SOFTWARE ENGINEERING LABORATORY SERIES

SEL-83-002

MEASURES AND METRICS FOR SOFTWARE DEVELOPMENT

MARCH 1984

NASA
National Aeronautics and
Space Administration

Goddard Space Flight Center
Greenbelt, Maryland 20771

SOFTWARE ENGINEERING LABORATORY SERIES SEL-83-002

MEASURES AND METRICS FOR SOFTWARE DEVELOPMENT

MARCH 1984

NASA
National Aeronautics and
Space Administration

Goddard Space Flight Center
Greenbelt, Maryland 20771

FOREWORD

The Software Engineering Laboratory (SEL) is an organization sponsored by the National Aeronautics and Space Administration/Goddard Space Flight Center (NASA/GSFC) and created for the purpose of investigating the effectiveness of software engineering technologies when applied to the development of applications software. The SEL was created in 1977 and has three primary organizational members:

NASA/GSFC (Systems Development and Analysis Branch)
The University of Maryland (Computer Sciences Department)
Computer Sciences Corporation (Flight Systems Operation)

The goals of the SEL are (1) to understand the software development process in the GSFC environment; (2) to measure the effect of various methodologies, tools, and models on this process; and (3) to identify and then to apply successful development practices. The activities, findings, and recommendations of the SEL are recorded in the Software Engineering Laboratory Series, a continuing series of reports that includes this document. A version of this document was also issued as Computer Sciences Corporation document CSC/TM-83/6061.

The contributors to this document include

David Card	(Computer Sciences Corporation)
Frank McGarry	(Goddard Space Flight Center)
Jerry Page	(Computer Sciences Corporation)
Victor Church	(Computer Sciences Corporation)
Leon Jordan	(Computer Sciences Corporation)

Single copies of this document can be obtained by writing to

Frank E. McGarry
Code 582
NASA/GSFC
Greenbelt, Md. 20771

ABSTRACT

This document reports the evaluations of and recommendations for the use of software development measures based on the practical and analytical experience of the Software Engineering Laboratory. It describes the basic concepts of measurement and a system of classification for measures. The principal classes of measures defined are explicit, analytic, and subjective. Some of the major software measurement schemes appearing in the literature are reviewed. The applications of specific measures in a production environment are explained. These applications include prediction and planning, review and assessment, and evaluation and selection.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

Figure

LIST OF TABLES

Table

SECTION 1 - INTRODUCTION

Effective software development management depends on the accurate measurement of project attributes. This document reviews the state of the art of practical software measurement, which is still, to some extent, an art rather than a science. Substantial research is in progress, however, and innovations have been rapid in this vital area. Major improvements in both the collection and interpretation of measures are expected in the next few years. However, certain lessons can be applied now.

Many different measures have been proposed in the literature (Reference 1). (No distinction is made in this document between the meaning of "measure" and that of "metric.") During the past 6 years, the Software Engineering Laboratory (SEL) has made a major effort to understand, verify, and apply these measures to the software development process, as well as to develop new ones and refine existing ones. This document presents some evaluations of and recommendations for the application of software development measures and metrics, based on the practical and analytical experience of the SEL.

Measures appeal to the software engineering researcher and software development manager as potential means of defining, explaining, and predicting software quality characteristics, especially productivity, reliability, and maintainability. The software manager in particular needs to be able to determine the quality of a software project at every point in its life cycle. Questions that measures can answer include the following:

- Is this software project on schedule?
- How many errors can be expected?

- Is this methodology effective?
- How good is this product?

The reader will obtain an understanding of the theory of software measures and their application to questions such as these. This document is intended to serve as a reference for the technical manager of software development projects who desires to monitor and review ongoing development, predict cost and quality, and evaluate alternative development techniques. Another document (Reference 2) discusses the difficulties and priorities of collecting measures and data in general.

This document presents the general concepts of software measurement, reviews the work done to date, and then demonstrates the application of these concepts in a production environment. Its scope will be expanded as the SEL learns more about measures. In particular, a major effort is under way to identify measures that can be applied early in the development process (i.e., during requirements and design). The results of these studies will move us closer to the final goal of putting the academics of measures into the hands of the software practitioner.

1.1 DOCUMENT ORGANIZATION

This document consists of four major sections. Section 1 introduces some concepts of software measurement and describes the source of the analyzed data and the basis of the practical experience. References 2 and 3 present more detailed explanations of this material.

Section 2 explains a classification scheme for software measures. Organizing the available measures in this manner facilitates their systematic consideration. Some commonly used software measures are explained within the context of

the classification scheme. The following classes of measures are defined by this scheme:

- Explicit
- Analytic
- Subjective

Section 3 summarizes the efforts of the SEL to evaluate the available software measures. Studies of each of the three classes of measures are described.

Section 4 demonstrates the application of measures to software development management for each phase of the software life cycle. The following applications of measures are considered:

- Prediction (for planning)
- Review (for assessment)
- Evaluation (for selection)

Section 5 reiterates the major conclusions and indicates the direction of current SEL research.

1.2 CONCEPTS OF MEASUREMENT

Measurement is the process of assigning a number or state to represent a physical quantity or quality. The need to measure the quantity and quality of developed software is self-evident. Measures of productivity, reliability, maintainability, and complexity, for example, are vital to software development planning and management.

A large number of measures have been proposed by researchers, not all of which are equally useful in practice. This document is an attempt to organize the available measures in a rational manner and to identify those that have been employed successfully in a production environment.

Reference 2 explains a three-dimensional scheme for the classification of software measures. These dimensions are development component, level of detail, and measurement method. The first two are useful in planning and implementing a program of data collection. The last is essential to the interpretation and application of software measures, which is the focus of this document.

The first dimension of classification is the development component. The software development activity can be divided into discrete components, as shown in Figure 1-1. The components included in this model are the following:

- Problem--The software problem as described in the requirements specification and constraints
- Environment--Characteristics of the development installation and personnel
- Product--The software and documentation produced by the development effort
- Process--The procedures, techniques, and methodologies employed in developing the product

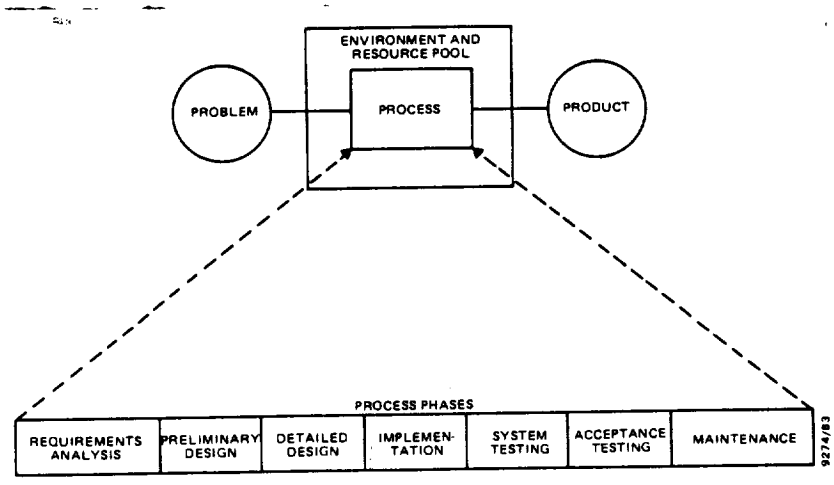

Figure 1-1. Software Development Model

Measurements can be classified based on the components (i.e., problem, environment, product, process) to which they apply. Examples of problem measures include the number and complexity of requirements. Programming language and development computer are characteristics of the environment. Product measures include lines of code and pages of documentation. Team size and methodology use are examples of measures that characterize the process.

The software development process is the component most easily manipulated by managers and must be carefully monitored. Simply measuring a software development project at its conclusion is inadequate for most purposes. Measurements must be made throughout the life of a software project. Figure 1-1 shows the decomposition of the process into seven life cycle phases. Although this is the basic life cycle definition used by the SEL, a simpler sequence consisting of design, implementation, and testing is also used in this document as a heuristic device. Table 1-1 compares the two life cycle definitions.

Table 1-1. Software Life Cycle Definitions (Based on SEL Experience)

Detailed Life Cycle		Simplified Life Cycle	
Phase	Percent of Schedule	Phase	Percent of Schedule
Requirements Analysis	5		
Preliminary Design	10	Design	30
Detailed Design	15		
Implementation	40	Implementation	40
System Testing	20		
Acceptance Testing	10	Testing	30

Another dimension of classification is level of detail (or resolution). Measurements can be performed at several levels of detail:

- Project--High-level summary
- Component--Discrete project parts, such as subsystems, modules, COMMON blocks, etc.
- Event--Occasional or periodic occurrences, such as changes, computer runs, etc.

The level of detail of measures collected depends on the manager's perspective and cost constraints. As discussed in Section 4, these measures can provide useful feedback to managers and developers. (Reference 2 presents a more detailed discussion of cost considerations.)

The final and most important dimension (from the perspective of this document) is method of measurement. Measurements can be obtained by several different methods:

- Explicit--Simple numeric counts, averages, and other directly obtained indicators (e.g., lines of code and errors per module)
- Analytic--Complex measures based on assumptions about the relationships among software features (e.g., Halstead length and cyclomatic complexity)
- Subjective--Ratings of quality and use arrived at by high-level review and comparison (e.g., traceability and completeness)

The specific measurement method employed implies certain beliefs about the nature of the software development process. This dimension of classification (method of measurement) is the basis for the organization of Sections 2 and 3

of this document. Section 2 reviews some of the major measurement proposals of each type. Section 3 summarizes the results of SEL research in each of these areas.

1.3 SOFTWARE ENGINEERING LABORATORY

The SEL is a cooperative effort of Goddard Space Flight Center (GSFC), Computer Sciences Corporation (CSC), and the University of Maryland (UM). The SEL collects and analyzes data from software development projects that support flight dynamics activities at GSFC. More than 40 projects have been monitored by the SEL during its 7-year life. SEL principals also participate in the management of these projects. The recommendations presented in this document are based on this analytical and practical experience. Reference 3 describes the SEL and its activities in more detail.

The general class of spacecraft flight dynamics software studied by the SEL includes applications to support attitude determination, attitude control, maneuver planning, orbit adjustment, and mission analysis. The attitude systems, in particular, form a large and homogeneous group of software that has been studied extensively. Table 1-2 summarizes the major characteristics of the software developed in this environment.

Measures have been collected and analyzed regularly from these projects. The bibliography included in this document contains numerous reports of the results of these analyses. Reference 4 describes a recent study incorporating more than 600 measures of special interest to software development managers.

Table 1-2. Software Development Environment

Type of Software: Scientific, ground-based, interactive graphic with moderate reliability and response requirements

Languages: 85 percent FORTRAN, 15 percent assembler macros

Machines: IBM S/360 and 4341, batch with TSO

Process Characteristics	Average	High	Low
Duration (months)	15.6	20.5	12.9
Effort (staff-years)	8.0	11.5	2.4
Size (1000 LOC)			
Developed[a]	57.0	111.3	21.5
Delivered[b]	62.0	112.0	32.8
Staff (full-time equivalent)			
Average	5.4	6.0	1.9
Peak	10.0	13.9	3.8
Individuals	14	17	7
Application Experience (years)			
Managers	5.8	6.5	5.0
Technical Staff	4.0	5.0	2.9
Overall Experience (years)			
Managers	10.0	14.0	8.4
Technical Staff	8.5	11.0	7.0

[a]New lines of code plus 20 percent of reused lines of code.

[b]Total lines of code.

SECTION 2 - SURVEY OF MEASUREMENT APPROACHES

A classification of the available software development measures is a prerequisite for any systematic evaluation of them. Many of the measures that have been proposed are similar to each other. A classification scheme provides a mechanism for avoiding unnecessary duplication while ensuring full coverage of all important software development characteristics. The following classes of measures will be discussed here:

- Explicit
- Analytic
- Subjective

The following sections define these classes, show their logical relationship to each other, and outline some of the major measurement proposals. The reader can consult the references for more detailed explanations.

2.1 EXPLICIT MEASURES

The class of explicit measures contains the easiest to understand and most widely used measures. This class includes counts and ratios directly determined from source code, staffing records, computer usage logs, and documentation. Values of these measures are fixed and unambiguous for a given project or component, although there is some variability in nomenclature. (Reference 5 provides an extensive set of definitions for these measures as well as other elements of software engineering.) The following are the most important explicit measures:

- Developed lines of code--All newly developed lines of code plus a fraction of reused lines of code; a measure of size

- Lines of code per staff hour--Lines of code developed for each staff hour expended; a measure of productivity
- Errors per thousand lines of code--Errors detected for every thousand lines of code developed; a measure of reliability

These are the most widely used measures of software size and "quality," probably because not enough is yet understood about other measures. The exact hours, lines, and errors counted must be defined locally. Table 2-1 lists some other measures typical of this class. Walston and Felix (Reference 6) studied the relationship of many such measures to productivity and reliability.

Table 2-1. Typical Explicit Measures

Component	Measure
Problem	Number of requirements Number of interfaces Number of functions
Environment	Programming language Development machine Programmer experience
Product	Lines of code Number of modules Pages of documentation
Process	Staff level Development time Methodology use

Although explicit measures are easily determined, they have several limitations: they are usually available only after the software development activity is complete; their scope is limited to the areas of size, productivity, and reliability; they have little explanatory power; and they are not sensitive to the specific objectives of a software

development project. The next two subsections discuss some alternatives and complements to explicit measures that attempt to counter these weaknesses.

