SOFTWARE ENGINEERING LABORATORY SERIES

SEL-81-001

GUIDE TO DATA COLLECTION

SEPTEMBER 1981

National Aeronautics and Space Administration

Goddard Space Flight Center
Greenbelt, Maryland 20771

GUIDE TO DATA COLLECTION

SEPTEMBER 1981

Goddard Space Flight Center

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 submitted as a Computer Sciences Corporation document CSC/TM-81/6102.

The primary contributors to this document include

Victor Church	(Computer Sciences Corporation)
David Card	(Computer Sciences Corporation)
Frank McGarry	(Goddard Space Flight Center)

Other contributors include

Victor Basili	(The University of Maryland)
Jerry Page	(Computer Sciences Corporation)

Single copies of this document can be obtained by writing to

Frank E. McGarry
Code 582.1
NAS/GSFC
Greenbelt, MD 20771

ABSTRACT

Guidelines and recommendations are presented for the collection of software development data. This guide discusses motivation and planning for, and implementation and management of, a data collection effort. Topics covered include types, sources, and availability of data; methods and costs of data collection; types of analyses supported; and warnings and suggestions based on Software Engineering Laboratory (SEL) experiences. This document is intended as a practical guide for software managers and engineers, abstracted and generalized from 5 years of SEL data collection.

TABLE OF CONTENTS

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

Figure

LIST OF TABLES

Table

SECTION 1 - INTRODUCTION

"Software engineering" is a term commonly accepted to describe the way in which software development ought to occur--and the efforts to define and implement such a process. The 1968 NATO conference (at Garmisch, West Germany) popularized the term to serve as exhortation as well as description (Reference 1). Increases in software costs, in requirements for reliability, and in complexity of system solutions involving computers all intensify the need for control and regularization of the software development process. What was arcane art in the early days of electronic computing must become a disciplined and predictable science if it is to meet the demands made of the computer industry.

Transforming the software development process from art to engineering requires a disciplined evaluation of methods and practices, which in turn implies that some aspects of the process are measured. The intent of this document is to describe--and to assist in--the process of extracting relevant and necessary data from the software development activity. All phases of data collection are discussed, starting with reasons for collecting data, through types and sources and costs of data collection, to applications of the data once assembled. Although the specific experiences of the Software Engineering Laboratory (SEL) at the Goddard Space Flight Center (GSFC) (Reference 2) form much of the basis for the guidelines and recommendations herein, this guide is intended not as a historical review, but as a prescription for the future.

This guide is aimed at software developers and managers who would like to be able to assess the value of present and proposed methodologies in their particular environments. Properly collected and analyzed, software development data can aid in identifying sources of problems and errors, in

comparing costs of different sizes and types of projects, in making accurate estimates of production rates and project schedules. This guide is intended to support such efforts by providing a firm foundation for the collection of software development data. By suggesting approaches, by providing a classification of types of data, by identifying costs and pitfalls, this guide should smooth the development of software development data bases and increase the value of software engineering efforts and analyses. The guidelines and recommendations in this document are primarily based on 5 years of direct SEL experience in data collection, and are distilled from discussions at SEL workshops and elsewhere within the software engineering community.

To provide a common terminology and a ready reference, this guide includes a description and classification of software development data (Section 2) and an identification of sources of data (Section 3). Guidelines for estimating the costs of data collection for different types and sources of data, including both direct and indirect costs, as well as suggestions for establishing priorities in a limited-budget environment, are provided in Section 4. The data collection procedure itself--collection by various means, processing the data, management of the data base, validation--is described in Section 5. A brief description of applications of the data is provided in Section 6 to demonstrate and suggest the uses of data in a software engineering setting. Specific recommendations for data collection methods, priorities, and applications are presented in Section 7.

As noted above, the thrust of this document is more "how you can do it" than "how it is done here." Some review of SEL experience and procedures is in order, however, and is provided in the appendixes.

1.1 MOTIVATIONS FOR COLLECTING DATA

Although the software community agrees that problems and shortcomings exist in the way software products are generated, there is often great difficulty in specifically identifying the problems and the means by which one should attempt to improve the process the next time. Each person has his/her own set of criteria for judging a software product (or process) to be successful or unsuccessful. Whether one looks at cost, productivity, reliability, modularity, document size, or other factors, many different parameters are used as evidence of whether a software project has been successful or not. The level and extent to which these measures of software quality are used, or are meaningful, are certainly a long way from being commonly accepted by different software development groups.

Before developers can attempt to improve the software development process and thereby the product, a clear understanding of the strengths and weaknesses of the current mode of operation must be attained. Although it is somewhat unlikely, perhaps the current approach to developing software has no shortcomings and cannot be improved upon. If so, developers should be aware of this fact and should continually reinforce the successful practices. On the other hand, if flaws and problems exist in the approach to software development, they must be identified before they can be corrected. That step is certainly the first in improving the software development process. This is the primary reason that anyone should want to collect software development data. It is the medium by which one can understand strengths and weaknesses, for only then can one aspire to improve the process and the product.

The motivation for collecting software development data can be divided into two categories:

1. Understanding. Unless developers are completely satisfied with each software product generated, there is a need to understand the strengths and weaknesses of the process and the product. The archived information is the only means by which repeating poor practices can be avoided and effective development techniques can be reapplied. As software developers, the questions should always be raised, "How am I doing?" and "How can I do better?"

2. Evaluation. The practices applied to developing software seem to be quite dynamic in that new methodologies, models, and tools are continually becoming available. It is difficult to accept any approach blindly and to believe that it will have the same favorable effects on the software that others may claim. As developers adopt newer and changing approaches, it greatly behooves the evolving process to concurrently measure the effects of particular software development approaches. For example, with the availability of numerous and varied software resource estimation models, it is very problematical to select an approach that is applicable and useful to one's own environment. Before adopting any one of the models, one would most certainly want to evaluate that approach utilizing development data within one's own software area. To evaluate the model, one must have access to the software development data. This certainly applies to the evaluation of any other facet of software engineering.

Although understanding and evaluation are the two key motivators for collecting development data, there are other related reaons. To experiment with varying development technologies, data is needed; to justify changes to development plans (e.g., schedules or expenditures), data is needed; to

provide the basis for hoping to advance the state-of-the-art, data is needed; but the summary fact that calls for the data collection effort is the desire and the need to understand and improve the software development process and product. Without accurate, descriptive data of the software process, these attempts at understanding and improving will be futile.

1.2 HOW TO USE THIS GUIDE

The intent in producing this guide is to suggest answers to the "How do I start?" and "What do I do?" questions following the recognition that the software development process can be improved. The answers are: "Understand what you're doing now" and "Collect relevant data." This introduction will give the reader a high-level overview of the process without going into details. Section 1.4 particularly identifies the data to be collected and from where it is collected. Sections 2 and 3 expand on those topics for reference purposes and may be skimmed or skipped on first reading.

