software changes. On-site personnel for systems test.
		Configuration	Development of site-specific baselines incorporating site changes.
		Project	As in Table 8.6.
	Training	Initial User Training	User training of software items.
	Documentation	Technical	Site-specific changes to technical documents.
		Configuration	Site-specific changes to configuration documents.
		Program Management	As in Table 8.6.
	Support		As in Table 8.6.

TABLE 8.13

REPORTING ELEMENTS: OPERATIONS AND SUPPORT PHASE (Milestone Definition)

	Reporting Element (Activity)	Item	Description
CPC Level	Maintenance	Priority Error Maintenance	Recoding and integrating corrections in response to identified software errors (priority).
		Software Program Maintenance	Recoding and integrating corrections in response to non-priority errors. Work parallels ECP development.
CPCI Level	None		
System Level	Analysis	Special Studies	Compare current system capabilities with existing and future mission requirements, define required improvement areas, evaluate the impact of proposed ECP changes upon system software and perform permanent core impact and segmentation studies.
	Testing*	Testing	Contractor qualification test of integrated error corrections.
	V&V*	Independent Qualification Testing	Independent qualification test of above.
	Management	Engineering	As in Table 8.6.
		Configuration	As in Table 8.6.
		Project	As in Table 8.6.
	Training		As in Table 8.6.
	Documentation	Technical	As in Table 8.6
		Configuration	As in Table 8.6.
		Program Management	As in Table 8.6.
	Support		As in Table 8.6.

*Can be reported to CPCI level if more visibility desired.

TABLE 8.14

REPORTING SYSTEM ELEMENTS AND LIFE-CYCLE PHASES

LIFE-CYCLE PHASES

Reporting System Elements	Analysis Contract Award to PDR	Design PDR to CDR	Coding & Checkout CDR to Source Baseline	Internal Test and Integration Source Baseline to PQT	Qualification Test PQT to PCA	Installation	Operations & Support
CPC Level							
Design	Authorized Early Design	Detailed Design, Flowchart	Authorized Late Design				
Coding	Authorized Early Coding	Authorized Early Coding	Coding and Debugging				
Integration				Error Response	Error Response		
Installation						Site-Specific Changes	
Maintenance							Priority Error Correction
CPCI Level							
Design	Authorized Early Design	Module/ Function Allocation, Interface Design					
Testing		Test Procedures		CPCI Tests	CPCI Tests		
Independent V&V					V&V Tests		
System Level							
Analysis	Special Studies	Authorized Late Analyses					Special Studies
Testing	Test Plan	System Test Procedures			System Tests	Contractor Off-Site Tests of Changes	Contractor Qualification Test of Error Correction
Independent V&V	Requirements, Validation	V&V Plan	V&V Procedures		V&V Tests	On-Site Qualification Test of Changes	Independent Qualification Testing of Error Correction
Management:							
Eng.	X	X	X	X	X	X	X
Config.		X	X	X	X	X	X
Project	X	X	X	X	X	X	X
Training		Training Plan		Training Manual Preparation		Initial User Training	Training and Demonstration
Documentation							
Technical	X	X	X	X	X	X	X
Config.		X	X	X	X	X	X
Pro.Mgmt	X	X	X	X	X	X	X
Support	X	X	X	X	X	X	X

Note the provision for early starts and late finishes in coding, design, and analysis reporting elements. This is provided because it is so often the practice. We do believe that this overlap should be limited to specific CPCs with authorization at contract award, PDR, or CDR. Authorization should be accompanied with supporting rationale. This should aid in control, as well as ultimately facilitating an assessment of the cost impacts of planned overlapping.

Note also that some activities are spread throughout the life-cycle phases. This includes testing, V&V, and training. Life-cycle (milestone) definitions of the term "test" are obviously misleading.

The activities listed under each life-cycle phase were compiled after reviewing about six WBSs for NASA and ESD software projects. Operations and maintenance is an exception. It is based upon the SDC reporting system to SAMSO as Computer Program Integration Contractor for the Satellite Control Facility (SCF). For documentation requirements, we referred to AFM 800-14.[11] Some special comments are listed below.

The analysis performed by the development contractor is usually not all the analysis performed for the software development. A considerable amount can already have been spent during the concept formulation and validation phases of the system acquisition life cycle (Fig. 2.1) to prove out concepts or build prototypes. Problems of data comparability are thus introduced. There will be more about this topic in Sec. 8.10.

V&V has been separately identified. Ordinarily, this will be done by a separate contractor, which simplifies the reporting. Qualification testing during maintenance and installation is shown as V&V, since it is typically done on-site, or at least separately from the development group. This is not the case for the Formal Qualification Test of the original product, in which the developing contractor may run the demonstration.

Note also that internal testing during installation could be shown at the CPCI level, but this level of detail does not add much information and we do not suggest it.

Finally, operations and maintenance has a peculiar problem. A major activity--product improvement through ECPs--is not included because we feel this should be shown as a new development and not a maintenance activity. The reason for this split is that product improvement should have no relation to errors in the original product.

Unfortunately, it will be somewhat arbitrary to separate management, documentation, and support hours between these two major activities (ECP development and error correction). In analyzing the costs of ECPs (or error correction) some method for allocation of these support costs will have to be used for comparability.

Also note that a great deal of the analysis will have been completed before the ECP is accepted. Developments (ECPs) during the maintenance phase, in essence, are lacking a recorded Analysis phase. Analysis is performed as special studies. Also, it is common practice not to repair low-priority errors during qualification testing of the ECP development. These are merely reported as errors (DRFs) and repaired when there is time. Hence, care will have to be taken in comparing ECP developments among themselves or especially with new software developments. More is said on this topic in Sec. 8.9.

8.6.3 Frequency of Reporting and Special Reports

As stated earlier, cost and man-hour data should be reported against the proposed elements periodically (say monthly). From this data, cost per man-hour can be calculated by life-cycle phase or by resource element. Furthermore, a comparison with progress towards product completion (Sec. 8.3) gives a good indication of whether a project is on target. For example, what percentage of the coding dollars

and man-hours have been consumed and how does this compare to the percentage of completed code? This comparison enhances the Program Office's ability to control the development.

Adding a requirement for the contractor to report updated estimates of man-hours and costs at key milestones will provide greater insight. Furthermore, this information can be required by skill category. Therefore, we recommend that estimates of man-hours by skill category be required for each reporting element and life-cycle phase. The method of estimation should also be shown.* Total direct labor costs (not by skill category) should also be reported so that an actual cost per man-hour can be derived for each combination of reporting element and life-cycle phase.

This estimate should be made for all development phases (through PCA) at contract award. Since CPCs and CPCIs will in general not have been defined at contract award, estimates will be aggregared to the system level. At PDR, these estimates can be updated for the remaining software development phases. Visibility to the CPC level will then be possible and should be required. These estimates should again be updated at CDR, to the CPC level.

Since dollar-per-man-hour factors may be calculated from these estimates, the Program Office will be able to compare these to actuals to see if the factors are changing significantly. If they are significantly higher than planned, troubles with particular CPCs may be occurring which require higher-priced talent to fix. Hence the higher cost. If the actuals are significantly lower than estimates, required progress may not be taking place or the CPC may be coming in ahead of time.

---

*This recommendation has also been made by Captain Devenny (Ref. 1, pg. 88).

Hence, the Program Office will have indications of anomalies, both good and bad, long before they are officially reported.

Requiring these detailed estimates will also build up a Government data base of factors upon which to convert man-hours, by reporting elements, to dollars. Hence, it will aid in future cost estimation. Also, this information can be used to evaluate contractor proposals (does he have the right mix of talent?, are they reasonably priced?, etc.).

8.6.4 Typical Skill Category Definitions

A standardized list of skill categories is required if the contractor estimates are to have all the benefits mentioned above. We have developed the following tentative list of skill categories that will be required during various phases of the system life cycle:*

- Management personnel
- Scientists and engineers
- Systems analysts
- Systems programmers
- Applications programmers
- Program librarian/secretaries
- Data entry operators

Excluded from this list are computer operators, who are included under computer resources (Sec. 8.7), and support personnel who are not assigned full-time to the software development.

*Skill category material was derived principally from Hansen's Weber Salary Survey on Data Processing Positions, as well as various other related salary surveys.

Each of these categories, except management personnel, may be further classified according to level of skill:

Senior Level: Usually competent to work at the highest technical level of all phases of a project. Supervises lower-level personnel to provide technical guidance.

Level A: Able to work under general supervision on most phases of a project but requires some technical guidance on other phases.

Level B: Works under direct supervision on several phases of a project but requires technical guidance and instruction on most other phases.

Level C: Works under immediate supervision on individual tasks, with the work being carefully checked.

Appendix C presents job descriptions for each of the skill categories, and current salary information.

8.7 COMPUTER RESOURCES

There are three primary methods of using computer resources in software development and maintenance. First, and most common for large software developments, is a completely dedicated system, often utilizing equipment which will become part of the operational system, or is a duplicate of the operational system (a useful option if software is to be centrally maintained). The Defense Support Program-CONUS Ground Station (DSP-CGS) software development by IBM is an example of the former, and the Satellite Control Facility (SCF) software development by SDC is an example of the latter. (Both of these software developments are part of SAMSO.)