2.2 ANALYTIC MEASURES

Analytic measures are based on some assumption or hypothesis about the nature of software and the software development process. They are intended to be sensitive to defined "critical" properties. Examples include cyclomatic complexity, program length, and reference span (see following sections). The value of these measures depends on the validity of the underlying assumption or hypothesis. Validation of these hypotheses is an active area of software engineering research. Analytic measures generally deal with one of three basic software properties (Reference 7): program size, control structure, or data structure. Each of these properties has been studied with several different conceptual approaches. Although researchers frequently disagree on the importance of each property and the calculation of specific measures of them, some analytic approaches to measures have become well established. These approaches are reviewed in the following sections.

2.2.1 PROGRAM SIZE

One of the most comprehensive theories and sets of measures for software development was proposed by Halstead (Reference 8). This "software science" is a set of relationships between the size of a program and other software qualities. The essential premise of software science is that any programming task consists of selecting and arranging a finite number of program components (operators and operands). The number of these components then determines the implementation effort required and the number of errors produced. An operator is a symbol denoting an operation, function, or action. An operand is a symbol representing a data item or

target of the action of an operator. The following basic measures are defined by Halstead:

- Number of unique operators (n_1), e.g., +, -, IF
- Number of unique operands (n_2), e.g., X, Y, I, 200
- Total number of appearances of operators (N_1)
- Total number of appearances of operands (N_2)

Figure 2-1 shows the source listing of a simple FORTRAN program. Its component operators and operands are identified in Figure 2-2. (This identification was done by the Source Analyzer Program described in Reference 9.) The values of the basic Halstead measures for the sample program are $n_1 = 16$, $n_2 = 21$, $N_1 = 59$, and $N_2 = 50$. These measures can be combined to calculate some important software properties, as shown in Table 2-2.

Table 2-2. Software Science Relationships

Quality	Equation
Vocabulary (n)	$n = n_1 + n_2$
Length (N)	$N = N_1 + N_2$
Volume (V)	$V = N \log_2 n$
Level (L)	$L = V/V^*$
Effort (E)	$E = V/L$
Faults (B)	$B = V/S^*$

NOTES: V* is the minimum volume represented by a built-in function performing the task of the entire program.

S* is the mean number of mental discriminations (decisions) between errors ($S^* \approx 3000$).

Ostensibly, software science provides equations for estimating the cost (effort) and reliability (faults) of developed software (see Table 2-2). These equations are based on assumptions about the mental process of programming. Although

```
 100       SUBROUTINE TDIST (N, X, Y, DIST)
 200C                                                PASSED
 300       INTEGER    N
 400       REAL       X(N), Y(N), DIST
 500C                                                LOCAL
 600       INTEGER    I, MSGNUM, K
 700       REAL       XL, YL, DX, DY, X2, Y2, R2, R
 800       LOGICAL    ERR
 900C                                                GLOBAL
1000       REAL       SQRT
1100C                                               INITIALIZE
1200       XL = 0.0
1300       YL = 0.0
1400       DIST = 0.0
1500C                                               FOR ALL POINTS
1600       DO 200 I=1, N
1700         DX = X(I) - XL
1800         X2 = DX**DX
1900         DY = Y(I) - YL
2000         Y2 = DY**DY
2100C                                               CALC./CHECK SEPARATION
2200         R2 = X2 + Y2
2300         CALL VERIFY (R2, ERR)
2400C                                               OBTAIN SEPARATION
2500         IF (ERR) THEN
2600           K = I - 1
2700           WRITE (6, 100, ERR=300) K, I
2800  100      FORMAT (1X, 'ERROR, POINTS ', I3, ' AND ', I3,
2900     *             ' TOO CLOSE')
3000           R = 0.0
3100         ELSE
3200           R = SQRT (R2)
3300         END IF
3400C                                               ACCUMULATE
3500         DIST = DIST + R
3600         XL = X(I)
3700         YL = Y(I)
3800  200 CONTINUE
3900C                                               NORMAL RETURN
4000       RETURN
4100C                                               ERROR WRITING MESSAGE
4200  300 CONTINUE
4300       MSGNUM = 27
4400       CALL ERRMSG (MSGNUM, *400)
4500       RETURN
4600C                                               UNABLE TO WRITE ANY
4700C                                               MESSAGES, ABORT RUN
4800  400 CONTINUE
4900       CALL ABORT
5000C
5100       END
```

Figure 2-1. Sample FORTRAN Program

```
11:12:46 19-AUG-82 SOURCE ANALYZER PROGRAM V2   MODULE STATISTICS SUMMARY FILE        LUN  7   TDIST

SAMPLE.FOR/HL/DB/SL/XP

                                HALSTEAD OPERATORS

DELIMETERS
          0  //       0  **      3  *       0  /       2  +       3  =
         15  =        5  (       2  ,       0  &       0  .NE.    0  .LT.
          0  .LE.     0  .EQ.    0  .GE.    0  .GT.    0  .AND.   0  .OR.
          0  .XOR.    0  .EQV.   0  .NOT.   0  .NEQV.

KEYWORDS
          0  IF()          0  IF(),,       1  .IF()    0  ELSE IF     1  ELSE          1  DO=,,
          0  DOWHILE       0  ASSIGNTO    20  EOS

PROCEDURES
          1  VERIFY        1  SORT         1  ERRMSG   1  ABORT

TRANSFERS
          1  ERR=       300
          1  ALT.RET.   ERRMSG 400

                               HALSTEAD OPERANDS

          2 1          6 I          1 K          1 N          3 B          2 X          2 Y          1 27
          3 R2         2 X2         2 Y2         4 0.0        1 300        1 400        3 DX         3 DY
          3 XL         3 YL         2 ERR        3 DIST       4 MSGNUM
```

Figure 2-2. Tabulation of Operators and Operands

some early studies supported the validity of software science (Reference 10), recent work has challenged this theory on both empirical (Reference 11) and theoretical (Reference 12) grounds.

2.2.2 CONTROL STRUCTURE

Another well-developed concept of measurement, cyclomatic complexity, was introduced by McCabe (Reference 13) in an attempt to quantify control flow complexity. The original objectives of the measure were to determine the number of paths through a program that must be tested to ensure complete coverage and to rate the difficulty of understanding a program. However, many researchers have attempted to relate cyclomatic complexity directly to software reliability.

The basic measure is the cyclomatic number derived from the graphic representation of a program's control flow. Figure 2-3 is an example of a graphic representation of the control flow of the program shown in Figure 2-1. The cyclomatic number of the program represented in the figure is equal to the number of disjoint regions defined by the edges of the graph, or the number of binary decisions plus one. The following is a more general formula:

$$V(G) = e - n + 2p$$

where $V(G)$ = cyclomatic number of graph G
e = number of edges
n = number of nodes
p = number of unconnected parts

The cyclomatic number of the graph shown in Figure 2-3 is 4 ($4 = 10 - 8 + 2 \times 1$). McCabe suggested that any program (or module) with a cyclomatic number greater than 10 is too complex. Another measurement scheme similarly based on counts of decisions was proposed by Gilb (Reference 14).

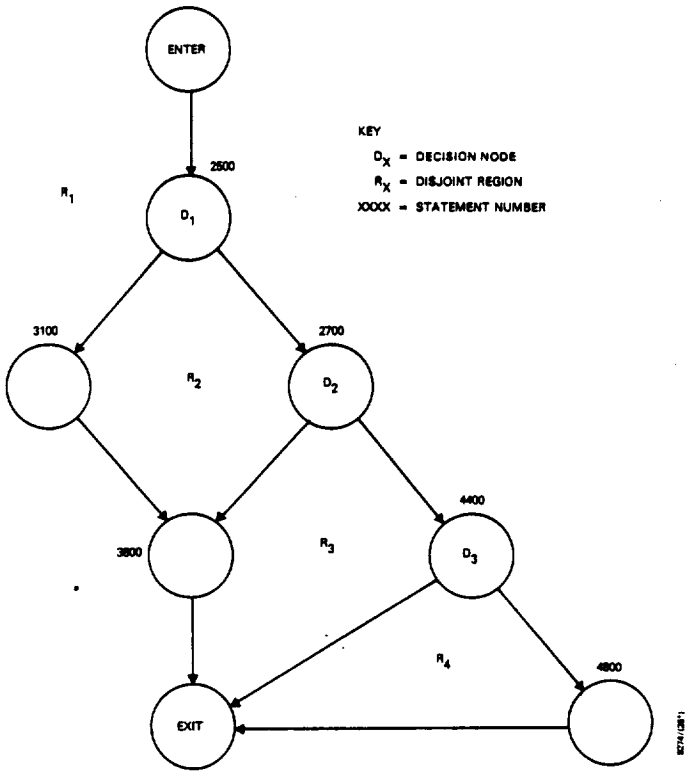

Figure 2-3. Graphic Representation of Cyclomatic Complexity (of Program in Figure 2-1)

McCabe's theory was extended by Myers (Reference 15) to include decisions based on compound conditions. Although early research (Reference 16) and theoretical consideration (Reference 17) gave favorable indications, more recent study has not supported the value of cyclomatic complexity as an indicator of development effort or reliability. Evangelist (Reference 18) suggested that the measure could be reformulated to more accurately reflect control flow. Hansen (Reference 19) has proposed a promising measure incorporating both cyclomatic complexity and software science measures that has yet to be evaluated.

2.2.3 DATA STRUCTURE

The reference span approach to measurement is based on the location of data references within a program. The span of

reference of a data item is the number of statements between successive references to that data item (Figure 2-4). For example, the reference span of variable DIST in the sample program (Figure 2-1) is 16. Elshoff (Reference 20) has shown that the reference span measure varies widely. He also noted its implications for program complexity and readability. Long distances (reference spans) between occurrences of a variable make a program difficult to understand and maintain. According to this theory, minimizing the length of reference spans minimizes program complexity.

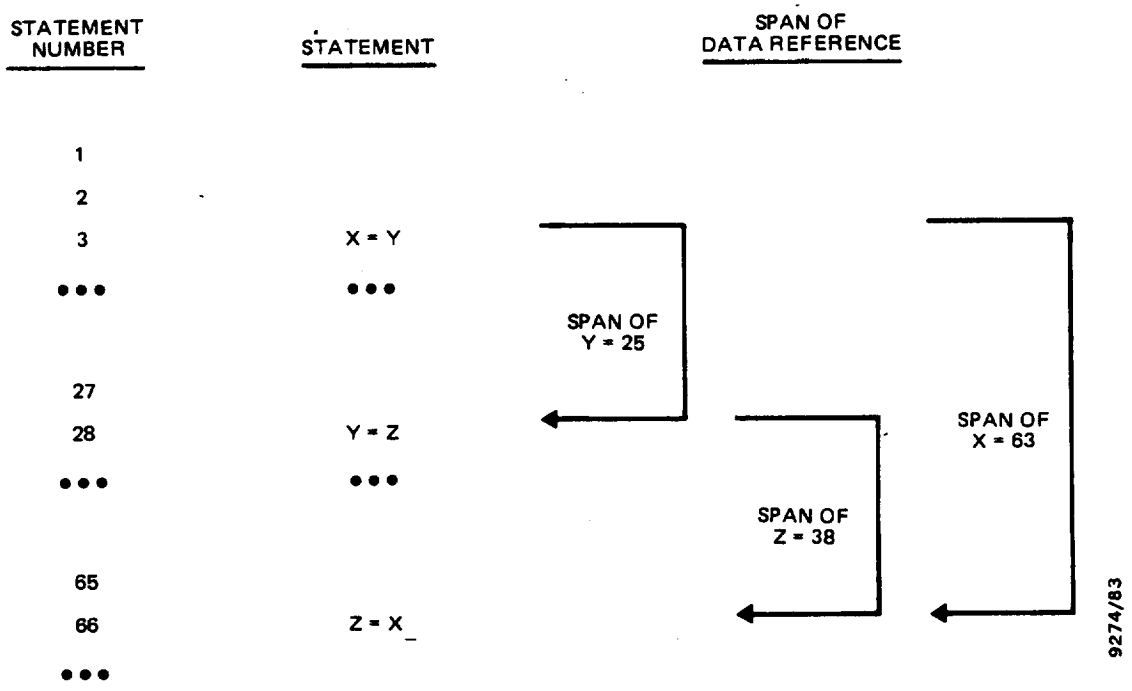

Figure 2-4. Graphic Representation of Reference Span

Other approaches to measuring data structures and data flow have been developed recently (Reference 21). They attempt to consider the effects of how data are used and structured within a program as well as their volume. The relevant measures are, however, frequently difficult to compute, although useful simplifications can be made. For example,

Henry and Kafura (Reference 22) have developed some relatively straightforward measures based on concepts of information flow and system connectivity.

2.3 SUBJECTIVE MEASURES

Subjective measures are so called because they are relative ratings of quality rather than absolute measurements. The explicit and analytic measures just discussed are absolute measures of software properties. Absolute measures are deficient in that their scope is limited to tangible quantities. Consequently, they are not sensitive to the specific quality objectives of a software development project. In contrast, subjective measures are often used to compare the actual realization of a project with its ideal or target qualities.