The next question to be asked often is "How much will it cost?"; the reader is directed to Section 4 for this answer. Noting the iterative nature of this entire process (see Figure 1-1), Sections 4, 5, and 6 may be of value in any order. Section 4 addresses "what (and how much)"; Section 5 treats the "how" of data collection; and Section 6 deals with "why." The details of managing and organizing the data are described in Section 5 (and treated elsewhere in depth), along with the basic guidelines for collecting data.

As a reference to the process of data collection, Sections 2, 3, and 5 form a unit. Section 2 describes the types of data in more detail than required in an overview; Section 3 describes where the data is found; and Section 5 addresses how to get it.

The listed references and the related publications in Section 1.5 suggest to the reader what can be done with the data once it is collected.

1.3 ORGANIZING THE COLLECTION OF DATA

Crucial to the success of the overall data collection process is a clear, advance understanding of the local purposes and analyses to be supported by the data ("local" meaning specific to the installation supporting the data collection). The range and amount (and cost) of data to be collected must be directed and bounded by the uses to which the data will be put. Typical goals include

- Quantifying the phasing and staffing patterns of software development projects
- Numerically characterizing the developed software (e.g., lines of code per module, percent comments, complexity measures)
- Identifying major sources of errors (by phase, by activity, by personnel) and most commonly effective methods of detecting and correcting errors
- Comparing methodologies, personnel, management techniques as factors in productivity differences

These and other topics are treated briefly in Section 6 and discussed in detail in other publications (see References).

Data collection must be identified as one element in a cohesive plan for software engineering analysis (Reference 3). The goals of the entire effort must be clearly defined; these goals will be the driving factors in planning and implementing the data collection process.

The first step, then, is the identification of these goals for the local environment. Specific, demonstrable goals should be set (e.g., "Are there quantitative differences

between overall productivity measures of equivalent projects using FORTRAN or PL/1?" or "Do projects that have design walkthroughs have more or fewer problems during acceptance testing?").

Once the goals of the software engineering analysis effort have been determined, the specific data requirements of those analyses must be identified. Sources, procedures, and costs all will affect the selection of data types. Interdependencies must be identified to ensure that all requisite data are targeted for collection (e.g., resource data may be of little value witnout data on staffing patterns). Section 4 covers these topics.

For some types of software engineering analyses, quantitative models have been developed (Reference 4). Where these apply to the specified goals identified, the types of data needed may be spelled out in the descriptions of the models. For other analyses, identification of data items will involve more extensive planning. More detail is provided in Section 6.

Cost factors, model requirements, data dependencies, and data availability may mandate review and revision of the software engineering analysis goals. An iterative process (as shown in Figure 1-1) may be required to resolve questions of data availability, priorities, and budgetary limits. An evaluation of the anticipated return (e.g., more reliable products, more confidence in project estimation) should be performed as a justification for the data collection process. Failure to recognize and secure the level of commitment required may jeopardize the software engineering engineering effort when the costs become apparent before those benefits can be demonstrated.

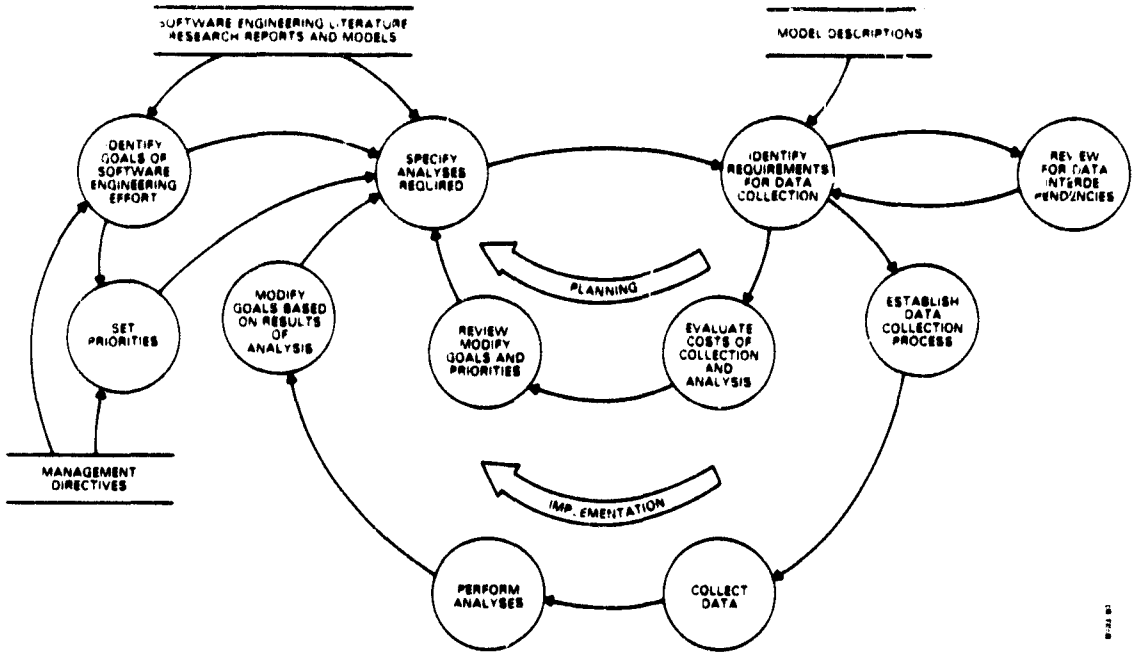

Figure 1-1. Data Collection--Functional Relationships and Iterative Processes

Definition of the data collection procedures to be used is implicit in the cost analysis. Data that is readily available in machine-readable form (e.g., staff charge hours, computer use records) is inherently less expensive than data collected on forms or by interview. The details of the collection procedure remain to be worked out, but the general sources and methods are defined in the early planning stages. Section 5 provides guidelines on the planning and design of the data collection process. Examples of forms used in data collection by the SEL are shown in the appendixes.

Implementation of the system, especially in those aspects that directly impact the software development process (e.g., requiring programmers to fill out forms), will require

planning, public relations, and support from management. A phase-in period must be anticipated, both for the technical personnel and for the data collection procedures.

1.4 OVERVIEW OF SOFTWARE DEVELOPMENT DATA

This subsection provides a high-level overview of the types of data pertinent to a software engineering effort and of the relevant data collection methods. Sections 2 and 3 expand on these topics; this overview is intended to provide an introduction and a common frame of reference and terminology for the sections that follow.

In the context of data collection, software development can be viewed as consisting of five elements: the problem to be solved, the resources needed for solution, the environment in which those resources are applied, the process of applying them, and the products generated. Data must be collected characterizing each of these elements (at least to some elementary level) to support a responsible analysis effort. The data-collection process described in this document will be defined in terms of these five elements, described as follows:

1. Problem data--the problem as described in the requirements specification, constraints (such as space or execution time), stability of the specification, type of changes, and so forth
2. Resource data--such as staff-hours and computer time
3. Environment data--characteristics of the installation which are relatively stable and invariant from project to project
4. Process data--the methodologies, tools, techniques used in developing software

5. Product data--measures and metrics which characterize the software and documentation (size, quality, complexity, etc.)