A second method, utilized in smaller programs, is to develop and maintain the software on a dedicated system which is not necessarily

the same as the target system. In this case, more general programming techniques are used so that conversion to the target system is simplified.

A third method, primarily used during debugging when I/O is limited, is to use a base computer which is shared (e.g., time sharing) with other projects.

In any case, costs can be divided into investment and operations. Investment costs include acquisition, installation, and rental charges for all contractor-furnished equipment. Investment costs can also be included in the service charge for a computer center (non-dedicated equipment) or a time-share service. Often, Government Furnished Equipment (GFE) is used and will not appear as a cost to the contractor (or to the Government in most reporting systems). Care must therefore be taken when comparing computer costs to identify just what is GFE and what is not.

Operating costs are charged to a separate account, often a separate contract, for a dedicated system and are included in a user rate for shared systems. For the latter, cost is directly a function of utilization, while for the former, it is a function of system availability. Of course, availability will be based on utilization. Therefore, it is important (for comparisons) that systems be properly sized. Too little capability may lead to smaller direct costs at the price of poor turnaround time which, in the long run, may be more expensive.

It is important to note that utilization is far from static. Many have reported increased utilization during development as shown in Fig. 8.3. This pattern also holds for ECP development during maintenance, as Table 8.15 illustrates for the two SCF associate contractors.

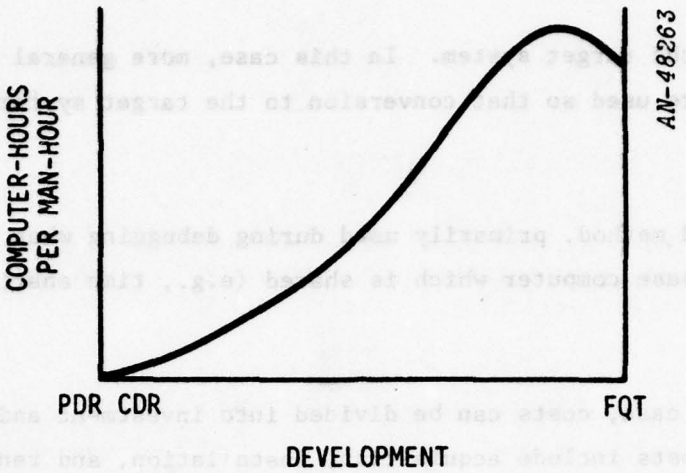


Figure 8.3. Utilization of Computer Resources During Development

TABLE 8.15

COMPUTER HOURS FOR MAINTENANCE

(Data on Satellite Control Facility Programs)

	Ratio of Computer-Hours to Technical Man-Hours				
	Contractor A		Contractor B		
Month	ECP	Maintenance	ECP	Maintenance	
1974					
Oct.	.09	.04	.11	.09	
Nov.	.07	.10	.06	.06	
Dec.	.10	.02	.09	.07	
1975					
Jan.	.08	.02	.16	.10	
Feb.	.16	.14	.14	.06	
Mar.	.24	.21	.12	.04	Version 15.0C* FQT
Apr.	.30	.08	.30	.03	Version 15.1** Test and Integration
May	.25	.07	.33	.02	
Jun.	.19	.11	.47	.03	
July	.26	.13	.76	.04	Version 15.1 FQT
Aug.	.12	.09	.43	.03	

*Version 15.0C contained small changes which were predominantly DRFs.

**Version 15.1 introduced major new capability.

During the period depicted in Table 8.15, two revisions were completed and installed. Version 15.0C went through FQT in March 1975. This was a small development with a lot of normal maintenance DRFs. Version 15.1, on the other hand, was a major new capability. It was in integration and test from April 1975 to July 1975.

Note that for both contractors, computer utilization for ECP development rose during Version 15.1 FQT. Although Contractor B showed little change for Version 15.0C FQT, Contractor A showed a significant change. Note also that utilization was at a fairly uniform level during maintenance. The exception to this was March 1975, but this too was connected with the maintenance actions incorporated with the Version 15.0C FQT.

Although no cost estimating relationships have been derived from this limited sampling of the SCF data, the data would support a model which could estimate computer utilization from technical man-hours. Costs could then be derived from this utilization based on type of operational availability, type of system, and amount of GFE.

Data collection that would be necessary to support any future development of a cost model includes a system description, cost data (including investment and operations), utilization data, and availability data. Each is discussed below.

8.7.1 Computer System Description

A complete description of the computer system should be recorded. Table 8.16 is a list of items that should be considered for a batch operation. Typical software items purchased with the system are also shown in Table 8.17. The description should record which items are included, how many, and their cost or rental charge. GFE should be designated. If the equipment will be a part of the operational system (a non-software cost), this should also be stated. Finally, a

TABLE 8.16

COMPUTER HARDWARE

Hardware Item	Typical Descriptive Parameters
MAINFRAME:	
CPU	Machine cycle time, add time, no. CPUs, no. registers, register size, no. buffers, buffer length, instruction repertoire
Main Storage	Storage type, read cycle time, write cycle time, bytes fetched/cycle, minimum capacity, maximum capacity, increment size, word size
Additional Core Storage	Type, total bytes, data transfer rate, cycle time, no. access channels
Controllers	No. devices/controller, data transfer rate
Cabinets	Dimensions, use
Cabling	Length, bit paths
I/O Channels	Data transfer rate, no. channels, channel type
PERIPHERALS:	
Operators' Console	CRT or hard copy, functions available
Printer	Print positions, lines per minute, no. character positions, lines per inch, skip speed, type of printing (drum, train, chain, other), paper width
Card Reader, Punch	Cards/minute read and punched, positions/sec. printed, card size, card hopper capacity, card thickness, card codes, reading method
Disk Pack:	
• Head/Track and Moving Head	Fixed or removable, fixed or moving head, average access time, average data transfer rate, bytes of storage, no. drives per pack, no. drives per controller, average rotational delay, no. channels
• Cassette	No. tracks, includes/sec., bits/inch, recording mode, data transfer rate
• Floppy	No. tracks, sectors/track, bytes/sector, no. drives, hopper capacity, records/min. read, RPM

TABLE 8.16 (Continued)

COMPUTER HARDWARE

Hardware Item	Typical Descriptive Parameters
Drum	Average access time, no. tracks, data transfer rate, bit density, cycle time
Tape Drive	No. tracks, inches/sec., bits/sec., recording mode, no. drives per controller, no. channels, rewind speed, no. drives/cabinet
Paper Tape Reader	No. tracks, characters/sec., reel size
Paper Tape Punch	No. tracks, characters/sec., reel size
Optical Scanner	Encoding type, document width, documents/minute, no. pockets, fonts read, document length, document thickness
Plotter	Table or cylindrical, size, plot records/sec., pen type, dimensions
Terminal:	
• CRT	No. characters, keyboard type, printer type, screen size, no. lines, no. characters/line, expansion feature, no. keys
• Teletype	Data transfer rate, character set capability, forms feeder capability
• Telephone	Size and weight, video capability, hard copy capability, print speed, TTY compatibility, telephone line discipline, data transfer rate
ANCILLARY EQUIPMENT:	
Key Punch:	
• To Cards	Card columns, character set, drum card capability, print capability, duplication capability, interpreting capability
• To Tape or Disk	Character set, format types, record length variability, programming functions
Card Sorter	Cards/min., character set, hoppers (no. and size)
Interpreter	Cards/min., card size
Reproducer	Cards/min., functions
Tape Cleaner	Reel size, speed, tape size
Degausser	Tape size, degaussing speed

TABLE 8.17

COMPUTER SOFTWARE

OPERATING SYSTEM:

- I/O Supervisor
- Permanent File Management
- CPU Scheduling/Memory Management
- Real-Time Support

PROGRAM DEVELOPMENT:

- Source Text Editor
- Assembler
- Compilers
- Linkage Editor/Loader

CHECKOUT AND TEST:

- V&V Tools:
  - Debugging Tools
  - Code Analyzers
  - Test Case Generators
  - Specification Language
- Configuration Management
- Test Bed:
  - Simulation System
  - Simulation Environment

DOCUMENTATION:

- Flow Charter
- Document Maintenance and Generation

description of the operating environment should be given: dedicated, shared, or time-shared. For the latter two, the service charge per computer hour should be recorded.

8.7.2 Computer Resource Costs

Although detailed information is required for direct labor, computer cost data can be more aggregated. For a dedicated system, this is the only logical way to keep the books. Even for shared systems, costs can be allocated on the basis of total system utilization. Thus, cost will depend on other users.

For each system in question, we recommend that the acquisition (investment) cost (or rental) be recorded and that total operating costs for providing the service be reported monthly. Operating costs include the cost of computer center operations personnel, user charges (excluding investment), equipment maintenance, etc. Facilities maintenance should be reported separately.

8.7.3 System Utilization

System utilization information should be divided into two parts: (1) that for the subject project and (2) that for the total computer system, including other clients, if shared. The latter can be reported at a system-wide level, in terms of hours per week, just as investment cost was reported at that level (Sec. 8.7.2). No special requirements are necessary for time-share reporting.

For the project in question, computer-hour information at a more disaggregated level should be collected to see if the patterns of computer-hour per technical man-hour are different for each CPC. Thus, we wish to track computer-hours by CPC for activities that take direct labor hours at that level. This is also true for each CPCI and for the system as a whole. There should be little difficulty in maintaining

computer hours in this manner. The direct labor account numbers will suffice.