The greater scope of subjective measures relative to the measures discussed in Sections 2.1 and 2.2 is demonstrated in Table 2-3. The subjective measures identified in the table were proposed by McCall (Reference 23). Most of these measures have no explicit or analytic counterparts. These measures are intended to be used to evaluate the performance of a software development project relative to specified quality targets. The McCall scheme is based on combining independent evaluations of multiple criteria to produce a value for each measure (or factor, as they are referred to by McCall).

Although McCall's is the best-known measurement scheme of this type, comparable schemes have been proposed by Gilb (Reference 14) and the SEL (Reference 4). The McCall measures have been extended for use early in the software life cycle (Reference 24), during maintenance (Reference 25), and with distributed systems (Reference 26). Values of subjective measures are, however, difficult to determine

Table 2-3. Definition of Software Quality Factors[a]

Factor	Definition
Correctness	Extent to which a program satisfies its specifications and fulfills the user's mission objectives
Reliability	Extent to which a program can be expected to perform its intended function with required precision
Efficiency	Amount of computing resources and code required by a program to perform a function
Integrity	Extent to which access to software or data by unauthorized persons can be controlled
Usability	Effort required to learn, operate, prepare input, and interpret output of a program
Maintainability	Effort required to locate and fix an error in an operational program
Testability	Effort required to test a program to ensure that it performs its intended function
Flexibility	Effort required to modify an operational program
Portability	Effort required to transfer a program from one hardware configuration and/or software system environment to another
Reusability	Extent to which a program can be used in other applications; related to the packaging and scope of the functions that programs perform
Interoperability	Effort required to couple one system with another

[a]From Reference 23, Table 3.1-1.

consistently. Thus projects, especially those from different environments, cannot easily be compared.

Another subjective measurement scheme and corresponding development methodology proposed by Myers (Reference 27) incorporates measures of both data and control structure. This theory is based on the concept of program modularity. Levels of "module strength" and "module coupling" are defined that correspond to degrees of control cohesion and data independence. Table 2-4 explains the levels of module strength and coupling.

Module strength is a measure of "singleness of purpose." A module performing only a single function has the greatest strength. A module performing several unrelated functions has low strength. Module coupling is a measure of "dependence" between modules. Two modules linked only through data passed in a calling sequence have the weakest coupling. The use of control flags and COMMON blocks, for example, increases the level of coupling. Myers suggested that software quality could be improved by maximizing module strength and minimizing module coupling. The determination of the actual strength and coupling of a module is, however, subjective, although some progress has recently been made in quantifying these concepts (Reference 28).

Table 2-4. Levels of Module Strength and Coupling

Level[a]		Description
Strength[b]	Functional	Single specific function
	Informational	Independent functions on common data structure
	Communicational	Multiple sequential functions related by data
	Procedural	Multiple sequential functions related by problem
	Classical	Multiple sequential functions
	Logical	Set of related functions
	Coincidental	No clearly defined function
Coupling[c]	Data	Share simple data items
	Stamp	Share common (local) data structure
	Control	Control elements passed
	External	Reference to global data item
	Common	Reference to global data structure (COMMON)
	Content	Direct reference to contents of other module

[a]Ordered from best to worst.

[b]Within a module.

[c]Between modules.

SECTION 3 - SUMMARY OF SEL RESEARCH

Although extensive research has been done in the area of software measures, much of it has been inconclusive. Many studies have had serious methodological flaws, and some important ideas have not been tested at all (Reference 29). Thus far, no measures have emerged that are clearly superior to lines of code, hours of effort, and errors detected. No other measures are widely used because none have been demonstrated to be effective in a production environment. This is partly due to the lack of relevant data. For example, there is no a priori reason to assume that a cyclomatic complexity of 10 is intrinsically superior to a cyclomatic complexity of 12. However, few software engineering data bases contain the detailed product information necessary to determine whether or not this is true.

Any evaluation of a measure must weigh the information it provides about productivity, reliability, and/or maintainability against its cost of collection. The SEL is conducting a continuing program of evaluating and refining existing measures and developing new ones. Reference 30 summarizes the results of SEL activities in this area. This section highlights some of the major findings about each of the classes of measures defined in the previous section.

3.1 EXPLICIT MEASURES

Early SEL experiments (Reference 31) with explicit measures attempted to verify the work of Walston and Felix (Reference 6). Although similar results were obtained, some important differences were noted. Table 3-1 compares the Walston-Felix data with SEL data. Differences between the data bases reflect the differences between the environments studied. Both data bases, however, showed consistent

Table 3-1. Comparison of Walston-Felix Data With SEL Data[a]

Measures	Walston-Felix Median	SEL Median
Total Source Lines (thousands)	20	49[b]
Percent of Lines Not Delivered	5	0
Source Lines per Staff-Month	274	601[b]
Documentation (pages) per Thousand Lines	69	26
Total Effort (staff-months)	67	96
Average Staffing Level	6	5
Duration (months)	11	15
Distribution of Effort		
Manager	22[c]	19
Programmer	73[c]	68
Other	5[c]	13
Errors per Thousand Lines	1.4	0.8

[a]From Reference 6, Table A-9.

[b]Lines are developed lines of code.

[c]Rescaled to sum to 100 percent

relationships among lines of code, pages of documentation, project duration, staff size, and programmer hours.

Table 3-2 shows the numerical relationships among these measures identified by the SEL and Walston-Felix. It should be noted that the coefficients and exponents in each pair of equations are of the same magnitude. This close agreement obtained from the analysis of two very different sets of data suggests that these relationships among explicit measures do indeed reflect basic properties of the software development process.

Table 3-2. Software Relationship Equations[a]

Measure	Software Engineering Laboratory		Walston-Felix	
	Equation[b]	CD[c]	Equation[b]	CD[c]
Effort (E) (staff-months)	$E = 1.4L^{0.93}$	0.93	$E = 5.2L^{0.91}$	0.64
Documentation (D) (pages)	$D = 30L^{0.90}$	0.92	$D = 49L^{1.01}$	0.62
Duration (T) (months)	$T = 4.6L^{0.26}$	0.55	$T = 4.1L^{0.36}$	0.41
Staff size (S) (average persons)	$S = 0.24E^{0.73}$	0.89	$S = 0.54E^{0.60}$	0.79

[a]From Reference 31, Table 1.

[b]In following equations L is total lines of code.

[c]Coefficient of determination (or r^2).

The SEL has achieved some success in applying explicit measures to cost estimation. Analysis of the relationships among productivity, lines of code, and other cost factors provided the empirical basis of the SEL Meta-Model for software cost (Reference 32). One of the liabilities of a model based on lines of code is that this quantity is known

accurately only after software development is complete. Reasonable early estimates can, however, be made based on other explicit measures, such as the number of subsystems or modules (Reference 33). In summary, the SEL has found explicit measures to be very effective for some software development characteristics.

3.2 ANALYTIC MEASURES

SEL studies of analytic measures have focused on program size and control structure. Measures of data structure are still in the early stages of investigation. The Halstead and McCabe measures (see Section 2.2) have been carefully examined by the SEL (References 34 and 35). Table 3-3 summarizes the relationship between several measures and productivity and reliability. As shown in the table, neither cyclomatic complexity nor Halstead effort was the best predictor of either productivity or reliability.

A recent SEL study (Reference 36) showed that higher correlations for cyclomatic complexity and Halstead effort could be obtained after carefully screening the data to ensure sample consistency. This suggests that the minute level of detail of these measures (operators, decisions, etc.) makes them sensitive to extraneous variations in the data collection process and programming style. They are thus unsuitable for use in production environments where extensive data verification is not possible.

Another approach to validating (Halstead) software science measures has been to show that they are internally consistent (Reference 34). For example, good agreement between the program length as predicted by software science and the actual program length has been taken as evidence of the validity of software science. Table 3-4 gives results from this type of analysis.

Table 3-3. Predicting Effort and Errors Using Size and Complexity Metrics[a]

Measure	Correlation	
	Effort	Errors
Calls	0.80	0.57
Cyclomatic Complexity	0.74	0.56
Calls + Jumps	0.80	0.58
Lines of Code	0.76	0.56
Executable Statements	0.74	0.55
Revisions	0.71	0.67
Halstead Effort	0.66	0.54

[a]From Reference 30, page 18, based on a study of SEL data.

Table 3-4. Internal Validation of Halstead Measures[a]

Relationship	Correlation	
	Large[b]	Small[c]
$N \sim \hat{N}$	0.79	0.83
$V \sim V^*$	0.52	0.50
$L \sim L$	0.71	0.62
$E \sim E$	0.61	0.42

[a]From Reference 30, page 20, based on a study of SEL data.

[b]Modules > 50 lines of code.

[c]Modules $\leq$ 50 lines of code.

The correlations reported in the table cannot, however, be taken at face value. For any given program studied, values A and B can be found so that the total number of operators and operands can be expressed as functions of the number of unique operators and operands. Consider the following equations:

$$N_1 = n_1 A \tag{1}$$

$$N_2 = n_2 B \tag{2}$$

$$N = n_1 A + n_2 B \tag{3}$$

and, according to Halstead (Reference 8):

$$\hat{N} = n_1 \log_2 n_1 + n_2 \log_2 n_2 \tag{4}$$

where $\hat{N}$ = predicted program length
N = actual program length ($N_1 + N_2$)
N_1 = total number of operators
N_2 = total number of operands
n_1 = number of unique operators
n_2 = number of unique operands
A, B = constants

Comparing Equations (3) and (4) shows that $\hat{N}$ and N are both functions of n_1 and n_2. Because the coefficients A, B, $\log_2 n_1$, and $\log_2 n_2$ are all always positive, a positive correlation must exist between $\hat{N}$ and N. The correlations shown in Table 3-4 may not be significant after accounting for the fact that all these quantities are functions of n_1 and n_2.

McCabe measures have also been the focus of extensive investigation by the SEL. Although McCabe made no such claims for his theory, others have attempted to relate cyclomatic complexity to error rate. As reported in Section 2.2.2, this has been only partially successful. The SEL (Reference 37) has found evidence that cyclomatic complexity and error rate may be uncorrelated or even negatively correlated--not a very satisfying conclusion. The position of the SEL is that, although analytic measures seem promising and are intellectually appealing, their practical value has not been demonstrated.

3.3 SUBJECTIVE MEASURES

The evaluation of software quality is, at present, a matter of the subjective interpretation of the results of a software development project relative to its functional requirements. This can be done best by managers and senior personnel associated with the project. Discussions among these individuals produce a consensus rating of the project relative to projects previously undertaken by the organization. Checklists and questionnaires can be employed to formalize the rating process.

Subjective measures offer the flexibility of easy tailoring to any situation. Hundreds of such measures have been suggested (see Section 2.3); however, the selection of appropriate measures is essential to systematizing the subjective process. Because each environment and application is unique, a single set of measures may not be appropriate to all.

The SEL conducted an exhaustive study to determine the measures that best characterize the flight dynamics environment (Reference 4). Over 600 measures were examined, from which 38 key properties (factors) were identified. Table 3-5 summarizes these results. The factors defined by the analysis

Table 3-5. Summary of Factor Analyses of Classes of Measures[a]

Class of Measures	No. of Measures	No. of Factors	Percent of Variance Explained
Software Engineering Practices	43	5	80
Development Team Ability	110	6	82
Difficulty of Project	54	5	74
Process and Product Characteristics	47	5	85
Development Team Background	144	5	86
Resource Model Parameters	73	6	73
Additional Detail	137	6	83

[a]From Reference 4, Table 4-2, based on a study of 20 flight dynamics projects from the SEL data base.

contained about 80 percent of the information (variance) of the original measures. This study showed that a concise set of subjective measures can be devised that effectively characterizes a given software development environment. One conclusion of this study was that project size influenced almost all aspects of software development, including staffing, methodology, and stability. Subsequent research efforts will define the relationship of these characteristics to the quality of the final software product. The general conclusion of the SEL is that, although subjective measures lack the precision and conciseness of explicit and analytic measures, they are an effective means of characterizing software quality.

SECTION 4 - APPLICATIONS OF MEASURES

Measures play many roles in the management of a software development project. These roles are illustrated at a high level in Figure 4-1. Measures provide the basis for management decisions about project control. The general roles of measures include the following:

- Predicting and planning--Estimating cost and quality to establish a baseline or development plan
- Reviewing and assessing--Measuring performance and quality during development
- Evaluating and selecting the best technology for an ongoing or future project

The goal of measurement is to detect significant departures from historical patterns. The regular and consistent application of measures will enable the software development manager to prevent or correct problems quickly and efficiently. This section describes the uses of some specific measures. More general guidelines for software development management are given in Reference 38.

The intent of this section is to demonstrate how measures can be used to answer some of the most common management questions. The recommendations presented here are not intended to constitute a complete or final guide to the uses of software measures; substantial improvements in this area will be forthcoming as additional research is performed. However, as Gilb (Reference 14) suggests, the currently available (if imperfect) measures should be used until others are developed. These specific measures have been used successfully in a software production environment.