Each of these classes of data is further categorized by reporting interval as summary or snapshot (i.e., at completion or in process) data, and by level of detail (from project or task level down to detailed component or module data). Section 2 provides the formal definitions and classification scheme.

Four major sources of software development data are described in Section 3. An understanding of the sources and terminology will be useful throughout this document. The four sources of data are as follows:

1. Technical staff (i.e., programmers, designers, analysts, operations and maintenance people)--information available via forms, interviews, activity logs
2. Managers--forms, interviews, personnel records
3. Computerized records--accounting data (machine time, number of runs), configuration control records (e.g., PANVALET), transaction records (if maintained for detailed analysis)
4. Products--source code, documents, design (may require tools such as a source analysis program for data reduction to a usable form)

1.5 RELATED PUBLICATIONS

This subsection provides a list of particularly relevant publications that may aid in the collection of software development data. A more extensive list is provided in the list of references.

1. The Software Engineering Laboratory, V. Basili, M. Zelkowitz, F. McGarry, R. Reiter, W. Truszkowski, U. Md TR77-535, and NASA-NSG-5123. Describes the background, goals, and intended data collection procedures of the SEL as of 1977.

2. Tutorial on Models and Metrics for Software Management and Engineering, V. R. Basili, IEEE, 1980. Describes purposes and methods of software data collection.

3. Software Engineering Laboratory (SEL) Data Base Organization and User's Guide, D. Wyckoff, CSC/SD-81/6011UD1. Provides a detailed description of the data management and dissemination procedures of the SEL.

4. Software Engineering: Concepts and Techniques, P. Naur, B Randell, J. N. Buxton (eds.) Petrocelli/Charter, New York, 1976. A conference report on the 1968 NATO Conference on Software Engineering at Garmisch, West Germany. Contains many seminal papers of the field, still highly relevant.

5. Software Engineering Laboratory (SEL) Data Base Maintenance System (DBAM) User's Guide and System Description, D. Card, CSC/SD-81/6079. Describes the interactive data entry and editing system used by the SEL.

SECTION 2 - CLASSIFICATION OF SOFTWARE DEVELOPMENT DATA

Software engineering data, as the term is used here, encompasses any information that describes the process or products of a software development effort and that is collected to support analyses and evaluations of that effort. Relevant data may pertain to any of a multitude of factors and be characterized by varying measures of objectivity and level of detail. Productive research and analyses at a particular facility will require a broad range of types of software development data to support comparisons and evaluations. This section describes several classes of data that together provide the range of data types required.

The classification scheme presented here is intended to provide a common framework for discussion; other classifications can and have been developed (Reference 3). To be useful, such a scheme must organize the data in some logical relationship to the goals of the research effort. While clear and distinct divisions cannot always be drawn, a classification scheme can aid in the research design by ensuring adequacy of data collection and clarifying data dependencies. The scheme presented here is intended to support research in improving the process and products of software development and to clarify the approach of the SEL to this research. The classes of data identified in this scheme are

- Problem data
- Resource data
- Environment data
- Process data
- Product data

Section 2.1 describes the dimensions of the classification scheme (subject matter, time of collection, level of detail) and relates this classification to the types and levels of

analyses that may be required. Section 2.2 defines the problem data class that describes the initial requirement for software and the changes and constraints involved in the development. Section 2.3 defines the resource data class, which is used to track and summarize the expenditures of staff time and computer time in the solution of the problem. The actual software development activity is characterized by the environment and process data classes (Sections 2.4 and 2.5), which deal with such items as invariant attributes of personnel, management procedures, languages, hardware support, methodologies, and standards. Section 2.6 defines the product data class, which describes the output of the software development activity. Section 2.7 describes change and error data--a composite of product and process data.

2.1 CLASSIFICATION STRUCTURE

For purposes of analysis, it is useful to be able to map classes of data onto elements of the software development model used in the analysis. In its simplest form, the model used by the SEL identifies an input (the problem or requirement), a process (software development methods and techniques), the environment (including types of machine and personnel resources available), an output (the products--software and documentation), and some measure of the resources utilized. Figure 2-1 shows the basic elements of this model.

To some extent, the mapping of data elements onto model elements is determined by the purposes and perspective of the analysts. From a process measurement viewpoint, it may be useful to view resources and problems as input that is processed to output a product. From a different standpoint (e.g., contractural), "resources expended" may be considered a product to be controlled by manipulating the process. In the same vein, change and error data can be considered to be

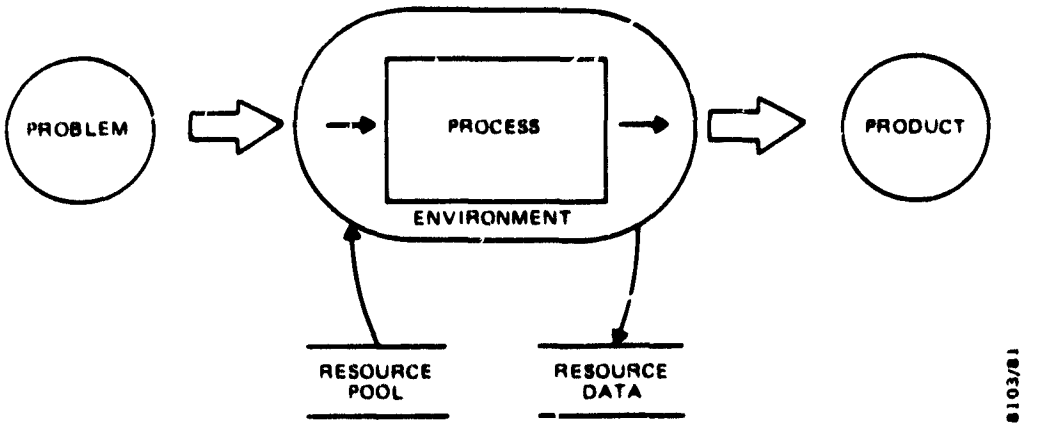

Figure 2-1. Research Model

either a product measure or a process measure, depending on the analyses being performed. Because of this ambiguity, the data classification scheme described here has been made flexible to accommodate a variety of investigations.

A rigorous analysis of some facet of the software development process will require some data on each element of the model--that is, some level of detail on the problem, resources, environment, process, and product. A study of productivity (products/resources) must consider as well the impact of variations in problem and environment and process. The major dimension in this classification scheme is, accordingly, subject class of data.

Two other dimensions (Figure 2-2) are used in classifying data: resolution and project status. Resolution, or level of detail, ranges from the broad brush-strokes of project overview data (executive summaries, etc.) to the precision of module-level and component-level detail. Low-resolution data, such as total lines of code or total staff-hours for a project, is useful for screening and comparing projects. High-resolution data, such as the purposes and results of individual computer runs and changes, provide insight into why a project progressed as it did.