Reporting on a monthly basis will also allow one to relate computer hours to activity within a CPC should more disaggregation of computer-hour data be deemed necessary.

8.7.4 System Availability

System availability is a little more difficult to measure. Two types of measures are desired: one for availability and one to see if availability is suitable. Availability can be measured if a log is kept of hours per week during which the system is available. Whether this meets actual needs is another question. For batch operations, turnaround time is a good measure (where turnaround time is defined as time of log out minus time of log in minus non-queued execution time). Both average and 95th percentile should be reported monthly.

For time-share systems, the same type of measure would be desirable. However, this should include such things as waiting time in line, etc. A better measure would be a ratio of total terminal time divided by cumulative core execution time. I/O time and queue time would constitute the difference. Again, average and 95th percentile would be required.

Note that all measures suggested are independent of program size and nominal execution time.

8.8 FEASIBILITY OF REPORTING SYSTEM

The reporting system has now been defined and it is useful to ask how practical the definition has been. First, it would help to summarize the data required.

Contractor's resource consumption data is summarized in Table 8.18 and computer resource description and utilization data is shown

TABLE 8.18
CONTRACTOR RESOURCE CONSUMPTION DATA

In each category, estimated costs are to be reported once, at the beginning of the contract, except as noted. Actual costs are to be reported monthly, except as noted.

	Notes
Direct Labor (in dollars and in man-hours)	1, 2, 3
Computer Resources	
Computer Hours	1
Investment	4
Operations	
Unique Facilities and Equipment	
Investment	4, 5
Operations	6
Overhead Rate	
G&A and Fee Rates	

NOTES:

1. Disaggregated by product component as in Table 8.14.
2. Estimates disaggregated by skill category as in Sec. 8.6.4.
3. Estimates revised at PDR and at CDR.
4. Actuals reported when incurred.
5. Disaggregated by type of facility.
6. Disaggregated by type of facility or equipment.

in Table 8.19. Software product description and status data were previously summarized in Table 8.1.

The first thing to note is that the frequency of reporting is different for the different data items. One-time reports or those produced at specific milestones, though detailed, should not be viewed as a hardship by the contractor or the Government. They are generally descriptive or represent back-up data for cost estimates which should have been developed for a credible bid.

Some examples may prove helpful. For the software product description and status, the following milestone-related reports are required (Table 8.1):

Computer Program Development Plan at Contract Award
CPC Definition and CPCI Update at PDR
CPC Update at CDR
Module Definition at CDR
Start Qualification Test at PQT
Completed Product at PCA
Installation at IOT&E

The first three are estimates and represent a detailed description of the product to be developed. Details are increased as the design is completed (CDR) and actuals are recorded against the estimates at Start of Qualification Test and PCA. The report submitted at PCA is a complete description of the product (Table 8.2), including descriptive information, development method, milestones, code size, code changes, code structure, etc. Modifications to the product are repeated at site installation. Data is detailed, but not beyond what can be collected. Since it is not repetitive, it should not be a burden.

Similarly, estimates of resource consumption (Table 8.18) are reported in depth at contract award and are refined at PDR and CDR.

TABLE 8.19

COMPUTER RESOURCE DESCRIPTION AND UTILIZATION

Item	Notes	Reporting Frequency
Computer System Description	Tables 8.16 and 8.17	One Time
System Utilization	Hours per week (exclude time-share service)	Monthly
System Availability	Hours per week	Monthly
Average Turnaround (Batch-type)	Time out, minus time in, minus execution time	Monthly
95th Percentile Turnaround (Batch-type)	Time out, minus time in, minus execution time	Monthly
Average Turnaround (Time-share)	Terminal time ÷ cumulative core execution time	Monthly
95th Percentile Turnaround (Time-share)	Terminal time ÷ cumulative core execution time	Monthly

Note: Separate records should be maintained for each computer system (e.g., development versus maintenance).

The end-item level becomes more disaggregated as the product is further defined and skill category data are required.

Actual investment costs of facilities, equipment, and computer hardware and software packages are reported once, at the time they are acquired. Similarly, a detailed description of the computer system is required, one time.

These requests, though detailed, are reasonable because of their infrequency.

For repetitive reporting items, we have attempted to keep the data as aggregated as possible, without losing significant information content. Still, due to the repetitive nature, the reporting requirement is an imposition, and the ease with which the data can be automatically processed must be addressed.

For product status information, requests during development are minimal. When a source deck is first baselined, size and code structure information is required (Table 8.2, F and H). If the contractor feels this is a burden, then the Air Force need merely request a copy of the source code and a listing, which is hardly unreasonable. Information can then be extracted and a determination that the code at least compiles successfully can easily be made.

During qualification testing, we recommend that detailed information on code changes be tracked. Information can be produced by an automated library system and access should not be a problem at this stage of the development, since Air Force representatives are present during this time.

During maintenance, the product description should be kept current through monthly maintenance change reports (Table 8.3) and updates of the

product description (Table 8.2), when new models are incorporated. The data seem a reasonable part of configuration control, and should be viewed in that manner.

Similarly, computer resource utilization information will be required monthly (Table 8.19). Reporting can easily be automated, since tracking is system-wide. Such reports are currently being generated in far more detail at SDC.

The only difficult items to track are cost, man-hour, and computer-hour data required in Table 8.18 at levels specified in Table 8.14. The resource elements themselves will probably not be a problem and can easily fit into an already existing reporting code. The real difficulty is assigning costs, man-hours, and computer-hours to the CPC level for some activities. This will result in an increase in the reporting effort required. However, it is the only method of building up cost histories on a standard list of software components, so that the software costing job can eventually be simplified. The estimator will eventually be able to build the total estimate from estimates of the software components. The data base will provide detail for component estimation and parametric techniques (as well as analogy techniques). Sizing as well as costing should become more a science and less an art.

Furthermore, we have carefully selected activities to be tracked at the CPC level so that artificial allocations are not required and information is technically easy to track. Also, CPCs have been selected so that they formed logical work packages. Thus, we believe the expense will not be severe and the information will be well worth the cost.

Contractor cooperation may remain a problem. Contractors simply may not want the Air Force to have this much visibility. In response, we suggest that the results could be made part of the public literature so that each contractor will have the opportunity to improve his

estimating techniques. Of course, data will have to be aggregated to protect contractor confidentiality. Perhaps CERs would provide the best means of presenting this information.

Another question is whether or not there is a convenient way of reporting the data. Fortunately, there is an automated reporting system available. It is described in AFSCM 173-4.[42] The basic reporting elements for electronic systems are shown in Table 8.20.[41]

The Program Breakdown Structure of Table 8.20 was initially derived from the Work Breakdown Structure of Mil Std. 881A. It was created to record the total cost of electronic systems, both hardware and software. At the time of its creation, software was a minor cost item; hence only one element, 4210, is solely devoted to software. Even this element does not include all software costs. For example, some software testing costs are included in 1050 and some software analysis costs are included in 1061.

John Glore of the MITRE Corporation has been modifying this reporting system to record costs of the different software activities and phases in such a way that total software costs could be aggregated under the Program Breakdown Code (PBC) 4200. He found that the reporting structure had room for more levels than shown in Table 8.20 (which was extracted from Glore's report).[41] His suggestions are reported below, after which we show how our reporting system elements can be related to Glore's suggestions.

TABLE 8.20

STANDARD WBS ELEMENTS FOR ELECTRONIC SYSTEMS

(Ref. 41, Table A2)

Level	PBC	Standard Element Name
1	1000	Electronic System
2	1010	Prime Mission Product
3	1110	Integration and Assembly
3	2110	Sensors
3	3110	Communications
3	4110	Automatic Data Processing Equipment
3	4210	Computer Programs
3	4310	Firmware
3	5110	Data Displays
3	6110	Auxiliary Equipment
3	8110	Air Vehicle
2	1020	Training
3	1021	Equipment
3	1027	Facilities
3	1029	Services
2	1040	Peculiar Support Equipment and Maintenance (including Maintenance Concept)
3	1041	Organizational/Intermediate
3	1044	Depot
3	1049	Other
2	1050	Systems Test and Evaluation
3	1051	Development Test and Evaluation
3	1053	Operational Test and Evaluation
3	1052	Combined DT&E and OT&E
3	1055	Mockups
3	1056	Test and Evaluation Support
3	1057	Test Facilities
3	1059	Other System Tests
2	1060	System Program/Project Management
3	1061	Systems Engineering Management
4	1061A	Reliability
4	1061B	Maintainability
4	1061C	Parts Control
4	1061D	Nomenclature
4	1061E	Aerospace Environment
4	1061F	Transportability
4	1061G	Electromagnetic Compatibility
4	1061H	Radar Frequency Management
4	1061J	Security
4	1061K	Survivability/Vulnerability
4	1061L	System Safety

TABLE 8.20 (Continued)