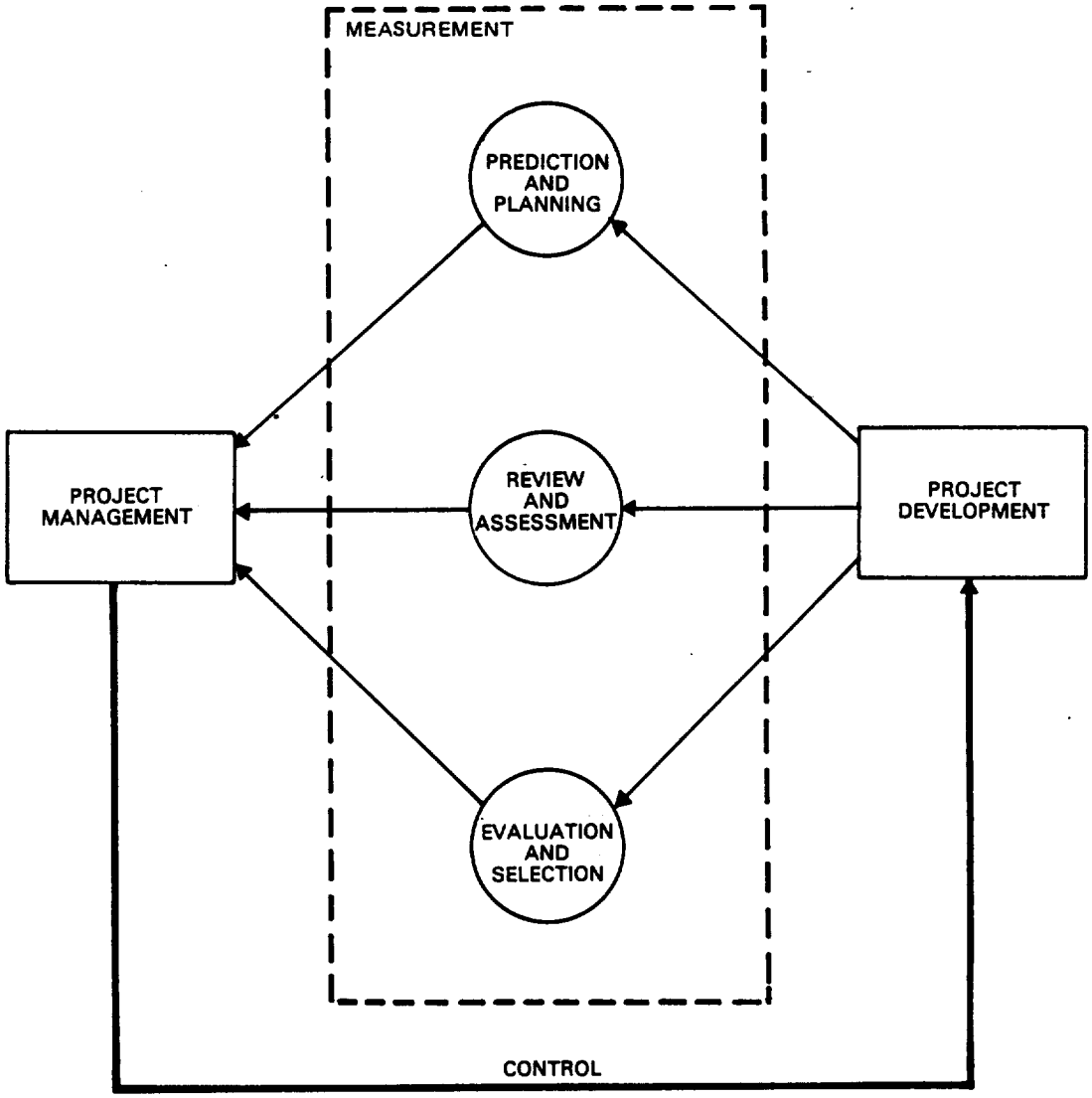

Figure 4-1. Role of Measures in Software Development Management

4.1 PREDICTION AND PLANNING

Effective software development management depends on reliable measurement and accurate estimation. The manager must produce initial estimates of system size, cost, and completion date. These estimates are incorporated in a development/management plan. Progress toward completion is measured against the plan. Plans and estimates must be updated periodically to reflect actual work accomplished.

This section shows how measures can be used to answer the manager's questions about the ultimate size, development cost, schedule, maintenance cost, and reliability of a software system. More detailed explanations of planning and estimation are presented in References 38 and 39, respectively.

How big will this system be when finished?	• Number or subsystems • Number of modules • Lines of code per subsystem • Lines of code per module • Lines of code developed to date • Current growth rate

An initial estimate of system size can be made by multiplying the number of subsystems by the average number of lines of code per subsystem. Once the high-level design is complete, an estimate can be made similarly by using the number of modules and the average lines of code per module. Table 4-1 lists values for these measures derived from SEL data. During implementation, the lines of code developed to date and the current growth rate can be combined to project the size of the completed system.

How much will this system cost to develop?	• Number or subsystems • Number of modules • Hours per subsystem • Hours per module • Percent of reused code • Expenditures to date • Life cycle effort model

Table 4-1. Basic Estimation Parameters[a]

Requirements Analysis[b]		Nominal Value[c]
Size:	Lines of code per subsystem	7500
Cost:	Hours per subsystem	1850
Schedule:	Weeks per subsystem per person	45
Preliminary Design[b]		
Size:	Lines of code per module	125
Cost:	Hours per module	30
Schedule:	Weeks per module per person	0.75
Detailed Design[b]		
Size:	Relative weight of reused[d] code	0.2
Cost:	Hours per developed line of code	0.3
Schedule:	Weeks per developed modules per person	1.0
Implementation		
Size:	Percent growth during testing	10
Cost:	Testing percent of total effort	25
Schedule:	Testing percent of total schedule	30
System Testing		
Cost:	Acceptance testing percent of total effort	5
Schedule:	Acceptance testing percent of total schedule	10

[a]At end of each phase, based on SEL data.

[b]Estimates of totals, not required to complete.

[c]Based on data collected in the flight dynamics environment.

[d]Does not include extensively modified reused module.

An initial estimate of system cost can be made by multiplying the number of subsystems by the average hours per subsystem. Once the high-level design is complete, an estimate can be made similarly by using the number of modules and the average hours per module. As modules that can be reused from other systems are identified, this estimate can be refined. Table 4-1 lists values for these measures derived from SEL data.

During development, the expenditures to date can be compared with the life cycle effort model (Table 4-2) to project the cost to complete the system. As shown in Table 4-2, the proportion of the total activity required for each life cycle phase is relatively stable. Thus, the actual expenditures to date at the end of any phase can be assumed to represent the corresponding percentage of the total expenditures required to complete development.

When will this system be completed?

- Weeks per subsystem per person
- Weeks per module per person
- Life cycle schedule model
- Time elapsed to date
- Modules per week
- Lines of code per week
- Modules per subsystem
- Lines of code per module

An initial estimate of development time can be made by multiplying the number of weeks required per subsystem per person by the number of subsystems, then dividing by the projected staff level (number of persons). Once the high-level design is complete, an estimate can be made similarly by using the number of weeks required per module per person and the number of modules. Table 4-1 lists values for these measures derived from SEL data.

During development, the time elapsed to date can be compared with the life cycle schedule model (Table 4-2) to project the time required to complete the system. As shown in

Table 4-2, the proportion of the total time required for each life cycle phase is relatively stable. Thus, the time elapsed to date at the end of any phase can be assumed to represent the corresponding percentage of the total time required to complete development.

Table 4-2. Life Cycle Effort/Schedule Model[a]

Life Cycle Phase	Percent of Total Schedule	Percent of Total Effort
Requirements Analysis	5	6
Preliminary Design	10	8
Detailed Design	15	16
Implementation	40	45
System Testing	20	20
Acceptance Testing	10	5

[a]Based on SEL experience.

The completion time for detailed design can be estimated by dividing the number of modules remaining to be designed by the current module design rate (modules per week). The completion time for implementation can be estimated by dividing the number of lines of code remaining to be produced by the current software production rate (lines of code per week).

Is this project on schedule?

- Stability of plans and staff
- Computer utilization
- Software production
- Staffing expenditures

Periodic reestimation of software size, cost, and schedule can necessitate changes in the development plan and personnel. Many such changes, however, can indicate that the development team does not have a good grasp of the software problem and is likely to fall farther behind in the future.

Finishing a project on time depends on the development team's doing the right thing, at the right time, fast enough to stay on schedule. Measurements of computer utilization and software production can suggest the type of activities the development team is engaged in and how fast they are working. Comparison of actual staff hours expended to date with planned expenditures at this date can indicate whether or not work is proceeding according to schedule.

Software production usually progresses at a constant rate throughout implementation, as shown in Figure 4-2. Plotting the percent completed of the total software estimated against the percent of the schedule elapsed indicates project status. A development project that starts producing code before the expected start of implementation may be working from an inadequate design. Too rapid code production during implementation suggests that inadequate unit testing is being performed. Slow code production results in the project falling behind schedule. Figure 4-2 identifies the regions of the software production graph associated with these problems.

Computer utilization follows software production, increasing constantly during implementation. Computer utilization should, however, stabilize and then fall rapidly during testing as tests are completed. Figure 4-3 shows this pattern. Significant computer use during design (unless automated design tools are used) suggests that coding has actually started too early. A low level of computer use during implementation indicates that code production and/or unit testing is behind schedule. A decline in computer utilization at any time during implementation is a sign that development has been interrupted. Any of these problems may result in an integration crunch during testing when resources are added to the project in an effort to complete