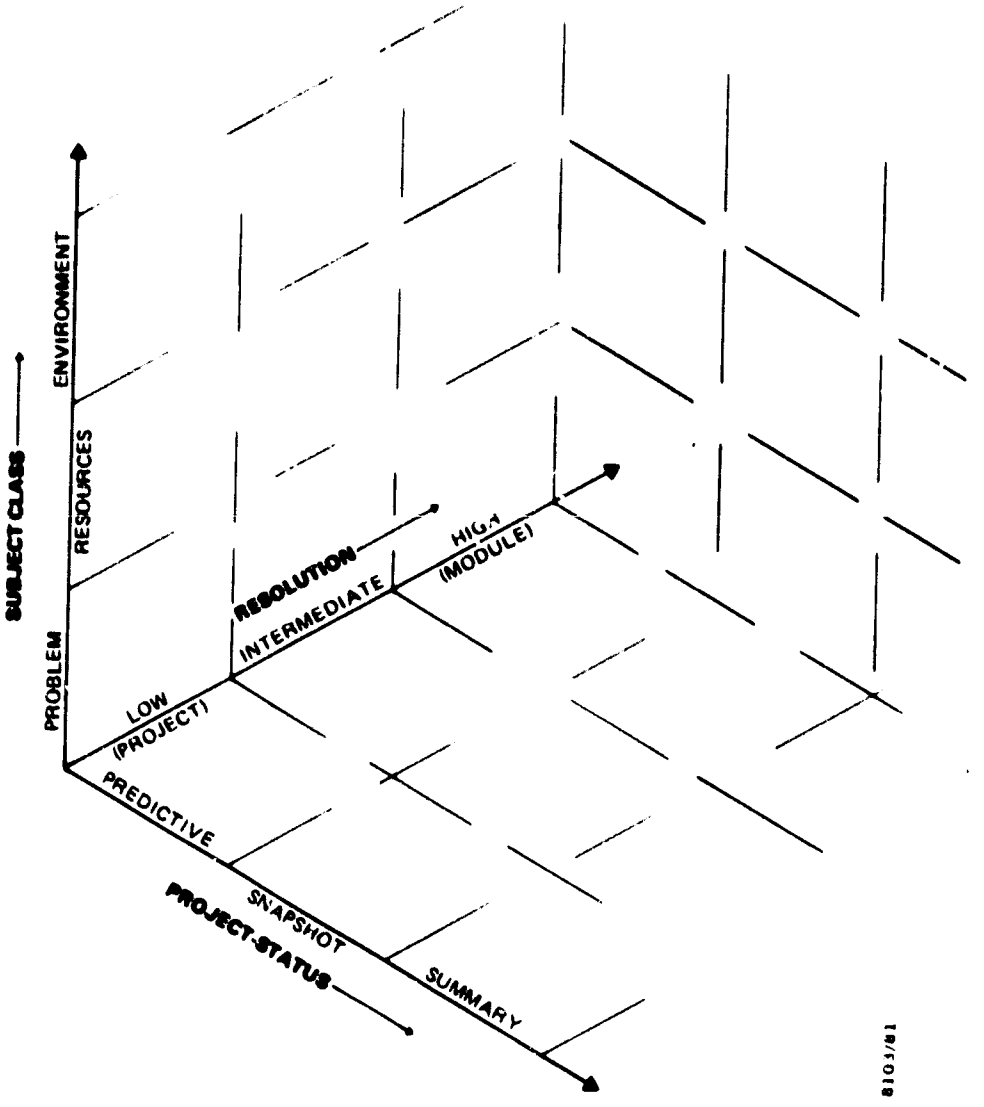

Figure 2-2. Dimensions of Data

project status, the third dimension in Figure 2-2, refers to the stage of the project or component about which the data has been collected. "Predictive" data is available before a project or phase begins, or before a component is fully defined. "Snapshot" data reports status at some intermediate point in the process. "Summary" data characterizes the project at completion of each major stage of development. Snapshot data thus describes the path from start to finish, or on this dimension of data, from prediction to summation.

Table 2-1 briefly shows the classification scheme described above and the sources of data in each class and subclass.

2.2 PROBLEM DATA

The driving force behind the software development effort is the software requirement which, in its original statement and the changes and corrections that inevitably occur, largely defines the scope of work in a development effort. The problem class of data is used to characterize the software requirement. The parameters used in this characterization are listed in Table 2-2. It should be noted that the major attributes of the problem are its size and the stability of the requirement. The latter attribute is particularly time-dependent (i.e., snapshot data) because the impact of changes depends greatly on the timing of those changes.

Problem data may also include constraints such as space or execution speed of the software or externally imposed deadlines (such as defined by a spacecraft launch date).

2.3 RESOURCE DATA

Life-cycle studies and detailed project performance analysis require substantially more detailed information than is provided by project-level summary data. Characterizations of actual effort expended on each phase or component of a system generally require that timely, detailed data be collected

Table 2-1. Classification Scheme and Sources of Data

SUBJECT	DETAIL LEVEL		
	LOW RESOLUTION (e.g., PROJECT)	INTERMEDIATE (e.g., PHASE, SUBSYSTEM)	HIGH RESOLUTION (e.g., COMPONENT, MODULE)
PROBLEM	REQUIREMENTS AND CONSTRAINTS DOCUMENT, ECO RECORD	PRELIMINARY DESIGN SPECIFICATION SYSTEM DESCRIPTION	
RESOURCES	BUDGET FIGURES AND ACCOUNTING RECORDS (STAFF AND COMPUTER)	TECHNICAL STAFF, ACCOUNTING RECORDS, MANAGEMENT REPORTS	TECHNICAL STAFF
ENVIRONMENT	PROJECT REPORTS, SUBJECTIVE MANAGEMENT DATA, INTERVIEWS	SUBJECTIVE MANAGEMENT DATA	
PROCESS	SUBJECTIVE MANAGEMENT DATA	SUBJECTIVE EVALUATIONS, TECHNICAL STAFF	SUBJECTIVE EVALUATIONS, TECHNICAL STAFF, AUTOMATED DATA COLLECTION
PRODUCT	AUTOMATED ANALYSIS, SUBJECTIVE MANAGEMENT DATA, TEST REPORTS	AUTOMATED ANALYSIS, SUBJECTIVE EVALUATIONS, TEST REPORTS	AUTOMATED ANALYSIS, TECHNICAL STAFF

TIME DEPENDENCY EACH SUBCLASS MAY CONTAIN SUMMARY (TERMINAL, COMPLETION) DATA AND SNAPSHOT (HISTORICAL, EVOLUTIONARY) DATA

8101/81

Table 2-2. Problem Data Parameters

- Problem Statement
  - Number of specifications
  - Clarity of specification
  - Nature of specification (e.g., machine processable)
- Problem Stability
  - Number and timing of changes to specifications
  - Impact of changes (cost, perturbation of product)
  - Nature of changes (e.g., correction, enhancement, cosmetic)
- Problem Characteristics
  - Magnitude of problem
  - Perceived reliability of specifications
  - Complexity
  - Similarity to previous problems
  - Constraints (calendar time, machine resources, interface to existing software)
- Product Delivery Requirements
  - Formality of documentation (especially transition documents)
  - Reporting and review procedures
  - Impact of software development data collection

from development personnel and/or from the machine accounting systems. Required data are

- Personnel resources applied
  - Management
  - Technical
  - Support (clerical, publications, etc.)
- Machine resources required
  - Computer time used
  - Terminal use (or other access records)
  - Data storage (e.g., disk utilization)

Summary data may be used to perform comparisons and evaluations of projects; detailed resource data is necessary to understand the differences and to fit individual tasks to models that can be used predictively. Personnel resources should include information on the type of effort and component (if any) supported. Machine resources should be collected in accordance with whatever cost or chargeback algorithms are in use, with additional information to identify user, type of access, purpose (e.g., of a test run), and component(s) involved.