Level	PBC	Standard Element Name
4	1061M	Communications Long Lines
4	1061N	Radio Frequency Management
4	1061P	Value Engineering
4	1061Q	Availability
3	1062	Supporting Project Management Activities
4	1062A	Program Management
5	1062AA	Program/Contract Work Breakdown Structure
5	1062AB	Cost Information System
5	1062AC	Cost Schedule Systems
5	1062AD	Life Cycle Costs
5	1062AE	Schedule Management
4	1062B	Manufacturing Management
4	1062C	Configuration Management
4	1062D	Integration of Analysis and Related Computer Support
4	1062E	Quality/Inspection
4	1062F	Photographic Documentation
4	1062G	STINFO
3	1063	Integrated Logistics Support
4	1063A	Preoperational Supply Support
4	1063B	Packaging
4	1063C	Transportation
4	1063D	Travel
4	1063E	Maintenance
4	1063G	Limited Spares/Repair Parts Provisioning
3	1064	Crew/Human Factors
4	1064A	Human Engineering
4	1064B	Biomedical/Life Support Equipment
4	1064C	Manpower/Personnel Requirements
4	1064D	Human Factors Test and Evaluation
2	1070	Data
3	1071	Technical Publications
3	1072	Engineering Data
4	1072B	Engineering and Configuration Documentation
4	1072H	Human Factors
4	1072R	Related Design Requirements
4	1072S	System/Subsystem Analysis
4	1072T	Test
3	1073	Management Data
4	1073A	Administrative Management
4	1073F	Financial
4	1073L	Logistic Support
4	1073P	Procurement/Production

TABLE 8.20 (Continued)

Level	PBC	Standard Element Name
3	NONE	Support Data
3	1074	Data Repository
2	1080	Operational/Site Activation
3	1081	Contractor Technical Support
3	1082	Site Construction
3	1083	Site Conversion
3	1084	System Assembly, Installation and Checkout on Site
3	1085	ADP Support Facilities
3	1089	Other Support Facilities
2	9200	Common Support Equipment
3	NONE	Organizational/Intermediate (including Equipment Common to Depot)
3	NONE	Depot
2	NONE	Industrial Facilities
3	NONE	Construction/Conversion/Expansion
3	NONE	Equipment Acquisition or Modernization
3	NONE	Maintenance
2	9600	Initial Spares and Repair Parts
3	NONE	(Specify by Allowance List, Grouping, or Hardware Element)

Glore's Interim Standard PBCs have the form: s421xxy, where

s is a letter (A, B, ...) that identifies the software's supplier

421 identifies the element as a software product

xx is an alphanumeric code* that designates the software type (for example the CPC identifier)

y when used, is a letter (A-F) that identifies the life-cycle phase to which the element applies; if y is not used, the element is presumed to encompass all phases covered by the contract.

These PBCs apply to all the direct costs of software development (or the purchase or rental of programs). In addition, Glore defined the following PBCs for software-related costs, analogous to the Level 2 categories for hardware-related costs in Table 8.20:

4220 Software-peculiar training

4240 Equipment required specifically for software development or maintenance

4250 Testing of software (includes PQT and FQT)

4260 Software-peculiar management and engineering

4270 Software documentation

4285 Software development and maintenance facilities

4290 Other software-related costs

4200 Summary of all software-related costs

Note that these PBCs have only four digits and are not expanded like the "421" PBCs.

*The first "x" is any alphanumeric character but "I" or "O" and the second is a numeral.

Glore also suggests adding three (or more) digits or letters to each element, after a "/" to designate the segment, functional area, CPCI, etc.* Thus, for example, 42134C/123 would cover the coding and checkout (life-cycle phase C) of CPC 34 in system segment 1, functional area 2, CPCI 3.

Glore suggests that where a particular item (e.g., management) includes both software and hardware costs, it should be allocated between the corresponding PBCs. Consistency is hard to maintain in this situation, however, and we recommend that only purely software items, or items whose software components can be explicitly separated and consistently tracked, be reported to the software level. Other costs should be collected as system-wide costs and allocated between software and hardware with a general overhead factor in CERs.

In general, we were able to fit our reporting elements (Table 8.6) into Glore's framework. For direct labor man-hours we suggest the assignments shown in Table 8.21. As can be seen, there is no category for software support personnel not directly chargeable to resource elements; we suggest defining a new category, 4230. A less desirable alternative is to charge these personnel to general overhead. We suggest charging the costs of operating this reporting system itself to project management (4262A) and program management documentation (4273). The testing PBC (425z), we suggest, should include only the resources required for explicitly testing software; testing that includes other components of the system should not be charged here.

We have made some modifications of Glore's definitions. First, we use the "y" in the PBCs 421xxy to designate resource elements instead

*"Segment" and "Functional Area" are more aggregated levels of the system that includes the software.

TABLE 8.21

REPORTING CODES FOR DIRECT LABOR HOURS

(Resource Element Definitions - Table 8.6)

Resource Elements	Interim PBC
CPC LEVEL	
Design	421xxB/abc
Coding	421xxC/abc
Integration	421xxD/abc
Installation	421xxE/abc
Maintenance	421xxF/abc
("xx" identifies the particular CPC; "abc" identifies the segment, functional area, and CPCI)	
CPCI LEVEL	
Design	42100B/abc
Testing Independent V&V	425z/abc

Note:

	Devel.	Deployment	Operation
z = Test	1	2	3
z = V&V	4	5	6

SYSTEM LEVEL	
Analysis	42100A/ab
Testing Independent V&V	425z/ab (see Note above)
Management	
Engineering	4261/ab
Configuration	4262C/ab
Project	4262A/ab
Training	4229/ab
Documentation	
Technical	4272/ab
Configuration	4274/ab
Program Management	4273/ab
Support	4230/ab (or charge to overhead)

of life-cycle phases (milestone definitions). The time of reporting suffices to identify reported costs with life-cycle phases. Our use is in keeping with the activity definition of life-cycle terms (i.e. analysis, design, etc.) We have designated 42100y to be an aggregated category for CPCI level or system-level activities.

Also, we have added analysis and design to the 421 codes, leaving only engineering management reported against 4261 instead of "engineering activities" as in Glore's original definition.

Reporting codes for costs other than direct labor are shown in Table 8.22. Costs include operation as well as acquisition. Since acquisition costs are also reported separately (Table 8.19), operating and investment costs can be separated if desired. Also, since availability is reported separately, an operating cost for the specified system can be derived as a function of availability.

Training facilities have been included as an item, although we do not expect separate software training facilities as a rule. This item, too, will include investment as well as operation. If new facilities are not required, then a rental charge should be used.

One remaining (and important) item has not been covered. Computer-hours should be broken out in the same manner as direct man-hours. We recommend reporting them under the 421 PBCs, distinguished from man-hours by using the letters G-L instead of A-F in the "y" position.

In summary, we are using the following definitions for PBCs:

4210: Computer Programs. All analysis, design, coding, and error correction during testing and maintenance. Also includes code modification for site-specific requirements. Costs of programs which are acquired instead of developed should also be included.

TABLE 8.22

REPORTING CODES FOR OTHER COSTS

Resource Elements	Codes
Computer Resources	
Development	4249A
Maintenance	4244A
Unique Facilities	
Development	4285A
Maintenance	4285B
Training	4227
Unique Equipment	
Development	4249B
Maintenance	4244B
Training	4221

4220: Software-Peculiar Training. Preparation of material and conduct of courses to instruct personnel in the operation of the software (e.g., analyzing diagnostics). Also includes demonstration of software when demonstration is aimed at instruction.

4240: Software-Peculiar Support Equipment. Special equipment required to develop and/or maintain the software. Equipment not required for development will be charged after the beginning of Qualification Test. Equipment required for deployment will not be included in the category even though used in development; separate one-time report describing such equipment should be included as a separate data item. Equipment related to the computer resources will be separated from other software-peculiar support equipment. Operating costs will also be reported here, including personnel to operate and maintain the computer center. The actual facilities, building, grounds, air conditioning, etc. will be reported under 4285.

4250: Testing of Software. Man-hours and costs necessary for the testing of software after initial debugging of the source deck. Specifically excludes computer resources (4240) or facilities (4285) necessary to perform the tests. Also excludes redesign and recoding to correct errors (4210). Includes time necessary to plan and prepare test runs. Formulation of test plans is also included, but test documentation goes under 4270.

4260: Software-Peculiar Management and Engineering. Management has been divided into three parts: engineering (4261) configuration (4262C), and project (4262A). Engineering management costs will be charged almost entirely by technical managers not assigned to specific sections of code. Special studies will largely be picked up in analysis and design (4210). However, there will still be requirements for preparation and conduct of technical review meetings and general technical management. This will generally be easy to separate from hardware.

Configuration management costs are those associated with the control of baselined software and probably the maintenance of the library. They exclude the writing of special reports, which is reported under 4270. Items are easily separable from hardware.

Project management includes financial, scheduling, and other non-technical management functions. This may be so integrated with hardware that separation is impossible. If so, do not report separately from 1060. An overhead rate can be applied.

4270: Documentation. This element includes all reports that can be separately identified with software. It is separated into three parts: technical, configuration, and program management. Technical documentation includes all technical reports (e.g., Subsystem Design Analysis Reports, Trade Study Reports, Test Plans, Test Procedures, Test Results, User's Manuals, etc.).

Configuration documentation includes documentation of the Part I and Part II Specification, data base, special status reports, etc.

Program Management documentation may not include any purely software reports. Perhaps the reporting system described here will be separated from other Cost Reports. If so, its cost could be shown here.

4285: Development and Maintenance Facilities. This element includes the acquisition and operating costs of special facilities. If facilities are shared, a rental charge will represent acquisition. No man-hours for facilities maintenance and operation are required.