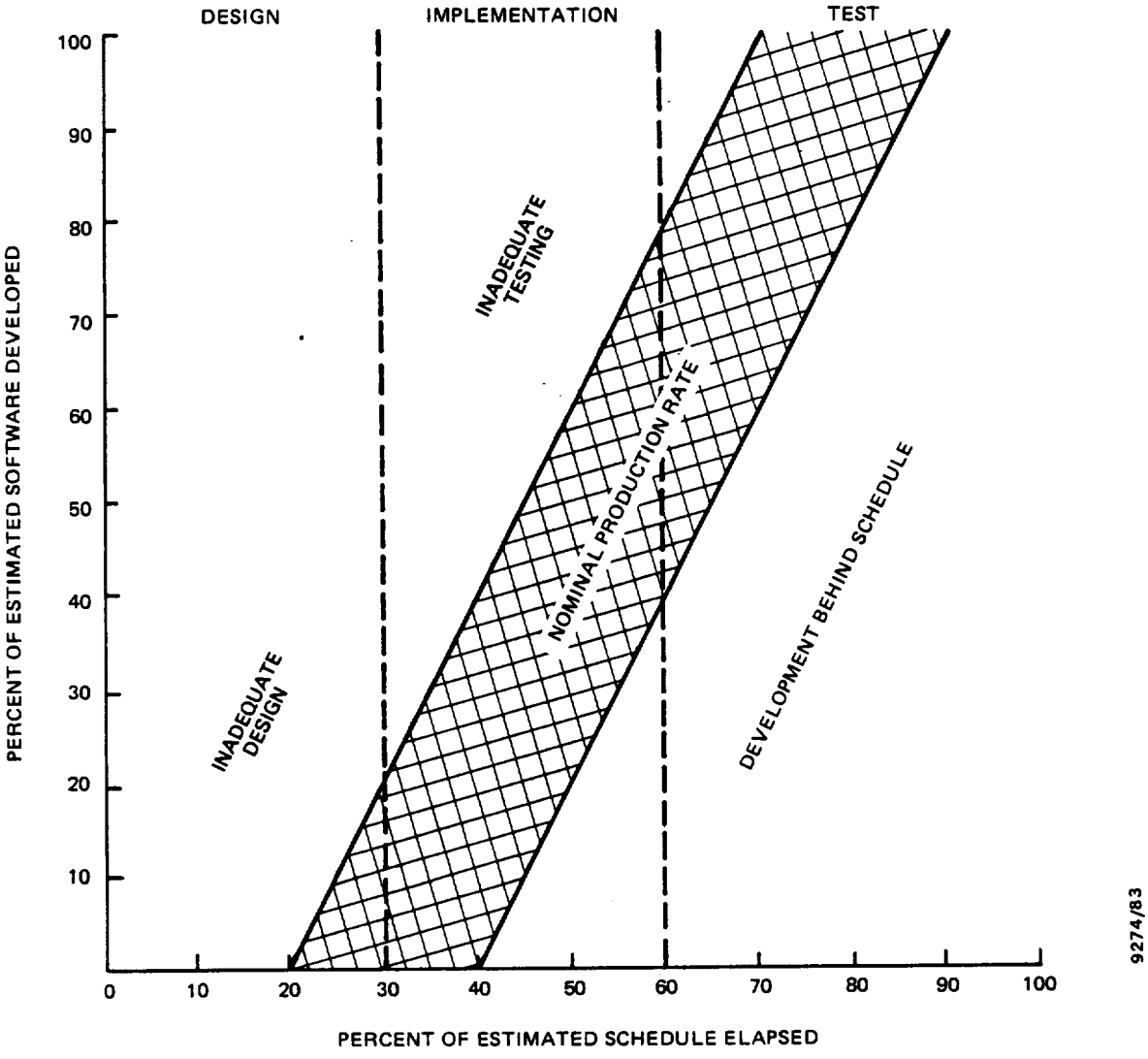

Figure 4-2. Nominal Software Production Pattern

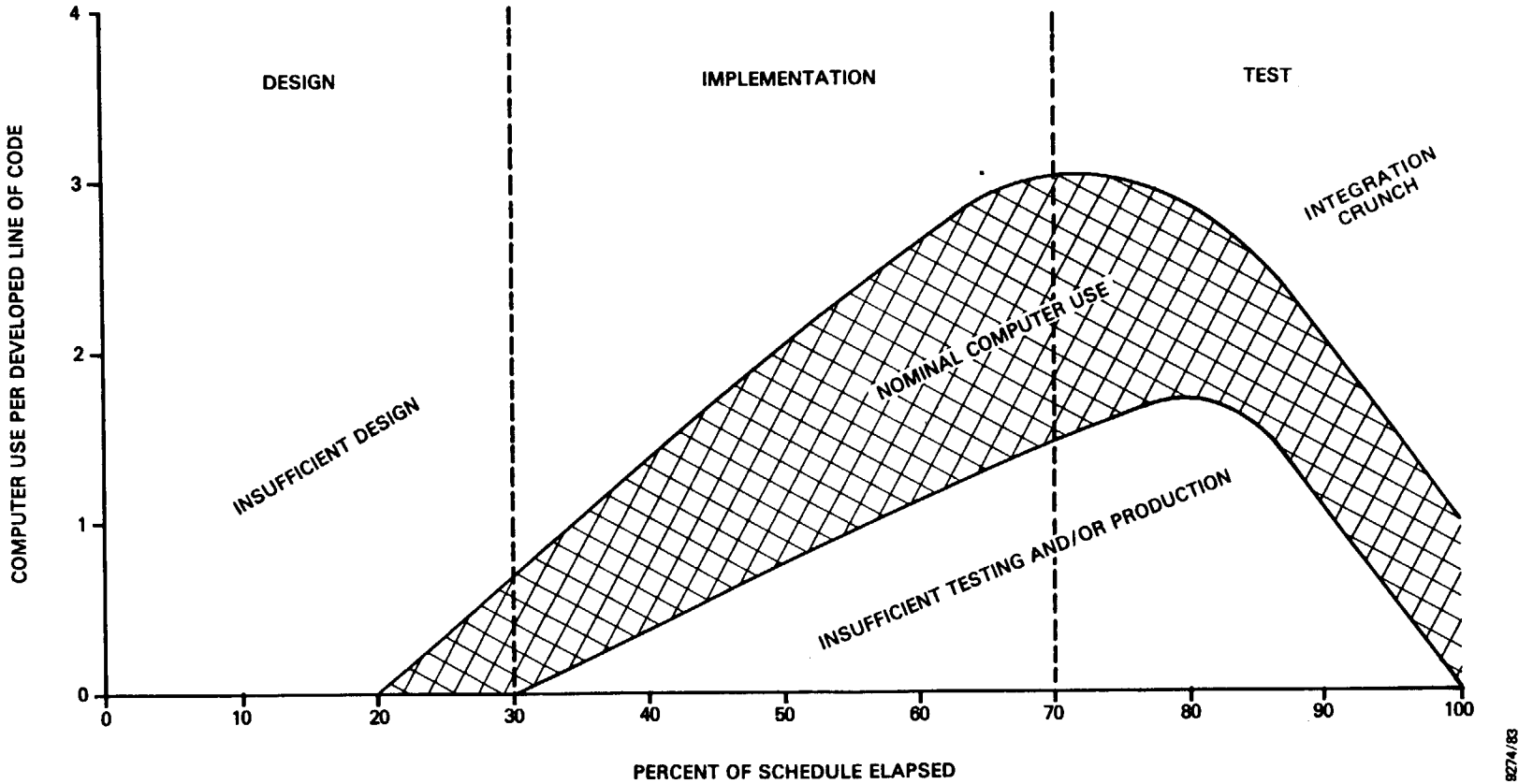

Figure 4-3. Nominal Computer Utilization Pattern

it on schedule. Figure 4-3 identifies the regions of the computer utilization graph associated with these phenomena.

How much will this system cost to maintain?

- Development cost
- Percent of reused code
- Implementation error rate
- Testing error rate
- Effort to change

The annual cost of software maintenance is about 25 percent of the development cost (Reference 39). However, any given project may cost more or less than that to maintain. A high error rate (errors per thousand lines of code) during implementation and/or testing suggests that a system will cost more than usual to maintain. The effort to change (hours per change) measured during development is another indicator of relative maintenance cost.

How reliable will this software be?

- Implementation error rate
- Testing error rate
- Software change rate
- Number of requirements changes

The implementation and testing error rates (errors per thousand lines of code) provide the first indications of the reliability of the delivered software product. During testing, the error rate should peak and begin declining. Failure of the error rate to decline during testing suggests that many undiscovered errors remain in the software. Late requirements changes can also introduce errors and inconsistencies into the system. Some of these effects can be traced in the software change rate.

The software change rate cannot be measured until implementation, when software production begins. Development techniques such as configuration control and online development affect the overall change rate. The cumulative change rate should, however, increase steadily throughout implementation and testing as shown in Figure 4-4. A static (level) change

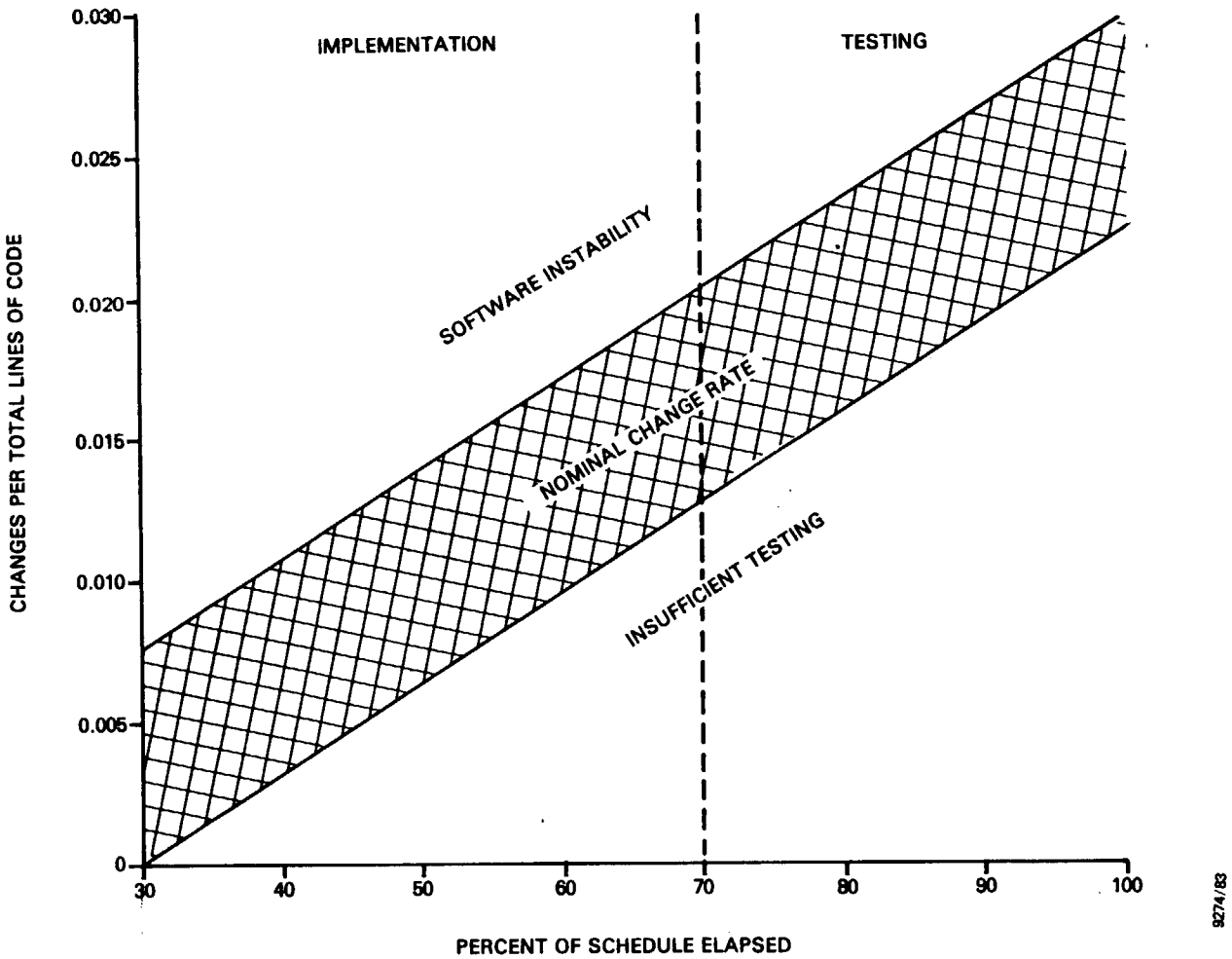

Figure 4-4. Nominal Software Change Pattern

rate indicates that testing and error correction are proceeding too slowly. A rapid increase in the cumulative change rate suggests that the software is unstable; that the developers are making ill-considered and possibly contradictory changes, perhaps in response to sudden requirements changes. Figure 4-4 identifies the regions of the software change graph associated with these problems.

4.2 REVIEW AND ASSESSMENT

Throughout the software life cycle, the development team produces and delivers products that eventually make up the completed software system. The manager must evaluate each of these products as well as the team's overall performance. This section addresses those questions that can be asked about the quality of requirements, design, software, testing, documentation, and performance. Table 4-3 lists nominal values for some applicable measures based on SEL data. The specific uses of these measures are discussed below.

Are the requirements complete?	• To be determined items (TBDs) • TBD rate • Severity of TBDs

Although there is no set of measures that answers this question directly, experience shows that the number and type of "to be determined" items (TBDs), as well as the rate of change in the TBDs are very strong indicators of the completeness of requirements. Any set of requirements will contain some TBDs, but an excessive amount can indicate trouble. An increase in the number of TBDs near the time when requirements are due to be completed is an even stronger indication that more work must be done before preliminary design can begin. Such an increase is interpreted as a sign that more weaknesses in the requirements are uncovered as they are looked at more closely. An assessment of the completeness of requirements must also incorporate the severity of TBDs. TBDs in specific algorithms and

Table 4-3. Measures for Assessment

Product	Measure	Nominal[a] Value
Requirements	TBDs per subsystem	3
	Questions per subsystem	8
	ECRs per subsystem	5
Design	Internal interfaces not defined (%)	5
	External interfaces not defined (%)	5
	Modules not defined (%)	5
Software[b]	Errors per thousand developed lines of code	7
	Changes per thousand developed lines of code	14
	Effort to repair (hours)	8
	Effort to change (hours)	8
	Modules affected per change	1
Testing	Module coverage (%)	100
	Function coverage (%)	100
	Errors per thousand developed lines of code	3
Documentation	Pages per module	2
	Checklist completeness (%)	100
Performance	Developed lines of code per staff hour	3
	Schedule changes	5[c]
	Reused code (%)	30
	Estimate Changes	5[c]

[a]Based on SEL historical data for flight dynamics software.

[b]Measured during implementation.

[c]Once per phase and build.

tolerances, for example, are much less critical than TBDs in external interface formats and operational constraints (e.g., memory, timing, data rate).

Are the requirements accurate?	• To be determined items (TBDs) • Engineering change requests (ECRs)

Much effort has been expended by research organizations (including the SEL) to develop specific approaches and measures for assessing the completeness and accuracy of software requirements. Success to date has been very limited. Approaches such as traceability matrices, requirements languages, and cross-check tables have not been fully effective.

At this time, two of the more reliable measures for determining the accuracy of software requirements are the number of TBDs listed in the requirements and the number of engineering change requests (ECRs) generated during the requirements analysis phase. Exceptionally large values for these parameters can indicate that the requirements need to be redeveloped.

Is the design complete?	• Number of modules not identified • Number of modules not defined • Number of module interfaces not defined • Number of external interfaces not defined

Although the definition of the design activity and corresponding criteria can vary from environment to environment, the informational content of a complete design is relatively standard. Four basic measures of design completeness are generally applicable. The structure chart must identify all modules (software items) to be produced. Processing descriptions (PDL or prologs) must be provided for all modules. Interfaces among modules (e.g., calling sequences and COMMON blocks) must be defined. All external interfaces must be defined to the bit level. It is not always possible to specify all of these items before starting implementation. However, counting the number of TBDs in each area

provides a good measure of design completeness. More than 5 percent TBDs in any area is an indication that the design is not ready and implementation should be postponed.

Is the design effective (relatively the best)?	• Module strength • Module coupling • External I/O isolation

No reliable objective measures of design quality have been identified. However, three subjective measures have been found to be useful in this context. These measures can be determined only by inspecting the module process descriptions, although efforts continue to develop corresponding automatable measures. High module strength (singleness of purpose) produces a relatively high-quality design when maintainability and robustness are concerns. Another relevant measure is module coupling (interdependence). Many interdependencies make changes to the software difficult and error prone. The degree of external I/O isolation is the number of modules accessing external files. Ideally, only one module should access each file. Failure to isolate external I/O activities often leads to lower reliability.

Is the software too complex (or is it modular)?	• Effort to change • Effort to repair • Modules affected per change • Module strength • Module coupling

Although numerous analytic measures have been proposed as straightforward means of determining the complexity of software, the SEL has been unable to verify their effectiveness in this application (see Section 3.2). Measures that have not proved effective include module size (average lines of code per module), cyclomatic (and central) complexity, and Halstead measures. Many successful software developers believe that smaller modules are generally less complex than larger ones, and therefore better. SEL research has not,

however, shown any direct correlation between module size and cost or reliability (Reference 37).