2.4 ENVIRONMENT DATA

With respect to software development, the environment consists of the relatively invariant factors of staff experience and ability, computer system availability, management procedures, and similar attributes of a software development facility. Other factors may also be included, depending on the particular installation. In the Flight Dynamics area at GSFC, for example, the programming language--FORTRAN--and the graphic executive support system (GESS) are part of the environment. The distinction between environmental data (which characterizes longer term tendencies, factors, and

attributes) and process data (which is concerned with specific project-related tools and methodologies) is highly installation-dependent.

The factors which are treated as environment data include

- Computer language(s) used
- Staff competence
- Staff experience with typical problems
- Staff experience with host, target computers
- Stability of software environment
- Availability of machine resources
- Stability of machine resources
- Staffing patterns and team organization
- Management competence
- Management experience with typical problems
- Support facilities (e.g., librarians, technical publications expertise)

2.5 PROCESS DATA

The specific tools and methodologies that may be applied on a project-by-project basis are described by process data. For example, to test the "chief programmer/team" methodology, a single project may be organized in this manner. Because this procedure may be imposed on one project and removed on the next, the chief programmer/team technique is a process factor. On the other hand, a decision to switch to a new computer system, operating system, or language usually cannot be as easily reversed; for better or worse, the change will probably impact a number of projects. Such a change would be treated as environment data.

From the standpoint of software engineering analyses, environmental data describes those factors which must be canceled out of the results. (That is, the effects of the environment must be identified, accounted for, and disregarded so that other effects can be analyzed.) Process data describes the tools, methodologies, and techniques which are being evaluated and undergoing experimentation. Table 2-3 lists some of these factors.

2.6 PRODUCT DATA

A tremendous amount of objective data can be derived from the products of the development process, particularly by analysis of the actual software. Elaborate and ambitious models of software development have been based on such analyses (e.g., Halstead's "software science," Reference 5). The development products that can be so analyzed include the software source code, design and specification documents, process documentation (e.g., design notebooks), and product documentation (user's guide, system description), as listed in Table 2-4.

An important characteristic of product data is that it is relatively nonvolatile. Time-based information is required, of course, such as growth histories and implementation patterns, but significant amounts of data can be acquired as late as the end of a project. Change and error data, in contrast, are very dependent on timely recordation.

2.7 CHANGE AND ERROR DATA

Although change and error data does not form a distinct category (in terms of the proposed model of software development), it is sufficiently important to software engineering (reliability) to be described separately here. From the standpoint of operations and maintenance, changes and errors are a product measure. During development, change and error data may also describe the process.

Table 2-3. Process Data Factors (Representative)

Graphic representations and expressions

- Flowcharts
- HIPO (hierarchy-input-process-output) charts
- Data flow diagrams
- Hierarchy diagrams (baselines, tree charts)
- System verification diagrams (SVDs)

Development methodologies

- Top-down design (stepwise refinement)
- Top-down development (stubs, drivers)
- Structured programming
- Standards and protocols
- Use of librarian
- Chief programmer/team approach
- Unstructured ("egoless") team

Quality assurance mechanisms

- Design, code reading
- Design, code walkthroughs
- Traceability analyzers
- Standards compliance auditors
- Verification, validation teams

Configuration control and management

- Source code control library system
- Design, development notebooks
- Change reporting and control protocol
- Milestone charts
- Configuration reporting tools

Table 2-4. Product-Derived Data

Source Code

- Number of modules
- Component coupling and connectivity
- Component size
  - Various measures of lines of code: old versus new, developed versus delivered, with and without comments, executable and specification statements
  - Memory requirements: words of machine code
- Complexity of code (e.g., McCabe, Page measures)
- Halstead's "software science" metrics
- Execution characteristics

Specifications and Requirements (if automated)

- Traceability analysis
- Change records (e.g., changes to specifications)
- Complexity measures

Documentation

- Page counts
  - Text
  - Figures
  - Copies of listings
- Change history (from notebooks)

Users of an interactive develoment system, with the instantaneous turnaround and immediate response of a terminal-oriented environment, may use different testing strategies than would a one-run-per-day batch user. The change history is different because the interactive user can afford to make one correction per pass, whereas the batch user must attempt to correct all errors for each new submission. Change and error data can be a characteristic of the process, or even of the environment.

Prevention and correction of errors and adaptability to change are central to the efforts of software engineering. The analyses that can measure the effectiveness of various methodologies in these respects require substantial amounts of data on the occurrence and processing of changes and (as a subset) on detection of errors. This type of data is of sufficient importance and volume to justify its independent classification, even though there may be overlap with resource data.

The definition of changes (and errors) should be related to the configuration management tools and procedures employed in the development effort. At each stage of the development cycle, changes would be recorded for all items that had been accepted for control. The SEL uses the software module as the basis for defining changes, and relates code changes to modifications in specification or requirements (as applicable). Change reporting and control is important throughout a software development task; the emphasis and depth of coverage will depend on the specific goals of the research.

Many of the methodologies and techniques being investigated in the field of software engineering are directed toward maintaining flexibility and integrity of a development effort in the face of changes (whether internal or external). Top-down design, iterative refinement, structured

programming, specification languages--all are partly motivated by this concern. To measure the impact and frequency of changes, data should be collected as changes are identified and should include type of change, source, means and resources to implement, and magnitude and impact of the change.

For the purposes of software engineering analysis, errors are perhaps the most interesting type of change because of the desire to limit error occurrence by applying appropriate techniques. Methods of detecting, preventing, and correcting errors are of concern to software engineering. A variety of models of error occurrence have been devised to aid in predicting the number of errors, or errors remaining, in a software system. Data to support the use of these models and analyses is included within this class. Information should be collected on a timely basis to identify the source of errors, the means used for error detection, and the type of error. As with other change data, records of resources required and impact of the error should be collected.

SECTION 3 - SOURCES OF SOFTWARE ENGINEERING DATA

Sources, availability, and methods of collection of software development data are of major concern to the structuring of a data collection effort. Reliability, accuracy, consistency and completeness all have impact on the types of analyses that can be performed, and all are affected by the source of the data. Particularly important is the availability of the same (or closely related) data from disparate sources, for purposes of cross-checking. This section describes the primary sources of software development data, the types of data available from each source, and, briefly, the means of collection from each source. (More on the last topic is found in Section 4.2.)