We have now shown that an automated reporting system is available and can be used to collect required man-hour, cost and computer-hour data on a regular basis. Even so, the reader is reminded that the limitation on what should be reported is directly related to the requirement for cooperation by contractors. The reporting requirements are within

contractors' technical capability; however, their willingness may be another matter. We have tried to find a level of detail which meets Government goals and yet requires only a moderate addition to the contractors' effort.

8.9 ENGINEERING CHANGE PROPOSALS

A problem which has been ignored to this point is how to handle changes to requirements as represented by ECPs changing the Part I Specifications. They present a special problem in that they represent an approved change to the flow of normal development and maintenance activities. They should be isolated and their impact separately measured. Thus, they should be treated as a separate development with their own reporting system.

During maintenance, this is a reasonably straightforward task. Some care has to be taken in evaluating the results, however. First, as discussed in Sec. 8.6, most analysis has taken place prior to the formal submission of the ECP (Table 8.13). Therefore, few if any analysis hours are involved. Secondly, non-critical errors discovered during qualification testing are often treated as maintenance problems to be solved with the next ECP. Third, management hours will be hard to separate from the ongoing maintenance activities and, except for configuration management, should not be attempted. Instead, they should be treated as an overhead item. Equipment and facilities have similar problems.

All of the above understate the cost of the ECP, but adjustments can be made with overhead rates when comparing ECPs to new developments. As long as what is excluded is understood, no problem should arise.

It is important to note that a model change (product improvement contract) may close out a number of ECPs and Discrepancy Report Forms (DRFs). In this case, it is the model change that should be tracked and not each separate ECP. However, all work on a particular ECP should

be included under only one model revision contract. Furthermore, reporting should separate error corrections (DRFs) from product improvement (ECPs), making it possible to separate maintenance from the model change.

During development, the ECP problem is much more complicated because the impact of the ECP does not have a specific end date with the delivery of a product. Instead, the ECP will be integrated into the development program, affecting some completed tasks, causing work to be modified or redone, and to a lesser extent affecting tasks not initiated, by changing requirements or scope.

We recommend that each ECP be tracked separately until it is totally integrated with the rest of the development. This can be done by identifying the portion of the software made obsolete by the requirement change and estimating the resources required to replace the obsolete portion and add the new capability.

Cost (man-hours and computer-hours) should then be tracked until the work has been redone. Thus, integration and testing for the ECP would not be reported if the ECP were introduced during design and the changes were incorporated prior to module baselining.

8.10 PRE-CONTRACT ACTIVITIES

One final point should be made that will be useful when using the results of this reporting system. A development is often made up of a series of contracts, not just one (for full-scale development). When the history of the development of the total defense system is finally put together, all contracts, with their separate costs and histories, will have to be integrated for comparative purposes.

Thus not only should the full-scale development contractor be reported, but so should the independent V&V contractor, as well as the

concept formulation and validation contractors, if software analysis has been included in those phases or computer-program prototypes developed.

It should be noted that the automated reporting system discussed in AFSCM 173-4 has provision for multiple contractors. Different letters are used at the front of the code. Thus, the code is actually s421xxy/ABC, where s designates the supplier.

9 CONCLUSIONS AND RECOMMENDATIONS

In the past months GRC has studied the problems of software cost estimation, hypothesizing relationships, gathering and analyzing data, and examining reporting systems. In this section, significant contributions to the improvement of software cost estimating and data collection are summarized (Sec. 9.1). The subject is large and much remains to be done. Our recommendations for future work are presented in Sec. 9.2.

9.1 CONCLUSIONS

Significant findings of this study are organized around the two major topics of the study: (1) the definition of resource elements, and (2) the development of cost estimating relationships.

In Sec. 8, our recommendations for resource data elements are presented. They call for the regular collection of data on costs, man-hours, computer-hours, and completion status. Other information is also specified, such as a detailed description of the software product, the characteristics of the computer system used to develop or maintain the software, etc. We recommend collection of this latter information at specified milestones only. Its level of detail is much greater than that required of the regularly reported data; however, the infrequency of reporting should ease the burden on the contractor.

Important innovations in the resource element definitions include the following:

1. A moving milestone (source baseline) providing a continuous measurement of progress towards software completion. This milestone separates coding and checkout from test and integration. Each time a source deck is completed (coded and debugged), the deck is baselined and future work on that part of the program will be recorded against test and integration. Thus, work in these two phases will go on concurrently and data will be continuously available on the percentage of

the product which has finished initial coding and checkout and is in the test and integration phase.

2. No requirement to separate redesign and recoding efforts in the Test and Integration Phase. We speculate that this requirement in prior cost reporting systems has been one of the reasons for their non-implementation. Redesign and recoding in this period are often done by the same person and therefore difficult to separate. If separate reporting were required, it would probably be artificial, so the information content is suspect at any rate.

3. A preliminary standardized list of end items (CPCs) for comparing resource consumption among software developments. This list will serve to disaggregate the software product into meaningful parts. Ultimately, software will be described by its parts, with estimates made at the parts level and then aggregated, much as hardware estimates are made today. The list is preliminary in that we are confident that additions will be made upon review by the software community.

4. Separation of maintenance into error response and product improvement. Only error response should be associated with the original software development. Product improvement (through ECPs) should undergo separate approval and be subject to the same data gathering as any other development.

5. Identification of differences in "milestone" and "activity" definitions for Analysis, Design, Coding, and Testing. The inconsistent use of these terms has been a source of confusion during the entire effort, and undoubtedly in other data gathering and analysis attempts. Ultimately, different terms for the milestone and activity uses should probably be selected.

In addition to defining the data elements, we have demonstrated one way of mapping these elements into an already existing reporting system. Hence the mechanics of data retrieval will not be severe.

In summary, we believe these innovations will improve the chances that contractors will accept a reporting system, without sacrificing the needs of the Program Offices for cost control data and the Air Force's long-term needs for better data.

Our second major objective was to develop improved software cost estimating relationships. A significant amount of work had been previously devoted to this task and reported in the literature. This previous work was performed by competent groups and focused on estimating total man-hours or costs. Results had been disappointing, with derived relationships exhibiting large variances.

As a result, we were directed to take a new approach and focus our attention on the development of estimating relationships for each of the life-cycle phases. Our efforts have been concentrated on man-hour estimation. During the contract, we first hypothesized relationships. We then identified, collected, and analyzed data. Important findings include the following.

1. Accurate estimating relationships for each life-cycle phase cannot be developed independent of the other phases. Data when plotted exhibits significant variation. As a consequence, an estimate of total man-hours made by estimating each life-cycle phase and then adding will have less precision than that made on the total.

2. Estimating the trade-offs between the life-cycle phases is of prime importance. Only then will the variation in the total man-hour estimate be reduced, i.e., the accuracy of the estimating relationship be increased. In effect covariances must be incorporated in the estimators. Reducing the risk in software cost estimates is dependent on quantifying these interrelationships between the life-cycle phases.

3. Estimating the trade-offs can also lead to the development of rules for optimal allocation of resources among life-cycle phases. Quantifying the trade-offs between life-cycle phases makes it possible to derive such rules. Application of the rules, in turn, will lead to less costly or lower-risk developments. In effect, these rules will represent estimating relationships for software developments with an efficient allocation of resources. Note that we are in effect asserting that much of the variance in previous estimating techniques is the result of mixing software developments which have inefficient as well as efficient allocations of man-hours (among life-cycle phases) in the same data base.

Although the data collected was insufficient to quantify these trade-offs, we were able to locate a data base (PARMIS) that offers promise. Detailed man-hour data has been maintained in PARMIS for 2,000 projects. The development environment, language, and application areas are all similar, so that these significant parameters will not cloud the search for trade-off relationships.

In addition to the life-cycle phase work we have examined, in aggregate, a number of basic relationships which can be incorporated into today's estimating techniques. Among those examined are:

1. The similarity between the development processes for "ADP" and defense-system software. It is our belief that a model which explains ADP software costs will also be a good predictor of defense-system software costs. The coefficients of the model will most certainly change, but the model forms can be the same because the procedures are similar. Therefore, we examined the relationship of man-hours to lines of source code for a mixture of software developments, stratifying on type. Defense-system programs were more expensive, but followed the same trend as ADP programs.

2. The relationships of aggregate software cost to key driving variables such as size, language, space constraint, and requirements changes (ECPs) during development. Results are given in Sec. 6.3.

3. The relationships between different measures of software product such as lines of source code, size of Part II Specifications, and number of object instructions. Conversion ratios between object and source code for different languages and machines were developed.

4. The magnitude of maintenance activities and the separation of error correction from product improvement. Examples were given in which product improvement was the major consumer of resources during maintenance. Relationships for estimating error rate were derived, and the difference in maintenance manning requirements for source and object programs was quantified.

9.2 RECOMMENDATIONS FOR FUTURE WORK

Based upon this study, we believe the following efforts will best contribute to the complicated job of software cost estimation.

First, and of utmost importance, implement a cost reporting system. If the SDC system,[3] developed in 1967, had been implemented, we would have had a usable cost data base today. We hope our suggested elements are the basis for the approved cost reporting system, as we believe they avoid some of the impediments to contractor acceptance, and offer a great deal of control for the Program Offices. However, the important thing is to implement some system. Furthermore, we agree with Devenny[1] that use of the reporting system must be a contractual requirement.