Based on SEL experience, the most effective measures of software complexity and modularity are the effort required to make a change, the effort required to repair an error, and the number of modules affected by a change. Simple modular development will minimize these quantities. Values for these measures can be determined during implementation by monitoring changes and errors. The most reliable measure of software complexity and modularity, however, is the judgment of an experienced software development manager. This judgment may be based on an assessment of module strength (singleness of purpose) and module coupling (interdependence) as well as knowledge of the application area and similar systems.

Is the software maintainable?

- Effort to change
- Effort to repair
- Modules affected per change
- Errors outstanding

Because software maintenance is often the most expensive phase of the software life cycle, it is important that the completed software be easy to maintain. Low complexity and good modularity facilitate maintenance, so some of the relevant measures are the same. Effort required to make a change, effort required to repair an error, and number of modules affected by a change are good indicators of the relative difficulty and cost of software maintenance. Lower values for these measures imply better maintainability.

Another useful measure is the number of errors outstanding. Errors are discovered throughout the software life cycle and, after some delay, are repaired. When the rate of discovery exceeds the rate of repair during maintenance, it

may be time to redevelop or replace the software. This measure may indicate the end of the software life cycle.

Is the software reliable?

- Error rate
- Change rate
- Modules affected per change

The basic measure of software reliability is how often the software fails. It depends on the number and severity of errors. However, other indicators are also important in deciding how much confidence can be placed in a software system. The three measures relied on by the SEL are as follows:

1. Errors per thousand lines of code--This quantity, measured during system and acceptance testing, can be compared with values from previous systems to determine relative reliability. An error rate in excess of 3 per 1000 developed lines identifies an unreliable system. Furthermore, any increase in the error rate late in development indicates a problem with system reliability.

2. Changes per thousand lines of code--This quantity, measured during system and acceptance testing, can identify reliability problems. Although some changes may be requirements changes or clarifications, a high change rate usually indicates future unreliability.

3. Number of modules affected per change--Highly coupled software tends to propagate errors and confound change attempts. A high value for this measure indicates that maintenance will be difficult and reliability will be low.

Although many comprehensive reliability models have been developed and occasionally successfully applied, SEL

experience (Reference 37) suggests that they are not very effective in the Flight Dynamics environment.

Is system testing complete (or adequate)?

- Function coverage
- Module coverage
- Error discovery rate

The two basic approaches to software testing are functional and structural. Functional testing attempts to maximize the number of functional capabilities tested based on the description of functionality contained in the requirements specification. Structural testing attempts to maximize the number of software structures tested without regard for functionality. Approximate measures corresponding to these approaches are function coverage and module coverage.

Function coverage is the percentage of functions identified in the requirements that are exercised during system and acceptance testing. Module coverage is the percentage of modules and other software components that are exercised during system and acceptance testing. An effective test plan will exercise 100 percent of the functions and modules. It is not, however, generally possible to test every line of code or every path through the system. SEL research indicates that good functional testing may exercise only 70 percent of the code, but that has proven to be adequate (Reference 40). The number of individual tests defined in the test plan is not a good measure of test completeness.

The error discovery rate can also indicate when sufficient testing has been done. Failure of this rate to decline toward the end of planned testing suggests that more testing needs to be done. The error discovery rate during maintenance and operation will be the same as at the end of testing unless additional effort is expended to find and correct errors.

Is the documentation appropriate?

- Pages per module
- Checklist completeness
- Subjective assessment
- Expected lifetime

Much useful documentation is generated as an intrinsic part of the software development process (e.g., design description). However, any nontrivial system will require additional documentation to support users after development is complete. This documentation must support both maintenance (by programmers) and operation (by users). In the flight dynamics environment studied by the SEL, this information is frequently presented in two separate documents, a system description and a user's guide.

The amount and formality of documentation required depends on the size and expected lifetime of the system. Generally, about two pages of documentation per module should be produced. Excessive documentation can be as awkward as insufficient documentation. Long-lived systems need more detailed and formal documentation. Short-lived systems need only minimal documentation. Document completeness can be determined by comparison with a checklist of standard contents. Realistically, document quality can only be determined by a subjective assessment.

Is the product cost effective?

- Productivity
- Reused code
- Error rate
- Effort to change
- Effort to repair

The cost effectiveness of a product is a function of its initial cost to develop and subsequent cost to maintain. Although often criticized as an inadequate measure of productivity, SEL experience indicates that the measure "lines of code developed per staff hour expended" is a reliable way of evaluating the cost effectiveness of development when consistent historical data are available for comparison.

Lines of code and hours charged must, however, be clearly defined. Another measure of the cost effectiveness of development is the percentage of reused code in the new system. Reusing previously developed code costs only 20 percent as much as developing new code.

The error rate (errors per thousand lines of code), effort required to make a change, and effort required to repair an error, are good indicators of maintenance cost. A system that has few errors and that is easily modified will be inexpensive to maintain. Low maintenance and development costs characterize a cost-effective product.

Team performance during development can be monitored by plotting cumulative productivity. The starting point of the cumulative productivity graph depends on the amount of reused code. Figure 4-5 shows the productivity pattern for a project reusing up to 15 percent of code. In the figure, productivity increases steadily throughout implementation. A very rapid increase in productivity suggests that software is being developed without adequate unit testing. Too slow an increase implies that development is falling behind schedule.

Extensive reuse of existing code raises the starting level of cumulative productivity, and thus its path may be level or even declining during implementation. In all cases, however, productivity should be level or should decline slightly during testing. If productivity continues to increase instead, implementation is not complete; coding is still in progress. A sharp decline in productivity during testing reflects an integration crunch when resources are added to the project in an effort to complete it on schedule. Figure 4-5 identifies the regions of the cumulative productivity graph associated with these phenomena.

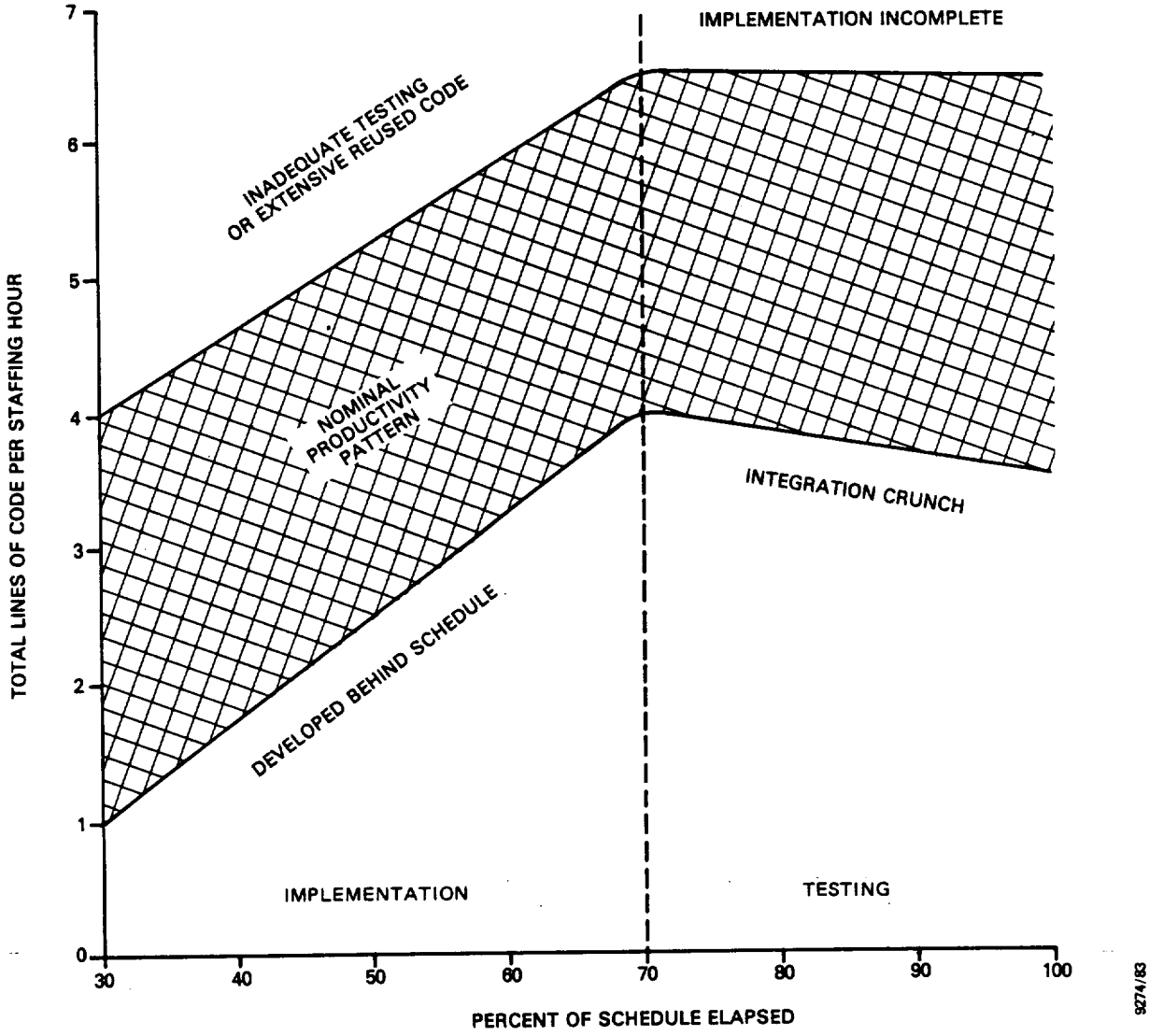

Figure 4-5. Nominal Productivity Pattern

Is staffing at the right level?

- Staffing profile
- Schedule changes
- Estimate changes
- Code production rate

Two major concerns of a software development manager are the team's size and its capabilities. The first concern is usually whether the team size is optimum--neither too many nor too few members. Two effective measures in this regard are the size of departures from the planned staffing profile (perhaps a Rayleigh curve) and the frequency of schedule and milestone changes.

Departures from planned staffing usually indicate that production will depart from plan, too. Unless major changes are made to the system requirements, schedules and milestones should be adjusted only once at the end of each life cycle phase. Frequent changes in size and cost estimates can imply that the estimates are being adjusted to fit an inappropriate staffing level or that the skill mix of the development team is inappropriate to the task.

During implementation, the code production rate (lines of code per month) can be used to project the completion date of development (see Section 4.1). Staff can be added or subtracted to make that date match the schedule. Rarely does individual productivity change during development, so the manager should not expect to change the team production rate except by altering the staff level and/or skill mix.

4.3 EVALUATION AND SELECTION

During the software development process as well as during predevelopment planning, the manager must select the tools, practices, and techniques most appropriate to the specific software development project in progress. Measures facilitate the comparison of the ongoing project with previous projects and highlight any special considerations. The

process of evaluating the overall cost effectiveness of individual technologies is not considered here.

This section addresses the use of measures to evaluate and select appropriate strategies for a planned or ongoing software development project. The major areas of concern are development methodology, testing approach, team organization, and level of standards.

Which methodology is appropriate?

- Percent reused code
- Number of external files
- Similarity to past projects
- Team experience
- Size of project

A methodology can consist of one or more integrated tools, techniques, and practices. Methodologies provide the development team with a common form of communication and organize its activities into integrated cooperative subactivities. Many different methodologies are used in the software engineering community. The general class of "structured" techniques is probably the most widely employed. SEL experience with over 40 flight dynamics projects has identified five principal measures relevant to selecting an appropriate software development methodology.

The percent of reused code is an important consideration when deciding that top-down design, coding, and testing are to be used. The SEL has found the strict application of "top-down" techniques to be less effective as the percent of reused code (or design) increases.

Another relevant parameter is the number of external files defined for the project. A large number of external files indicates that the software is "data processing" rather than "computational." The use of a structured design or structured analysis methodology has been found to be more valuable with data processing systems.

Another consideration is the similarity to past projects. The SEL has found that, as the similarity to past projects increases, the need for a completely structured development methodology decreases. New project types require a much more disciplined development approach.

A fourth useful measure is the relative experience of the development team. Although an experienced team does not always use a very structured and disciplined methodology, the SEL has found that an experienced team will automatically select an effective approach. On the other hand, a less experienced team should employ a single well-defined (typically structured) methodology.

Project size is also a major consideration when selecting a methodology. The manager will find that less formal, less structured methodologies are very workable for smaller projects (e.g., less than 2 or 3 staff-years). However, larger projects (especially those greater than 5 or 6 staff-years of effort) need the discipline of a structured approach.

What testing approach should be employed?

- Size of project
- Percent reused code
- Reliability requirements

Software testing and verification can consume the major portion of development resources. Consequently, a testing strategy must be selected with care. There are two general approaches to testing software: functional and structural (see Section 4.2). These can be implemented by an independent test team or the development team. The independent test team often assumes a verification role early in the development process, in which case it is referred to as an independent verification and validation (IV&V) team. Although the extent of testing is obviously a function of the software reliability requirements, several other measures also

may help in selecting the most appropriate approach to system testing.

Size and required reliability are the important determinants of whether or not the IV&V approach is worthwhile. For projects with average reliability requirements, IV&V is effective only for those projects greater than 20 staff-years of effort. IV&V is cost effective, however, for any project with an exceptionally high reliability requirement.