The primary sources of software development data are

- Developers (personnel)
  - Technical staff: analysts, designers, programmers
  - Managers
- Computerized records
  - Routine accounting records
  - Special-purpose activity monitoring
- Products: source code, documentation, specifications

3.1 TECHNICAL STAFF

As used here, the term "technical staff" refers to all personnel who contribute directly to the project products. This catchall term is used to distinguish productive effort from managerial activities and includes the designers, programmers, quality assurance (QA) staff, test teams, and support personnel. To the extent that managers contribute to

production, they also are included. (See also Section 3.2.) Because the technical staff is the major source of nonhardware resources that go into a project, they must also supply much of the detailed information on the process of software development.

Although automated data collection procedures should be used when possible, more intrusive methods--forms, interviews, activity logs--will commonly be required. The costs and impacts of various procedures are discussed in Section 5.

Table 3-1 lists the major data items that may be collected from the technical staff. The major classes of data are resource data (hours expended, computer usage), product data (including subjective data, such as perceived complexity, which is not available elsewhere), and change/error data.

3.2 PROJECT MANAGEMENT

Although the technical staff is the most valuable source of data concerning the detailed evolution of a project, more comprehensive and global project data should be obtained from project management. Administrative data (such as the amount of time actually charged to a project) can be used to cross-check (and fill gaps in) the more detailed resource data supplied by the technical staff. Being less absorbed in the details of a software development effort, managers are better able to provide evaluations of the project as a whole. Sensitive data such as experience and competence of the technical staff may not be available below the managerial level. Data on support facilities that may be shared among projects (e.g., typing support) is also available from project management.

Table 3-2 details the types of software development data for which project management is a source. Note that, in many cases, these data can provide verification of data provided by the technical staff.

Table 3-1. Data From Technical Staff

Staff Hours Expended (resource data)

- By activity
  - Design, code, test, travel, writing, etc.
- By phase
  - Requirements analysis, design, code and test, integration, acceptance
- By component or subsystem

Component Descriptions (product data)

- Subjective measures and predictions
  - Type of software (algorithmic, I/O,....)
  - Complexity, difficulty
  - Characteristics of specifications, design
- Status, size at each phase
- Relationship to other components
- Constraints (memory, execution speed)
- Language used

Computer Usage Records

- Purpose and status of each run or session
- Components involved
- Characteristics of computer use (batch, interactive....)

Change Error Data

- Origin of change
- Components affected
- Source of error, how (when) discovered
- Time required to effect change

Table 3-2. Data From Managers

Staffing Patterns and Characteristics

- Number and phasing of personnel
- Project organizational structure
- Quality and level of experience

Product Characteristics

- Quality of products
  - Reliability, maintainability
  - Efficiency, effectiveness
- Degree of structure
- Readability (especially documentation)
- Compliance with specifications, constraints

Development (Methodology) Data

- Development environment
  - Type of computer support
  - Congeniality
- Manageability (visibility)
- Adherence to standards or guidelines
  - Adequacy of standards or guidelines
  - Compliance and enforcement
- Methodologies used (degree used)

3.3 COMPUTERIZED RECORDS

Most computer installations provide a means of automatically collecting and recording details on the use of the facilities. These records typically include information on each instance of use (job submission, interactive session) identifying the user, the time, and the resources used. Although typically collected and analyzed for chargeback and performance evaluation purposes, these records can also provide valuable data on software development activities. In addition to project-level resource (cost) data, these records can provide a cross-check on programmer-supplied computer use data. The specific data available will depend on the type of accounting in use at a facility.

A different type of accounting information is provided by "librarian" systems (e.g., PANVALET). These systems can identify source code and library utilization by project and in some cases by component. Properly used, these systems can provide data on project size, growth history, and errors and changes over time.

Table 3-3 lists the types of information that are collected automatically (for other purposes) and can be used in software engineering analyses. Principal motivations in using these data are their low cost and their availability (although some effort will be required in the data reduction effort). Also of importance are the consistency and reliability of the data so collected and the negligible impact on the software development process.

In addition to accounting and library system records, transaction records may be collected specifically to aid in software development analysis. Such data collection requires an initial investment (for the development of software to collect such data) and continuing overhead costs, but may be

Table 3-3. Computerized Recordkeeping

Access and Use Records

- Computer loading by time and phase
  - CPU, memory
  - I/O, mass storage utilization
- Use characteristics
  - Frequency of use by person over time
  - Type of use (compilations, editing, executions)
- Printout volume

Librarian Accounting

- Size and number of modules
- Change history
- Growth history

Automatic Collection of Data

- Computer use data
  - Session purpose and status
  - Patterns of edits, compiles, tests
  - Modules involved in each access
- Product data
  - Change and growth history
  - Design evolution (PDL, prologs, baselines)
- Test history, status, results

capable of reducing the need for form-filling or replacing some data collection forms entirely. At present, the SEL has only begun to investigate costs and potential. Table 3-3 includes some of the types of data which are thought particularly amenable to this type of data collection.

The high cost of using data collection forms and the limits to data collection using these forms makes it desirable to pursue such alternatives, but no empirical recommendations can be made at this time.

3.4 DEVELOPMENT PRODUCTS

Many of the analyses that can be performed on software development data require detailed information on the products of the development process, including the documents and the deliverable software. This information should be derived from the products themselves to ensure accuracy and reliability. For the deliverable software, a software tool is extremely valuable in extracting such data. Manual methods may be required for special cases (e.g., segments of assembler code in a FORTRAN system) or non-machine-readable products (such as a design document). Table 3-4 identifies data elements derivable from the products of the development effort.

The most commonly used program-size metric is "lines of code," but there are several interpretations of this term. Comparability and comprehension both can be enhanced by using a source code analyzer program to compute line and statement statistics on project and component levels. These statistics typically include number of source lines, lines with comments, executable statements, and detailed statement type statistics.

Specific software models (e.g., McCabe's complexity measure, Halstead's metrics) (References 6, 7) will require additional

Table 3-4. Product-Derived Data

Source Code Analysis

- Size (various measures)
- Statement type distribution
- Module classification
- Complexity analysis
- Specific models
  - McCabe
  - Halstead

Document Analysis

- Page counts by type of page, by volume
- Count of changes to specifications or requirements
- Type of specification, design

Compliance With Constraints

- Execution time
- Memory loading
  - Loader maps
  - Dynamic measurements

detailed analysis. Because the actual product is used, the reliability of the data is maximized.

Dynamic analyses are more difficult to obtain, although some computer systems are able to collect data on memory, device, and CPU utilization to some level of detail. To measure compliance with constraints such as execution time or memory allocation, specific data collection procedures may be required. System load maps or execution analyzers may be usable or at least provide some guidance. In most cases, subjective management-level evaluations of performance and compliance will be adequate for the purposes of software development data analysis.

Document analysis is less easily automated, although the use of word processing may facilitate data collection and static analysis. Some normalization process may be required to provide comparability among different styles of document: pages of sample printout require much less preparation than text; complex figures require more. Collection of descriptive data is required to support staffing level analysis or prediction of costs for future products.