Second, we recommend that the Air Force continue the study of the PARMIS data base begun in this contract. PARMIS is the only readily available data base in which the trade-offs between phases can be examined. Although our results to date are quantitatively inconclusive,

due to the small sample size extracted from that data base, the existence of the trade-offs has been demonstrated. Furthermore, the fact that PARMIS is uniform with respect to major cost driving parameters such as language, applications, etc., and the detailed information available on the product developed, offer a real opportunity to quantify these interrelationships.

The development process for ADP software at AFDSDC is similar to that for major defense system software. Therefore, the functional forms of the relationships, if not the relationships themselves (with some scaling rules), can be used for defense system applications. It is most important to quantify these relationships because they are the key to identifying proper resource allocations among the life-cycle phases and thereby achieving risk reduction in software developments.

Third, and concurrently, a systematic collection of data from completed projects should be initiated. Although this is expensive and time-consuming, we cannot afford to wait 10 to 15 years until the data from a reporting system implemented today is usable for cost estimation.

APPENDIX A

SAMPLE PARMIS QUESTIONNAIRE

1. Project Originator Number (PON)_____

2. Project Description_____

_____

_____

3. Size of system described by PON:

   - Computer words of instructions_____
   - Computer words of instructions and data_____
   - Number of relocatable object instructions_____

4. Computer System (Manufacturer and Model):

   - Development_____
   - Operation_____

5. Computer Word Size:

   - Development_____
   - Operation_____

6. Constraints (space, time, core, peripheral storage, etc):

   - Development_____
   - Operation_____

7. Date baseline established:

   - Functional Specs_____
   - Design Specs_____
   - Source Programs_____

8. Programming Languages:

Language	Lines of Source Code
_____	_____
_____	_____
_____	_____

9. Assembler and Compiler:

- Was the assembler hosted by the target machine?_____
- Was the compiler hosted by the target machine?_____

10. Development Methods Employed:

- Top Down Design_____
- Chief Programmer Teams_____
- Structured Programming_____
- Software Development Libraries_____
- Other_____

11. Software Development:

- What % of software was based on requirements adapted from an existing system?_____
- What % of software was based on design adapted from an existing system?_____
- What % of code was adapted from existing code (e.g., translated from another language)?_____
- What % of code already existed?_____
- What special hardware and/or software items were provided specifically to support the development of the software (e.g., purchase of development software)?_____

  Cost?_____

12. Verification and Validation (V&V) tools:

- Was a separate V&V organization used?_____
- Were automated V&V tools used by:

  The developing organization_____

  A V&V organization_____

13. Software Installation:

- Number of installation sites_____
- Number of subsystems requiring modifications, by site_____
- Time required for installation, by site_____
- Total cost of installation, by site_____
- Total man-hours for installation, by site_____
- Total computer hours/dollars for installation, by site_____

14. Operation:

- Number of man-hours/month of the operational lifetime required to operate the software component of the system (i.e., not users, but operators)_____
- Cost per man-hour_____
- Planned system operational lifetime_____

15. Maintenance:

- Number of hours per operational month for maintenance:

  Functional Analyst_____

  Data Systems Analyst_____

  Programmer_____

  Support Personnel_____
- Computer hours/dollars for maintenance for each month of operational lifetime_____

16. Software Faults:

- For each software fault requiring maintenance action:

  Date reported_____

  Date closed_____

  Lines of Code_____

  Was documentation changed?_____

  Priority of fix activity_____

  Man-hours_____

  Computer hours_____

17. Software Changes:

- For each formal change order affecting software during development:

  One-line description_____

  Date filed_____

  Date approved_____

  Date work started_____

  Agency originating change_____

- For each subsystem affected by the change:

  Name_____

  Were functional specs changed?_____

  Were design specs changed?_____

  How does the change affect:

  - Work already completed?_____
  - Work remaining?_____

  How were milestones affected?_____

18. Computer Time:

- Cumulative computer time at each major milestone:

  Design completion_____

  Coding and debugging completion_____

  Unit testing completion_____

  Integration testing completion_____

  Environmental Systems Test I completion_____

  Environmental Systems Test II completion_____

19. Fault Isolation:

- Number of man-hours per month of operational lifetime required to isolate faults within the system_____
- Cost per man-hour_____
- Total number of anomalies reported by month_____
- Fraction of reported anomalies ultimately attributed to software, reported by month_____
- For each fault isolated:

  Date reported_____

  Date identified_____

  Action required to fix:

  - Low priority software fixes_____
  - High priority software fixes_____
  - System change notices_____
  - No action_____

  Date corrected_____

20. Documentation:

- Number of pages of functional specifications devoted to this system or subsystem when project initially approved_____
- Dates and affected numbers of pages of the functional specifications for each change during development_____
- Number of pages of first design specifications generated for this system when approved or baselined_____
- Dates and affected numbers of pages of each subsequent change to the design specifications_____
- Number of pages of programmer documentation prepared for this system when first released_____
- Dates and affected number of pages of all changes to programmer documentation prior to system release_____
- Number of pages of user documentation prepared for this system when first released_____
- Dates and affected numbers of pages for all changes to user documentation prior to system release_____

APPENDIX B

SOFTWARE RELIABILITY BIBLIOGRAPHY

Itoh, D., and T. Izutani, "FADEBUG-I, A New Tool for Program Debugging", 1973 IEEE Symposium on Computer Software Reliability, New York, NY, April 30 - May 2, 1973, pp. 38-43.

Schneidewind, N.F., "Analysis of Error Processes in Computer Software", ACM SIGPLAN Notices (Proceedings of the International Conference on Reliable Software), Vol. 10, No. 6, June 1975, pp. 337-346.

Shooman, M.L., "Operational Testing and Software Reliability Estimation During Program Development", 1973 IEEE Symposium on Computer Software Reliability, New York, NY, April 30 - May 2, 1973, pp. 51-57.

Littlewood, B., and J.L. Verrall, "A Bayesian Reliability Growth Model for Computer Software", 1973 IEEE Symposium on Computer Software Reliability, New York, NY, April 30 - May 2, 1973, pp. 70-77.

Jelinski, Z., and P.B. Moranda, "Software Reliability Research", Statistical Computer Performance Evaluation, W. Freiberger (ed.), Academic Press, New York, NY, 1972, pp. 465-484.

Shooman, M.,L., "Probabilistic Models for Software Reliability Prediction", Statistical Computer Performance Evaluation, W. Freiberger (ed.), Academic Press, New York, NY, 1972, pp. 485-502.

Schneidewind, N.F., "An Approach to Software Reliability Prediction and Quality Control", AFIPS Conference Proceedings, 1972 Fall Joint Computer Conference, Vol. 41, AFIPS Press, Montvale, NJ, pp. 837-847.

Coutinho, J., "Software Reliability Growth", 1973 IEEE Symposium on Computer Software Reliability, New York, NY, April 30 - May 2, 1973, pp. 58-64.

Boehm, B.W., "Some Experience with Automated Aids to the Design of Large-Scale Reliable Software", IEEE Transactions on Software Engineering, Vol. SE-1, No. 1, March 1975, pp. 125-133.

Musa, J.D., "A Theory of Software Reliability and Its Application", IEEE Transactions on Software Engineering, Vol. SE-1, No. 3, September 1975, pp. 312-327.

Endres, A.B., "An Analysis of Errors and Their Causes in System Programs", IEEE Transactions on Software Engineering, Vol. SE-1, No. 2, June 1975, pp. 140-149.

Rubey, R.J., et. al., "Quantitative Aspects of Software Validation", IEEE Transactions on Software Engineering, Vol. SE-1, No. 2, June 1975, pp. 150-155.

Shooman, M.L., and M.I. Bolsky, "Types, Distribution, and Test and Correction Times for Programming Errors", ACM SIGPLAN Notices (Proceedings of the International Conference on Reliable Software), Vol. 10, No. 6, June 1975, pp. 347-357.

Trivedi, A.K., and M.L. Shooman, "A Many-State Markov Model for the Estimation and Prediction of Computer Software Performance Parameters", ACM SIGPLAN Notices (Proceedings of the International Conference on Reliable Software), Vol. 10, No. 6, June 1975, pp. 208-220.

Littlewood, B., "A Reliability Model for Markov Structured Software", ACM SIGPLAN Notices (Proceedings of the International Conference on Reliable Software), Vol. 10, No. 6, June 1975, pp. 204-207.

Miyamoto, I., "Software Reliability in Online Real Time Enviornment", ACM SIGPLAN Notices (Proceedings of the International Conference on Reliable Software), Vol. 10, No. 6, June 1975, pp. 194-203.

Bloom, S., et. al., "Software Quality Control", 1973 IEEE Symposium on Computer Software Reliability, New York, NY, April 30 - May 2, 1973, pp. 107-116.

McGeachie, J.S., "Reliability of the Dartmouth Time Sharing System", 1973 IEEE Symposium on Computer Software Reliability, New York, NY, April 30 - May 2, 1973, pp. 117-123.

Stearns, S.K., "Experience with Centralized Maintenance of a Large Application System", COMPCON75 Digest of Papers, San Francisco, CA, April 25-27, 1975, pp. 281-288.