Although the projects studied by the SEL produced generally successful (reliable) software with the application of functional testing, some experiments indicated that for unusually high reliability requirements the structural (statement and path coverage) approach may be more appropriate. The selection of testing approach also depends on the percent of reused code. Above 30 percent, functional testing seems to be fully adequate.

What team organization is appropriate?

- Size of project
- Team experience
- Similarity to past projects
- Percent reused code

There are many general structures into which a software development project can be organized. The most common organization is the chief programmer team (CPT). In addition, the project can be subdivided into functional teams (e.g., quality assurance). The principal alternatives to CPT are fluid organizations such as the democratic team. The best organization for a given project depends on a number of factors. The most important of which is that the smaller the team, the better (Reference 41). Whenever possible, every team member should be assigned full time.

Project size is the principal criterion for deciding whether or not separate quality assurance or configuration control

teams are called for. If a project is less than 12 to 15 staff-years, it probably will not be cost effective to organize it into separate groups with these responsibilities.

Team experience is the principal criterion when deciding whether or not a CPT should be applied. SEL experience indicates that projects with very experienced personnel have not derived any benefit from CPT. On the other hand, teams with average or less than average experience with the specific application can benefit from CPT. However, the successful use of CPT requires an application expert with a natural capability for the chief programmer role.

Two other considerations when selecting the team structure are the similarity to past projects and percent of reused code. SEL experience shows that as these measures increase (higher similarity and higher percent of reused code), the need for the CPT organization and the need for independent functional organizations responsible for quality assurance and configuration control decrease.

What type and levels of standards should be applied?

- Size of project
- Schedule changes
- Change rate
- Error rate
- Similarity to past projects
- Percent reused code

Whether they are called standards, guidelines, policies, or something else, some such set of written development practices must be prescribed for every project. SEL experience with flight dynamics projects shows that, as projects increase in size, the need for design, coding, and implementation standards also increases. Projects of less than 2 staff-years can be completed quite satisfactorily with minimum written standards.

Three other measures can indicate a need for a change in the level of standards during development. If the error rate,

change rate, or frequency of schedule changes increases, the manager should reconsider the level and type of development standards being applied. Policies should be revised or enforced more strongly if these measures indicate problems. Finally, projects with a high percent of reused code and high similarity to a past project often benefit from a flexible set of design, code, and test standards.

SECTION 5 - CONCLUSIONS

The preceding sections showed that a wide range of software development measures are available to the software practitioner. Some have important applications. The SEL has arrived at the following general conclusions:

- Explicit measures are very effective for some software development characteristics.
- Although analytic measures seem promising and are intellectually appealing, their practical value has not been demonstrated.
- Subjective measures are an effective means of characterizing software quality.

Substantial work remains to be done in all of these areas. Formulation, evaluation, and application of measures is a continuous activity that contributes to and profits from a growing understanding of the software development process. A comprehensive system of measurement is a necessary prerequisite to any effort to evaluate or improve the software development process and the available software engineering technologies (References 42 and 43). This document will be revised and extended as more is learned about measures.

Currently, the SEL is making a major effort to identify measures of software size and complexity that can be applied early in the software life cycle (during requirements and design). The SEL is also attempting to automate the measurement process throughout the software life cycle. The ultimate goal of these activities is to produce a management tool that will monitor the progress of a software project and compare it with a historical data base of similar projects, thus allowing the manager to ask and answer questions such as those discussed in this document. Reference 44 explains these concepts in more detail.

REFERENCES

1. V. R. Basili, editor, Models and Metrics for Software Management and Engineering, New York: Computer Societies Press, 1980

2. Software Engineering Laboratory, SEL-81-101, Guide to Data Collection, V. E. Church, D. N. Card, and F. E. McGarry, August 1982

3. --, SEL-81-104, The Software Engineering Laboratory, D. N. Card, F. E. McGarry, G. Page, et al., February 1982

4. --, SEL-82-001, Evaluation of Management Measures of Software Development, G. Page, D. N. Card, and F. E. McGarry, September 1982, Volumes 1 and 2

5. --, SEL-82-005, Glossary of Software Engineering Laboratory Terms, M. R. Rohleder, December 1982

6. C. E. Walston and C. P. Felix, "A Method of Programming Measurement and Estimation," IBM Systems Journal, January 1977

7. V. R. Basili, "Product Metrics," Models and Metrics for Software Management and Engineering. New York: Computer Society Press, 1980

8. M. H. Halstead, Elements of Software Science. New York: Elsevier North-Hollard, 1977

9. Software Engineering Laboratory, SEL-78-102, FORTRAN Static Source Code Analyzer Program (SAP) User's Guide (Revision 1), W. J. Decker and W. A. Taylor, September 1982

10. A. Fitsimmons and T. Love, "A Review and Evaluation of Software Science," ACM Computing Surveys, March 1978

11. V. Y. Shen, S. D. Conte, and H. E. Dunsmore, "Software Science Revisited: A Critical Analysis of the Theory and Its Empirical Support," IEEE Transactions on Software Engineering, vol. 9, no. 2, March 1983

12. N. S. Coulter, "Software Science and Cognitive Psychology," IEEE Transactions on Software Engineering, vol. 9, no. 2, March 1983

13. T. J. McCabe, "A Complexity Measure," IEEE Transactions on Software Engineering, December 1976

14. T. Gilb, Software Metrics. Cambridge: Winthrop Publishers, 1977

15. G. Myers, "An Extension to the Cyclomatic Measure of Program Complexity," ACM SIGPLAN Notices, October 1977

16. W. Curtis, S. B. Sheppard, and P. Milliman, "Third Time Charm: Stronger Prediction of Programmer Performance by Software Complexity Metrics," Proceedings: 4th International Conference on Software Engineering. New York: Computer Society Press, 1979

17. A. L. Baker and S. H. Zweben, "A Comparison of Measures of Control Flow Complexity," IEEE Transactions on Software Engineering, vol. 6, no. 6, November 1980

18. M. Evangelist, "Software Complexity Metric Sensitivity to Program Structuring Rules and Other Issues in Software Complexity," Sixth Minnowbrook Workshop on Software Performance Evaluation, July 1983

19. W. Hansen, "Measurement of Program Complexity by the Pair (Cyclomatic Number, Operator Count)," ACM SIGPLAN Notices, March 1978

20. J. L. Elshoff, "Analysis of Some Commercial PL/I Programs," IEEE Transactions on Software Engineering, vol. SE-2, June 1976

21. W. Harrison, K. Magel, et. al., "Applying Software Complexity Metrics to Program Maintenance," IEEE Computer, September 1982

22. S. Henry and D. Kafura, "Software Structure Metrics Based on Information Flow," IEEE Transactions on Software Engineering, vol. 7, no. 5, September 1981

23. J. A. McCall, P. K. Richards, and G. F. Walters, RADC-TR-77-369, Factors in Software Quality, Rome Air Development Center, vol. 1, November 1977

24. R. C. San Antonio, Jr., and K. L. Jackson, "Application of Software Metrics During Early Program Phases," Proceedings of the National Conference on Software Test and Evaluation, February 1983

25. J. A. McCall and M. A. Herndon, "Measuring Quality During Software Maintenance," Sixth Minnowbrook Workshop on Software Performance Evaluation, July 1983

26. J. Post, "Software Quality Metrics for Distributed Systems," Proceedings of the Sixth Annual Software Engineering Workship, SEL-81-013, December 1981

27. G. J. Myers, Reliable Software Through Composite Design, New York: Petrocelli/Charter, 1975

28. R. D. Cruickshank and J. E. Gaffney, "Measuring the Software Development Process: Software Design Coupling and Strength Matrices," Proceedings From the Fifth Annual Software Engineering Workshop, SEL-80-006, November 1980

29. B. A. Sheil, "The Psychological Study of Programming," ACM Computing Surveys, vol. 13, no. 1, March 1981

30. V. R. Basili, "Evaluating Software Development Characteristics: Assessment of Software Measures in the Software Engineering Laboratory," Proceedings of Sixth Annual Software Engineering Workshop, SEL-81-013, December 1981

31. V. R. Basili and K. Freburger, "Programming Measurement and Estimation in the Software Engineering Laboratory," Journal of Systems and Software, February 1981

32. J. W. Bailey and V. R. Basili, "A Meta-Model for Software Development Resource Expenditures," Proceedings of the Fifth International Conference on Software Engineering, New York: Computer Society Press, 1981

33. D. N. Card, "Early Estimation of Resource Expenditures and Program Size," Computer Sciences Corporation, Technical Memorandum, June 1982

34. G. Hislop, "Some Tests of Halstead Measures" (paper prepared for the University of Maryland, December 1978)

35. S. F. Lange, "A Child's Garden of Complexity Measures" (paper prepared for the University of Maryland, December 1978)

36. V. R. Basili and T. Phillips, "Evaluating and Comparing Software Metrics in the Software Engineering Laboratory," Proceedings of the ACM SIGMETRICS Symposium/ Workshop: Quality Metrics, March 1981

37. F. E. McGarry, "What Have We Learned in 6 Years?", Proceedings of the Seventh Annual Software Engineering Workshop, SEL-82-007, December 1982

38. Software Engineering Laboratory, SEL-81-105, Recommended Approach to Software Development, F. E. McGarry, G. Page, S. Eslinger, et al., April 1983

39. --, SEL-83-001, An Approach to Software Cost Estimation, F. E. McGarry, G. Page, D. N. Card, et al., January 1984

40. J. A. Ramsey and V. R. Basili, "Structural Coverage of Functional Testing," Proceedings of the Eighth Annual Software Engineering Workshop, SEL-83-007, November 1983

41. R.C. Tausworthe, "Staffing Implications of Software Productivity Models," Proceedings of the Seventh Annual Software Engineering Workshop, SEL-82-007, December 1982

42. W. W. Agresti, F. E. McGarry, D. N. Card, et al., "Measuring Software Technology," Program Transformation and Programming Environments. New York: Springer-Verlag, 1984

43. D. N. Card, F. E. McGarry, and G. Page, "Evaluating Software Engineering Technologies in the SEL," Proceedings of the Eighth Annual Software Engineering Workshop, SEL-83-007, November 1983

44. Software Engineering Laboratory, SEL-83-006, Monitoring Software Development Through Dynamic Variables, C. W. Doerflinger, November 1983

BIBLIOGRAPHY OF SEL LITERATURE

The technical papers, memorandums, and documents listed in this bibliography are organized into two groups. The first group is composed of documents issued by the Software Engineering Laboratory (SEL) during its research and development activities. The second group includes materials that were published elsewhere but pertain to SEL activities.

SEL-ORIGINATED DOCUMENTS

SEL-76-001, Proceedings From the First Summer Software Engineering Workshop, August 1976

SEL-77-001, The Software Engineering Laboratory, V. R. Basili, M. V. Zelkowitz, F. E. McGarry, et al., May 1977

SEL-77-002, Proceedings From the Second Summer Software Engineering Workshop, September 1977

SEL-77-003, Structured FORTRAN Preprocessor (SFORT), B. Chu and D. S. Wilson, September 1977

SEL-77-004, GSFC NAVPAK Design Specifications Languages Study, P. A. Scheffer and C. E. Velez, October 1977

SEL-78-001, FORTRAN Static Source Code Analyzer (SAP) Design and Module Descriptions, E. M. O'Neill, S. R. Waligora, and C. E. Goorevich, February 1978

[1]SEL-78-002, FORTRAN Static Source Code Analyzer (SAP) User's Guide, E. M. O'Neill, S. R. Waligora, and C. E. Goorevich, February 1978

SEL-78-102, FORTRAN Static Source Code Analyzer Program (SAP) User's Guide (Revision 1), W. J. Decker and W. A. Taylor, September 1982

SEL-78-003, Evaluation of Draper NAVPAK Software Design, K. Tasaki and F. E. McGarry, June 1978

SEL-78-004, Structured FORTRAN Preprocessor (SFORT) PDP-11/70 User's Guide, D. S. Wilson and B. Chu, September 1978

SEL-78-005, Proceedings From the Third Summer Software Engineering Workshop, September 1978

SEL-78-006, GSFC Software Engineering Research Requirements Analysis Study, P. A. Scheffer and C. E. Velez, November 1978

SEL-78-007, Applicability of the Rayleigh Curve to the SEL Environment, T. E. Mapp, December 1978

SEL-79-001, SIMPL-D Data Base Reference Manual, M. V. Zelkowitz, July 1979

SEL-79-002, The Software Engineering Laboratory: Relationship Equations, K. Freburger and V. R. Basili, May 1979

SEL-79-003, Common Software Module Repository (CSMR) System Description and User's Guide, C. E. Goorevich, A. L. Green, and S. R. Waligora, August 1979

SEL-79-004, Evaluation of the Caine, Farber, and Gordon Program Design Language (PDL) in the Goddard Space Flight Center (GSFC) Code 580 Software Design Environment, C. E. Goorevich, A. L. Green, and W. J. Decker, September 1979

SEL-79-005, Proceedings From the Fourth Summer Software Engineering Workshop, November 1979

SEL-80-001, Functional Requirements/Specifications for Code 580 Configuration Analysis Tool (CAT), F. K. Banks, A. L. Green, and C. E. Goorevich, February 1980