SECTION 4 - COSTS AND PRIORITIES

Identification and prediction of costs are essential to the planning of any project. Because collection of software development data is inherently a long-term activity, cost identification is especially important to ensure that adequate resources will be budgeted. This section identifies the types of costs incurred in collecting software development data and suggests the magnitudes of those costs based on SEL experiences. Because the planning process will undoubtedly involve tradeoffs, cost comparisons (Section 4.2) and data dependencies (Section 4.3) are also discussed. The tradeoffs made at a particular facility will depend partly on these cost factors and partly on the goals of the particular software data analysis effort. Section 4.4 discusses priorities and recommendations for data collection in terms of those goals.

This section provides practical guidance for planning a data collection effort by suggesting how to maximize the return for a given level of investment. A successful software engineering research effort requires only--and all of--those data needed to support the desired analyses. This section supports that effort.

4.1 DATA COLLECTION COSTS

There are four primary sources of costs to collecting software development data:

- Impact on monitored tasks
- Processing (verifying, storing, disseminating) data
- Development and maintenance of support software
- Analysis of data

Table 4-1 illustrates these factors and the magnitude of the associated cost. Costs are normalized by relating them to the magnitude of the projects being monitored.

Table 4-1. Measuring Software Technology Costs (As a Percentage of the Tasks Being Measured)

TASK	SEL EXPERIENCES	GOAL
OVERHEAD TO TASKS (DEVELOPMENT PROJECTS) FORMS MEETINGS TRAINING INTERVIEWS COST OF USING TOOLS	5 TO 15%	5%
DATA PROCESSING COLLECTING/VALIDATING FORMS ORGANIZING DATA ENCODING INFORMATION DATA MANAGEMENT AND REPORTING	10-12%	6 TO 8%
SUPPORT SOFTWARE DATA BASE SOFTWARE CODE ANALYZERS REPORT GENERATORS STAFF TRAINING	6 MAN-YEARS, THEN 1 MAN-YEAR PER YEAR	½ MAN-YEAR PER YEAR
ANALYSIS OF INFORMATION MEASURING METHODOLOGIES DESIGNING EXPERIMENTS DESIGNING ANALYSIS TOOLS DEFINING MEASURES	15 TO 25%	10%

8103-87

4.1.1 TASK OVERHEAD

The data collection process adversely affects the monitored development projects (tasks) in several ways, depending on the types of data and the methods employed. Data collection forms have the greatest impact, particularly when used to collect component-level data. Forms used by the SEL (see Appendix B) are filled out by the technical staff on a weekly basis (to monitor resources) and for each change and each computer run. Additional time is spent on project-level quality assurance for forms data and in filling out forms for resource, component, and project summary data. A startup cost is also incurred in training development personnel to use the forms (to ensure consistency).

Task overhead also includes time spent in meetings to define subjective project-level data and in interviews to collect background data on changes, errors, and procedures.

Additional costs that may be less easily identified are due to the use of data collection tools (such as the PANVALET librarian system), which may be used more extensively because a project is being monitored. No attempt has been made by the SEL to isolate these costs. More sophisticated automatic data collection procedures have not been implemented by the SEL.

On tasks monitored by the SEL, overhead costs chargeable to the development tasks ranged from 5 to 15 percent of the total task cost. It should be noted that the SEL is collecting a very broad spectrum of data; nevertheless, it seems unlikely that the cost could be held under 5 percent, even with a less ambitious data collection effort.

4.1.2 DATA PROCESSING COST

A continuing staff effort will be required to perform the collection, quality assurance, validation, data entry, and reporting functions. Here again, data collection forms contribute most significantly to the cost, while other types of data processing (such as source code analysis) contribute to a lesser degree. Data management costs (error correction, status monitoring and reporting) must also be considered. The experience of the SEL has been that the cost of processing and managing the data amounts to 10 to 12 percent of development task costs. By simplifying the forms and streamlining the process, this cost can be cut in half.

4.1.3 SUPPORT SOFTWARE

An initial investment in support software is required in those cases where data management involves a computerized data base. In any case, training and documentation costs will be incurred during the startup phase. Because of the iterative nature of the data collection and analysis process (Figure 1-1), changes are likely to be required as the effort progresses.

The SEL chose to build a data management system tailored to our requirements as they evolved. Six staff-years of effort were required to develop the system to its present state (References 2, 11, 12). Maintenance of the software--primarily enhancements to support new data types--requires 1 staff-year per year. As the data management system and the collection process become more stable, this is expected to approach 6 staff-months per year.

4.1.4 ANALYSIS COSTS

The cost of analysis will depend on the availability of easily adaptable software for statistical processing and reporting. This cost element is extremely sensitive to the

types of analysis required and to the level of detail of the research. The SEL, which is investigating a very broad range of concepts and factors, has experienced analysis costs that amount to 15 to 25 percent of development task costs. This figure probably represents an upper bound to this cost factor. The SEL expects to limit this to 10 percent in the future by eliminating unprofitable studies and avoiding dead ends. An effort of more limited scope would require a commensurately smaller investment.

4.2 COST COMPARISONS

For each class of data or data source, both startup and continuing costs must be considered in making comparisons. The initial investment involved in building a source code analyzer may be large, but the cost of use is quite small. Conversely, the cost of processing data forms remains high as long as data are being collected. Figure 4-1 diagrams the relative startup and continuing costs of data collection.

4.3 DATA DEPENDENCIES

The data collection process is intended to support analytic efforts to improve the software development process. It is important, therefore, to collect all the types of data needed for a particular analysis. Some of the required pieces of data, however, may not be directly obtainable; the problem is more complex when the needed items must themselves be derived from other data types before they are usable. This section focuses on some of the interdependencies of such derivative data. The SEL is currently investigating (through factor and cluster analyses) consistent formulations for such hard-to-quantify factors as quality and and maintainability. For the present, some simpler dependencies will be exposed.

Productivity requires not only the obvious lines of code and staff-hours but also the life-cycle phases included in the