Buda, A.O., et. al., "Implementation of the ALPHA-6 Programming System", ACM SIGPLAN Notices (Proceedings of the International Conference on Reliable Software), Vol. 10, No. 6, June 1975, pp.371-381.

Akiyama, F., "An Example of Software System Debugging", Proceedings of the IFIP Congress 71, Vol. 1, North-Holland Publishing Co., Amsterdam, 1971, pp. 353-359.

Yourdon, E., "Reliability Measurements for Third Generation Computer Systems", Proceedings of the 1972 Reliability and Maintainability Symposium, San Francisco, Calif., January 1972, pp. 174-183.

Shooman, M.L., "Quantitative Analysis of Software Reliability", Proceedings of the 1972 Reliability and Maintainability Symposium, San Francisco, Calif., 1972, pp. 148-157.

Schneidewind, N.F., A Methodology for Software Reliability Prediction and Quality Control, Naval Post Graduate School, NPS-55SS72111A, Monterey, Calif., November 1972.

Ellingson, O.E., A Predictor Tool for Estimating the Confidence Level of a Computer Program Subsystem in the Space Programs Department, System Development Corporation, Technical Memo TM-(L)-3335/000/00, Santa Monica, Calif., December 1967.

APPENDIX C

JOB DESCRIPTIONS

C.1 MANAGER

Managers direct and control the performance of technical staff so as to satisfy the customer's technical and contractual requirements within the project budget. They require the skills to perform the following functions:

- Specify and schedule tasks to fulfill project requirements.
- Recommend and negotiate for technical staff to perform the tasks.
- Direct technical and cost performance of the project team to ensure meeting the project's goals and objectives.
- Monitor the preparation of reports and briefings required by contract or requested by customer.
- In some cases, serve as a task leader or technical contributor.

Managers are classified according to level of experience, expertise, and management skills as follows:

1. Project. Administratively directs and controls an entire project. Usually a "pure" manager who makes little or no technical contribution.
2. Technical. Technically directs and controls an entire project. Concerned with those technical aspects of the project that influence management decisions.
3. Configuration. Ensures "quality control" of the software product. Manages the process of baselining specifications and the source code as it progresses through the phases of the system life cycle.

Managers may be selected from most of the other job categories identified in this section, depending upon the specific requirements of the project.

C.2 SCIENTIST AND ENGINEER

These persons perform basic and applied research; participate in the analysis and design phases of system development; and assist in the evaluation of systems. No attempt is made here to differentiate between specialties (e.g., electronics, propulsion systems, communications). We simply identify a general category of scientists and engineers who possess the skills to perform the following functions:

- Develop general system concepts and procedures.
- Specify formulas and algorithms to be implemented.
- Design, develop, and test prototypes of critical elements.
- Analyze available equipment to evaluate its suitability.
- Provide technical guidance to persons responsible for developing the software required to support the operation of the system.

Scientists and engineers are classified according to level of experience, expertise, and management skills as follows:

1. Senior Scientist or Engineer. Directs and coordinates all technical activities necessary to complete a single large scientific or engineering project, or several smaller projects.
2. Scientist or Engineer A. Directs a major technical portion of a large project, or an entire project of lesser complexity than those normally assigned to a senior scientist or engineer. May also supervise other technical personnel, and may be responsible for the administrative duties of a small

work group, although the technical work performed is more important than the supervision.

3. Scientist or Engineer B. Analyzes complex scientific or engineering problems with minimal supervision and guidance.

4. Scientist or Engineer C. Analyzes scientific or engineering problems under the immediate supervision of a higher-level person.

C.3 SYSTEMS ANALYST

Systems analysts require the technical skills to perform the following functions:

- Confer with officials, scientists, and engineers to facilitate understanding of the operational needs and goals of a new application or refinement of an existing application.
- Formulate a comprehensive statement of a problem and develop, review, and recommend methods, procedures, and systems to solve it.
- Develop specifications for computer hardware and software, including block diagrams, flow charts, record and form layouts, etc.
- Verify that the software design meets each of the operational needs and goals.
- Document the final design in a manner suitable for use in programming.
- Conduct follow-up studies (as needed) to ensure that the evolution of the software during the development process has not altered the design to the point that it no longer satisfies system requirements.

AD-A053 020 GENERAL RESEARCH CORP SANTA BARBARA CALIF F/G 9/2
COST REPORTING ELEMENTS AND ACTIVITY COST TRADEOFFS FOR DEFENSE--ETC(U)
MAY 77 C A GRAVER, W M CARRIERE F19628-76-C-0180
UNCLASSIFIED CR-1-721-VOL-1 ESD-TR-77-262-VOL-1 NL

4 OF 4
AD
A053020

END
DATE
FILMED
5-78
DDC

Systems analysts are classified according to level of experience, expertise, and management skills as follows:

1. Senior Systems Analyst. Analyzes and evaluates potential new software applications, refines existing applications, formulates statements of problems and devises solutions, prepares general block diagrams, and supervises systems analysts in the development of assigned portions of the software design.

2. Systems Analyst A. Conducts a significant portion of the applications analysis and resulting system design, including development of system hardware and/or software specifications, data verification methods, etc. May also assist in supervising other systems analysts.

3. Systems Analyst B. Independently develops assigned portions of the system design, including block diagrams, flow charts, record layouts, and other portions of the system design documentation.

4. Systems Analyst C. Conducts analyses of a less complex nature and, in general, provides assistance on specific tasks as directed by other members of the systems analysis team.

C.4 SYSTEMS PROGRAMMER

Systems programmers require the skills to perform the following functions:

- Conceive, code, test, modify, and maintain the systems programs that carry out the internal functions of a digital computer.

- Develop extremely complex systems software such as operating systems, sophisticated file management routines, large telecommunications networks, advanced mathematical and scientific software packages, multiprogramming routines, compilers, link editors, and assemblers.
- Develop moderately complex systems software such as utilities, job control language, macros, subroutines.
- Install vendor-supplied utilities, application packages, and engineering releases.

Systems programmers are classified according to level of experience, expertise, and management skills as follows:

1. Senior Systems Programmer. Develops extremely complex systems software such as an entire operating system or complex subsystems. May also provide significant technical direction to lower-level personnel.
2. Systems Programmer A. Assists in the development and modification of extremely complex systems software as defined by higher-level personnel.
3. Systems Programmer B. Assists in the definition and programming of moderately complex systems software, as well as some portions of extremely complex systems software.
4. Systems Programmer C. Assists in coding and maintaining systems software of moderate complexity, program libraries, and technical manuals under the technical direction of higher-level personnel.

C.5 APPLICATIONS PROGRAMMER

Applications programmers require the skills to perform the following functions:

- Assist other members of the project team in analyzing and defining specific applications, methods of approach, specifications or requested modifications of software, and correction of software errors.
- Logically analyze a problem, prepare block diagrams and flow charts, and code programs, including the partitioning of software systems into individual modules.
- Check programs for logical sequence, perform test runs, compare test results against known solutions, determine causes of software malfunctions, and make corresponding corrections, as needed.
- Document the developed software in the form of program descriptions, block diagrams, flow charts, listings, test data, test results, etc.

Application programmers are classified according to level of experience, expertise, and management skills as follows:

1. Senior Applications Programmer. Provides assistance (as needed) in analyzing and evaluating software applications; assists and supervises programmers in the development and modification of applications software.
2. Applications Programmer A. Applies programming techniques and mathematical theories in the development and modification of applications software.
3. Applications Programmer B. Independently prepares short applications programs, or prepares portions of more complex applications programs in accordance with specifications.
4. Applications Programmer C. Assists in the preparation of short applications programs or portions of more complex applications programs, in accordance with specifications and specific instructions.

C.6 PROGRAM LIBRARIAN/SECRETARY

Program librarian/secretaries require the skills to perform the following functions:

- Maintain a program library which contains all source code, relocatable modules, linkage-editing statements, object modules, job control statements, test information, etc.
- Maintain current listings of the essential portions of the library as appropriate for each programmer.
- Update the library.
- Retrieve modules for compilation, run jobs, and store results, including those for test runs.
- Produce library status listings.

Program librarian/secretaries are classified according to level of experience, expertise, and management skills as follows:

1. Senior Program Librarian/Secretary. Directs and coordinates all activities associated with the program library, including the instruction of personnel regarding operating procedures and assignments.
2. Program Librarian. Secretary A. Maintains the program library and reports its status to the immediate supervisor.
3. Program Librarian/Secretary B. Maintains portions of the program library for particular programmers.
4. Program Librarian/Secretary C. Submits decks for compilation and/or subsequent execution, enters test data, forms the data base, etc.

C.7 DATA ENTRY OPERATOR

Data entry operators require the skills to perform the following functions:

- Operate data entry devices such as keypunch, key-to-tape, and key-to-disk.
- Verify entered data.
- Perform related clerical tasks.

Data entry operators are classified according to level of experience, expertise, and management skills as follows:

1. Senior Data Entry Operator. Supervises data entry operators, schedules their assignments and activities, instructs operators on procedures used to perform assignments, and trains new operators assigned to the project.
2. Data Entry Operator A. Operates data entry devices in recording a variety of data, instructs operators on procedures used to perform routine assignments, and assists in training new operators, with minimal supervision.
3. Data Entry Operator B. Operates date entry devices in recording a variety of data, verifies entered data, and performs related clerical duties, under direct supervision.
4. Data Entry Operator C. Performs data entry on a volume basis with only one or two input formats and a single type of data entry device, under direct supervision. May also occasionally assist in other related types of work.