SEL-80-002, Multi-Level Expression Design Language-Requirement Level (MEDL-R) System Evaluation, W. J. Decker and C. E. Goorevich, May 1980

SEL-80-003, Multimission Modular Spacecraft Ground Support Software System (MMS/GSSS) State-of-the-Art Computer Systems/Compatibility Study, T. Welden, M. McClellan, and P. Liebertz, May 1980

[1]SEL-80-004, System Description and User's Guide for Code 580 Configuration Analysis Tool (CAT), F. K. Banks, W. J. Decker, J. G. Garrahan, et al., October 1980

SEL-80-104, Configuration Analysis Tool (CAT) System Description and User's Guide (Revision 1), W. Decker and W. Taylor, December 1982

SEL-80-005, A Study of the Musa Reliability Model, A. M. Miller, November 1980

SEL-80-006, Proceedings From the Fifth Annual Software Engineering Workshop, November 1980

SEL-80-007, An Appraisal of Selected Cost/Resource Estimation Models for Software Systems, J. F. Cook and F. E. McGarry, December 1980

[1]SEL-81-001, Guide to Data Collection, V. E. Church, D. N. Card, F. E. McGarry, et al., September 1981

SEL-81-101, Guide to Data Collection, V. E. Church, D. N. Card, F. E. McGarry, et al., August 1982

[1]SEL-81-002, Software Engineering Laboratory (SEL) Data Base Organization and User's Guide, D. C. Wyckoff, G. Page, and F. E. McGarry, September 1981

SEL-81-102, Software Engineering Laboratory (SEL) Data Base Organization and User's Guide Revision 1, P. Lo and D. Wyckoff, July 1983

[1]SEL-81-003, Data Base Maintenance System (DBAM) User's Guide and System Description, D. N. Card, D. C. Wyckoff, and G. Page, September 1981

SEL-81-103, Software Engineering Laboratory (SEL) Data Base Maintenance System (DBAM) User's Guide and System Description, P. Lo and D. Card, July 1983

[1]SEL-81-004, The Software Engineering Laboratory, D. N. Card, F. E. McGarry, G. Page, et al., September 1981

SEL-81-104, The Software Engineering Laboratory, D. N. Card, F. E. McGarry, G. Page, et al., February 1982

[1]SEL-81-005, Standard Approach to Software Development, V. E. Church, F. E. McGarry, G. Page, et al., September 1981

[1]SEL-81-105, Recommended Approach to Software Development, S. Eslinger, F. E. McGarry, and G. Page, May 1982

SEL-81-205, Recommended Approach to Software Development, F. E. McGarry, G. Page, S. Eslinger, et al., April 1983

SEL-81-006, Software Engineering Laboratory (SEL) Document Library (DOCLIB) System Description and User's Guide, W. Taylor and W. J. Decker, December 1981

[1]SEL-81-007, Software Engineering Laboratory (SEL) Compendium of Tools, W. J. Decker, E. J. Smith, A. L. Green, et al., February 1981

SEL-81-107, Software Engineering Laboratory (SEL) Compendium of Tools, W. J. Decker, W. A. Taylor, and E. J. Smith, February 1982

SEL-81-008, Cost and Reliability Estimation Models (CAREM) User's Guide, J. F. Cook and E. Edwards, February 1981

SEL-81-009, Software Engineering Laboratory Programmer Workbench Phase 1 Evaluation, W. J. Decker and F. E. McGarry, March 1981

[1]SEL-81-010, Performance and Evaluation of an Independent Software Verification and Integration Process, G. Page and F. E. McGarry, May 1981

SEL-81-110, Evaluation of an Independent Verification and Validation (IV&V) Methodology for Flight Dynamics, G. Page and F. McGarry, December 1983

SEL-81-011, Evaluating Software Development by Analysis of Change Data, D. M. Weiss, November 1981

SEL-81-012, The Rayleigh Curve As a Model for Effort Distribution Over the Life of Medium Scale Software Systems, G. O. Picasso, December 1981

SEL-81-013, Proceedings From the Sixth Annual Software Engineering Workshop, December 1981

SEL-81-014, Automated Collection of Software Engineering Data in the Software Engineering Laboratory (SEL), A. L. Green, W. J. Decker, and F. E. McGarry, September 1981

SEL-82-001, Evaluation of Management Measures of Software Development, G. Page, D. N. Card, and F. E. McGarry, September 1982, vols. 1 and 2

SEL-82-002, FORTRAN Static Source Code Analyzer Program (SAP) System Description, W. A. Taylor and W. J. Decker, August 1982

SEL-82-003, Software Engineering Laboratory (SEL) Data Base Reporting Software User's Guide and System Description, P. Lo, September 1982

SEL-82-004, Collected Software Engineering Papers: Volume 1, July 1982

[1]SEL-82-005, Glossary of Software Engineering Laboratory Terms, M. G. Rohleder, December 1982

SEL-82-105, Glossary of Software Engineering Laboratory Terms, T. A. Babst, F. E. McGarry, and M. G. Rohleder, October 1983

[1]SEL-82-006, Annotated Bibliography of Software Engineering Laboratory (SEL) Literature, D. N. Card, November 1982

SEL-82-106, Annotated Bibliography of Software Engineering Laboratory Literature, D. N. Card, T. A. Babst, and F. E. McGarry, November 1983

SEL-82-007, Proceedings From the Seventh Annual Software Engineering Workshop, December 1982

SEL-82-008, Evaluating Software Development by Analysis of Changes: The Data From the Software Engineering Laboratory, V. R. Basili and D. M. Weiss, December 1982

SEL-83-001, An Approach to Software Cost Estimation, F. E. McGarry, G. Page, D. N. Card, et al., January 1984

SEL-83-002, Measures and Metrics for Software Development, D. N. Card, F. E. McGarry, G. Page, et al., January 1984

SEL-83-003, Collected Software Engineering Papers: Volume II, November 1983

SEL-83-004, SEL Data Base Retrieval System (DARES) User's Guide, T. A. Babst and W. J. Decker, November 1983

SEL-83-005, SEL Data Base Retrieval System (DARES) System Description, P. Lo and W. J. Decker, November 1983

SEL-83-006, Monitoring Software Development Through Dynamic Variables, C. W. Doerflinger, November 1983

SEL-83-007, Proceedings From the Eighth Annual Software Engineering Workshop, November 1983

SEL-RELATED LITERATURE

[2]Agresti, W. W., F. E. McGarry, D. N. Card, et al., "Measuring Software Technology," Program Transformation and Programming Environments. New York: Springer-Verlag, 1984

[3]Bailey, J. W., and V. R. Basili, "A Meta-Model for Software Development Resource Expenditures," Proceedings of the Fifth International Conference on Software Engineering. New York: Computer Societies Press, 1981

Banks, F. K., "Configuration Analysis Tool (CAT) Design," Computer Sciences Corporation, Technical Memorandum, March 1980

[3]Basili, V. R., "Models and Metrics for Software Management and Engineering," ASME Advances in Computer Technology, January 1980, vol. 1

Basili, V. R., "SEL Relationships for Programming Measurement and Estimation," University of Maryland, Technical Memorandum, October 1979

Basili, V. R., Tutorial on Models and Metrics for Software Management and Engineering. New York: Computer Societies Press, 1980 (also designated SEL-80-008)

[3]Basili, V. R., and J. Beane, "Can the Parr Curve Help With Manpower Distribution and Resource Estimation Problems?", Journal of Systems and Software, February 1981, vol. 2, no. 1

[3]Basili, V. R., and K. Freburger, "Programming Measurement and Estimation in the Software Engineering Laboratory," Journal of Systems and Software, February 1981, vol. 2, no. 1

[2]Basili, V. R., and B. T. Perricone, Software Errors and Complexity: An Empirical Investigation, University of Maryland, Technical Report TR-1195, August 1982

[3]Basili, V. R., and T. Phillips, "Evaluating and Comparing Software Metrics in the Software Engineering Laboratory," Proceedings of the ACM SIGMETRICS Symposium/ Workshop: Quality Metrics, March 1981

[2]Basili, V. R., R. W. Selby, and T. Phillips, "Metric Analysis and Data Validation Across FORTRAN Projects," IEEE Transactions on Software Engineering, November 1983

Basili, V. R., and R. Reiter, "Evaluating Automatable Measures for Software Development," Proceedings of the Workshop on Quantitative Software Models for Reliability, Complexity and Cost, October 1979

[2]Basili, V.R., and D. M. Weiss, A Methodology for Collecting Valid Software Engineering Data, University of Maryland, Technical Report TR-1235, December 1982

Basili, V. R., and M. V. Zelkowitz, "Designing a Software Measurement Experiment," Proceedings of the Software Life Cycle Management Workshop, September 1977

[3]Basili, V. R., and M. V. Zelkowitz, "Operation of the Software Engineering Laboratory," Proceedings of the Second Software Life Cycle Management Workshop, August 1978

[3]Basili, V. R., and M. V. Zelkowitz, "Measuring Software Development Characteristics in the Local Environment," Computers and Structures, August 1978, vol. 10

Basili, V. R., and M. V. Zelkowitz, "Analyzing Medium Scale Software Development," Proceedings of the Third International Conference on Software Engineering. New York: Computer Societies Press, 1978

[3]Basili, V. R., and M. V. Zelkowitz, "The Software Engineering Laboratory: Objectives," Proceedings of the Fifteenth Annual Conference on Computer Personnel Research, August 1977

[2]Card, D. N., "Early Estimation of Resource Expenditures and Program Size," Computer Sciences Corporation, Technical Memorandum, June 1982

[2]Card, D. N., "Comparison of Regression Modeling Techniques for Resource Estimation," Computer Sciences Corporation, Technical Memorandum, November 1982

Card, D. N., and V. E. Church, "Analysis Software Requirements for the Data Retrieval System," Computer Sciences Corporation Technical Memorandum, March 1983

Card, D. N., and V. E. Church, "A Plan of Analysis for Software Engineering Laboratory Data," Computer Sciences Corporation Technical Memorandum, March 1983

Card, D. N., and M. G. Rohleder, "Report of Data Expansion Efforts," Computer Sciences Corporation, Technical Memorandum, September 1982

[3]Chen, E., and M. V. Zelkowitz, "Use of Cluster Analysis To Evaluate Software Engineering Methodologies," Proceedings of the Fifth International Conference on Software Engineering. New York: Computer Societies Press, 1981

[2]Doerflinger, C. W., and V. R. Basili, "Monitoring Software Development Through Dynamic Variables," Proceedings of the Seventh International Computer Software and Applications Conference. New York: Computer Societies Press, 1983

Freburger, K., "A Model of the Software Life Cycle" (paper prepared for the University of Maryland, December 1978)

Higher Order Software, Inc., TR-9, A Demonstration of AXES for NAVPAK, M. Hamilton and S. Zeldin, September 1977 (also designated SEL-77-005)

Hislop, G., "Some Tests of Halstead Measures" (paper prepared for the University of Maryland, December 1978)

Lange, S. F., "A Child's Garden of Complexity Measures" (paper prepared for the University of Maryland, December 1978)

McGarry, F. E., G. Page, and R. D. Werking, Software Development History of the Dynamics Explorer (DE) Attitude Ground Support System (AGSS), June 1983

Miller, A. M., "A Survey of Several Reliability Models" (paper prepared for the University of Maryland, December 1978)

National Aeronautics and Space Administration (NASA), NASA Software Research Technology Workshop (proceedings), March 1980

Page, G., "Software Engineering Course Evaluation," Computer Sciences Corporation, Technical Memorandum, December 1977

Parr, F., and D. Weiss, "Concepts Used in the Change Report Form," NASA, Goddard Space Flight Center, Technical Memorandum, May 1978

Reiter, R. W., "The Nature, Organization, Measurement, and Management of Software Complexity" (paper prepared for the University of Maryland, December 1976)

Scheffer, P. A., and C. E. Velez, "GSFC NAVPAK Design Higher Order Languages Study: Addendum," Martin Marietta Corporation, Technical Memorandum, September 1977

Turner, C., and G. Caron, A Comparison of RADC and NASA/SEL Software Development Data, Data and Analysis Center for Software, Special Publication, May 1981

Turner, C., G. Caron, and G. Brement, NASA/SEL Data Compendium, Data and Analysis Center for Software, Special Publication, April 1981

Weiss, D. M., "Error and Change Analysis," Naval Research Laboratory, Technical Memorandum, December 1977

Williamson, I. M., "Resource Model Testing and Information," Naval Research Laboratory, Technical Memorandum, July 1979

[3]Zelkowitz, M. V., "Resource Estimation for Medium Scale Software Projects," Proceedings of the Twelfth Conference on the Interface of Statistics and Computer Science. New York: Computer Societies Press, 1979

[2]Zelkowitz, M. V., "Data Collection and Evaluation for Experimental Computer Science Research," Empirical Foundations for Computer and Information Science (proceedings), November 1982

Zelkowitz, M. V., and V. R. Basili, "Operational Aspects of a Software Measurement Facility," Proceedings of the Software Life Cycle Management Workshop, September 1977

[1]This document superseded by revised document.

[2]This article also appears in SEL-83-003, Collected Software Engineering Papers: Volume II, November 1983.

[3]This article also appears in SEL-82-004, Collected Software Engineering Papers: Volume I, July 1982.

INDEX OF MEASURES