TABLE C.1

1975 SALARY LEVELS FOR EACH SKILL CATEGORY

Skill Category	Hourly Rates				
	Low	1st Q	Avg	3rd Q	High
Management Personnel:[1]					
Project	$11.78		$16.87		$20.48
Technical	12.02		15.29		17.69
Configuration[2]	--		--		--
Scientists and Engineers:[3]					
Senior			$13.62		
Level A			12.84		
Level B			10.18		
Level C			6.60		
Systems Analysts:[4]					
Senior	$ 5.10	$ 8.20	$ 9.45	$10.55	$15.50
Level A	4.40	7.10	8.18	9.13	13.40
Level B	3.90	6.30	7.25	8.10	11.90
Level C	3.40	5.50	6.33	7.08	10.38
Systems Programmers:[4]					
Senior	$ 5.28	$ 7.88	$ 8.93	$ 9.85	$15.15
Level A	4.63	6.90	7.83	8.65	13.30
Level B	3.80	5.65	6.40	7.08	10.90
Level C	3.35	4.98	5.65	6.23	9.60
Applications Programmers:[4]					
Senior	$ 4.15	$ 6.78	$ 7.70	$ 8.45	$14.15
Level A	3.65	5.98	6.80	7.45	12.48
Level B	3.20	5.25	5.98	6.55	10.98
Level C	2.83	4.63	5.28	5.78	9.68
Program Librarians/Secretaries:[5]					
Senior			$ 5.94		
Level A			5.31		
Level B			4.96		
Level C			4.56		

TABLE C.1 (Continued)

Skil Category	Hourly Rates Low	1st Q	Avg	3rd Q	High
Data Entry Operators:[4]					
Senior	$2.65	$3.80	$4.43	$4.90	$8.60
Level A	2.35	3.35	3.93	4.35	7.60
Level B	2.10	3.00	3.53	3.90	6.83
Level C	1.85	2.68	3.10	3.45	6.05

[1]Figures taken from private nationwide survey of R&D companies for 1975.

[2]Information for this skill category is currently not available.

[3]Figures taken from Battelle's nationwide survey of aerospace scientists and engineers in R&D for 1975.

[4]Figures taken from Hansen's Weber nationwide survey of data processing personnel for 1975.

[5]Figures taken from private, southern-California survey of computer science secretaries for 1975.

REFERENCES

1. T. J. Devenny, An Exploratory Study of Software Cost Estimating at The Electronics Systems Division, USAF Air University, Air Force Institute of Technology, Wright-Patterson Air Force Base, Ohio, July 1976.

2. G. F. Weinwurm, Data Elements for a Cost Reporting System for Computer Program Development, System Development Corporation, AD 637 804, ESD-TR-66-411, August 1966.

3. E. A. Nelson, T. Fleishman, A System for Collecting and Reporting Costs in Computer Program Development, System Development Corporation, TM-3411/000/00, 11 April 1967.

4. L. Farr, H. Zagorski, Factors that Affect The Cost of Computer Programming; A Quantitative Analysis, System Development Corporation, TM-1447/001/00, August 1964.

5. G. F. Weinwrum, H. Zagorski, Research Into The Management of Computer Programming; A Transitional Analysis of Cost Estimation Techniques, ESD-TR-65-575, November 1965.

6. E. A. Nelson, Management Handbook for The Estimation of Computer Programming Costs, System Development Corporation, RM-3225/000/01, 20 March 1967.

7. R. L. Smith, Structured Programming Series, Vol. IX, "Management Data Collection and Reporting," IBM Federal Systems Center, AD A008 640, 24 October 1974.

8. J. A. Clapp, A Review of Software Cost Estimation Methods, The MITRE Corporation, ESD TR 76-271, August 1976.

9. L. H. Morin, Estimation of Resources for Computer Programming Projects, University of North Carolina, 1974.

10. B. C. Frederic, A Provisional Model for Estimating Computer Program Development Costs, Tecolote Research, Inc., TM-7/Rev. 1, December 1974.

11. Acquisition and Support Procedures for Computer Resources in Systems, AFR 800-14, 26 September 1975.

12. Automatic Data Processing Resource Estimating Procedures (ADPREP), Planning Research Corporation, PRC R-1527, August 1970.

13. J. S. Gansler, "Software Management in the Department of Defense," Proceedings of the Software Management Conference, Anahiem, California, April 1975.

14. L. Farr, B. Nanus, Factors That Affect The Cost of Computer Programs, System Development Corporation, Technical Memorandum TM-1447/000/02, 1964.

15. V. LaBolle, Estimation of Computer Program Costs, System Development Corporation, Professional Paper, SP-1747, 1964.

16. B. Nanus, L. Farr, "Some Cost Contributors to Large-Scale Programs," AFIPS Conference Proceedings, Spring Joint Computer Conference, Vol. 25, 1964, pp. 239-248.

17. V. LaBolle, Development of Equations for Estimating The Costs of Computer Program Production, System Development Corporation, Technical Memorandum TM-2918/000/00, 1966.

18. W.L. Schoeffel, An Air Force Guide to Software Documentation Requirements, MITRE Corporation, ESD TR 76-159, June 1976.

19. R. W. Wolverton, "The Cost of Developing Large-Scale Software," IEEE Transactions on Computers, Vol. C-23, No. 6, June 1974, pp. 615-636.

20. F.P. Brooks, Jr., The Mythical Man-Month, Essays on Software Engineering, Addison-Wesley, Reading, Massachusetts, 1974.

21. A. Williman, C. O'Donnell, "Through the Central Multi-Processor Avionics Enters the Computer Era," Astronautics & Aeronautics, July 1970, pp. 44-52.

22. E. N. Dodson, et al., Advanced Cost Estimating and Synthesis Techniques for Avionics, General Research Corporation, Final Report, CR-2-461, 1975.

23. F. P. Brooks,"Why is Software Late," Data Management, August 1971.

24. C. A. Graver, Historical Simulation: A Procedure for The Evaluation of Estimating Procedures, Vol. I, "Procedure Development and Description," General Research Corporation, CR-0364-1, June 1969.

25. J. W. Franc, "The BMD and BMDP Series of Statistical Computer Programs," Comm. of the ACM, Vol. 19, No. 10, October 1976, pp. 570-576.

26. L. A. Belody, M. M. Lehman, "A Model of Large Program Development," IBM Systems Journal, Vol. 15, No. 3, 1976, pp. 225-252.

27. A. M. Pietrasanta, Resource Analysis of Computer Program System Development, On The Management of Computer Programming, G. Weinwurm Ed., Auerbach Publishers, Inc., Princeton, N.J., 1970, pp. 67-87.

28. J. D. Aron, Estimating Resources for Large Programming Systems, International Business Machines Corporation, Federal Systems Center, Gaithersburg, Maryland, 1969.

29. Proprietary Source, a major software development house.

30. B. Nanus, L. Farr, "Some Cost Contributors to Large Scale Programs," Proceedings, Spring Joint Computer Conference, 1964, pp. 239-248.

31. B. W. Boehm, "Software and Its Impact: A Quantitative Assessment," Datamation, May 1973, pp. 48-59.

32. D. W. Kosy, Air Force Command and Control Information Processing in the 1980s: Trends in Software Technology, RAND Report R-1012-PR, June 1974.

33. Proceedings of a Symposium on the High Cost of Software, Naval Post Graduate School, 17-19 September 1973.

34. W. Hahn, J. Stone, Jr., Software Transfer Cost Estimation Technique, The MITRE Corporation, M70-43, July 1970.

35. Government/Industry Software Costing and Sizing Workshop, USAF, ESD-TR-76-166, June 1975.

36. G. F. Meyers, Estimating the Costs of a Programming System Development Project, International Business Machines Corporation, TR 00.2316, 23 May 1972.

37. L. Putnam, ADP Resource Estimating: A Macro-Estimating Methodology for Software Development, U.S. Army Computer Systems Command.

38. W. Delaney, "Predicting the Costs of Computer Programs," Data Processing Magazine, October 1966.

39. J. Engelland, Operational Software Concept (Phase 1), 5 volumes, June 1974, GENERAL DYNAMICS--FORT WORTH; AFAL-TR-74-168

40. W.L. Trainor, Software - From SATAN to SAVIOUR, US Air Force Avionics Laboratory.

41. J.D. Glore, Software Acquisition Management Guidebook: Statement of Work Preparation, MITRE Corporation, ESD TR 77-16, January 1977.

42. Cost Analysis: Program Breakdown Structure and Codes, HQ Air Force Systems Command, Andrews Air Force Base, AFSC Manual 173-4, 24 November 1972.

43. J.B. Glore, Software Management Acquisition Guidebook: Life Cycle Events, MTR-3355, January 1977; ESD-TR-77-22, FEB 77; AD-A037115

44. E.N. Dodson, et al., Phase I Summary Advanced Cost Estimating and Synthesis Techniques for Avionics, General Research Corporation, Final Report, CR-1-461, 1974.

45. P. Gehring, A Quantitative Analysis of Estimating Accuracy in Software Development, Texas A&M, August 1976.