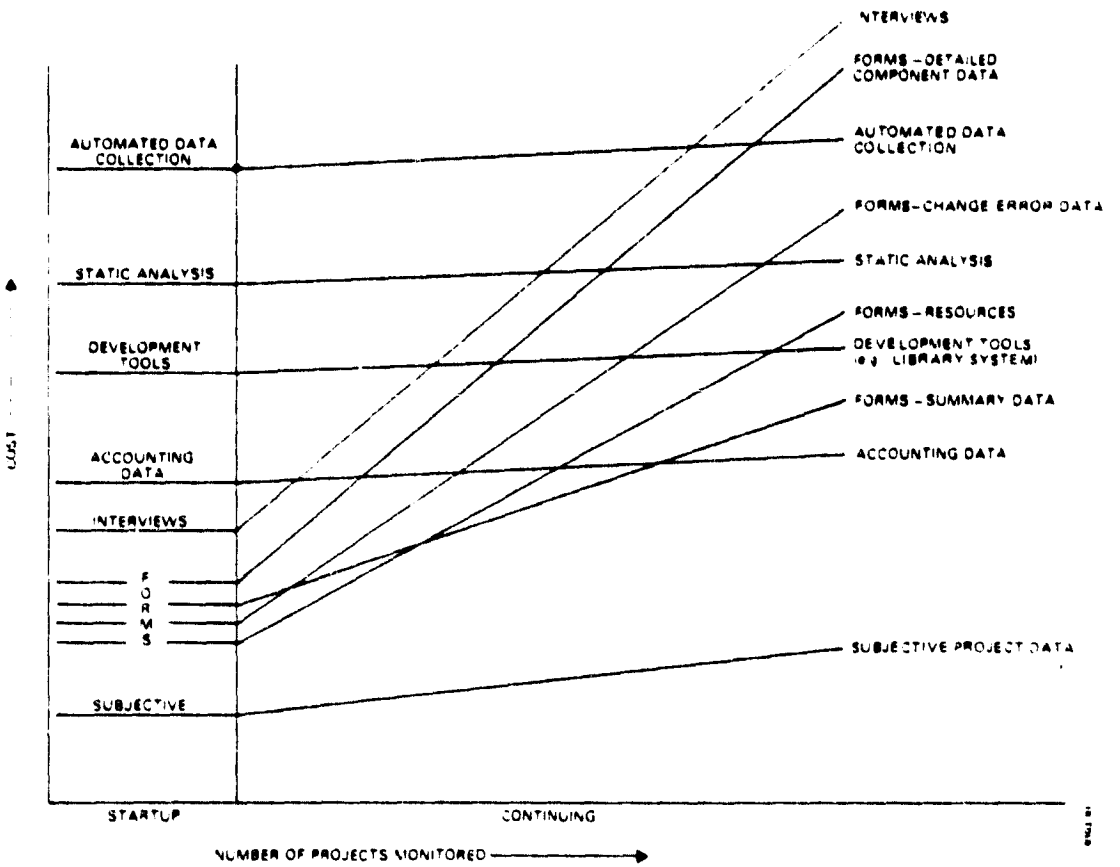

Figure 4-1. Comparative Data Collection Costs

calculation. Comparison of coding-only productivity with full-development productivity does not reveal anything.

Similarly, change data can be compared only if the data are normalized to a specific life cycle period. This is particularly important when projects with different methodologies are compared.

The most glaring data dependency concerns the experience level of the technical personnel. Current data indicates that this factor can swamp other considerations. Some means (typically, data supplied by project managers) must be found to normalize project data for this factor before meaningful comparisons between projects can be made.

4.4 PRIORITIES OF THE DATA COLLECTION

Obviously the type of data and the detail of data to be collected depend on the objectives of the efforts as well as the extent of resources available to support the collection process. If one is merely interested in studying or developing resource estimation models, the heaviest and possibly the complete emphasis would be placed on collecting detailed data representing the resource expenditures for a project on a daily or a weekly basis. In this case, one naturally would not be concerned with collecting detailed change and error data.

In this section, we will attempt to generalize the relative importance or significance of information that could be extracted from a software development project. These generalities are based on the extensive experiences of the SEL during the past 5 or 6 years. It is assumed that the person interested in collecting the data does not have a singular area of study in mind (such as software errors only), but that the reader has more general objectives as stated in Section 1.1: first, to gain a clearer understanding of the software development process in the particular environment

and second, to support efforts when attempting to make a rational judgment as to the methodologies and approaches to be used in developing software in future projects.

In outlining this priority schedule, we refer to the classification scheme described in Section 2. Here, there are five classes of data that may be collected (process, product, project, resources, and error data); and within each type of data, there is a varying level of detail that one may request. These levels of information are summarized in Table 4-2.

Obviously, some of the information defined is nearly useless without some of the other information; for instance, it does little good to know how much we spend on building a product if we know nothing about the size or characteristics of the product under consideration.

Based on the experiences of the SEL, the following priority scheme is suggested to anyone pursuing the general task of collecting software data for studying software development strategies, models, or tools. The derived priorities are based on relative usefulness and difficulty in collecting. The list is ordered from the most important or highest priority (1) to the lowest priority (10):

1. Level 1 of Resource Data. This covers the man-hours expended on the project, as well as the computer usage and general support hours such as technical publications hours, ODC technical hours, and librarian support time. This information is critical in evaluating the total cost of a project and the general profile or model of how resources were consumed. The data has been widely used in helping to evaluate and build models for future cost and resource estimation.

2. Level 1 of Product Data. To support even the most elementary analysis of any software project, it is mandatory

Table 4-2. Levels of Detail in Gathering Various Types of Data

Class of Data	Level-1 Detail	Level-2 Detail
Process	General description of requirement	Detailed characteristics of methodologies applied
	Standards used	Description of each phase of the life cycle
	Tools applied	
	Team organization	
Project	Phase dates	Subjective quality measures of project performance
	Development machine	Staffing details
	Development language	Environmental perturbations
	Level of staffing	
Resources	Weekly manpower expenditure on project	Manpower by component by phase
	Computer usage by week	Computer usage by run
Change	Error information as discovered and fixed	Change information
		Change history
Product	Project size Number of lines of code	Individual component characteristics
	Number of modules	Growth history
	Number of new lines	Subjective rating of project
	Document size	

that the characteristics of the product be recorded. This basic information includes such things as lines of code, number of modules, size of documents, and amount of new code versus amount of reused code. This data is not overly difficult to capture and may be extracted once at the end of the project, but it is required by nearly all meaningful analyses that could be performed on a software project.

3. Level 1 of Change/Error Data. The two greatest concerns of building a software product relate to cost and reliability. Nearly all measures of quality are based on these two factors. The basic error data help to characterize the error-proneness and the reliability of the product; this information complements the information collected that pertains to resource expenditures to provide the basis for characterizing the cost and the reliability. The most important information included here consists of error type, date that the error is found, cause of the error, and level of effort required to correct the error.

4. Level 1 of Project Data. To compare characteristics or profiles of different projects, it is quite important to record the general project characteristics including phase dates, staffing characteristics, type of software being developed, and the manager's general view of the development effort.

5. Level 1 of Process Data. If one hopes to measure or evaluate the effectiveness of particular approaches to developing a software product, one must be aware of the development characteristics or approaches (methodologies and techniques) used during the development process. This first level of process data must describe such characteristics as team structures, standards followed, testing strategies, documentation requirements, approaches to configuration

control, and methodologies utilized. This information can be captured quite easily on this general high level.

6. Level 2 of Resource Data. This is the first of the more detailed level of data that should be collected from the software project. This level includes the weekly detailed manpower expended on each component (subroutine, function, and so on) as well as the type of effort put forth on each component (such as designing, coding, or testing). This level of detail allows one to determine such things as the amount of design effort put forth versus the amount of code effort. It also determines the relative amount of time required for the development of each of the system components.

The Level-2 resource data should also include the detailed usage (by run) of the support computer facility. Here, one captures the reasons that runs were made and the general profile of run results (such as number of successes or failures).

7. Level 2 of Project Data. This information consists of the subjective information describing project characteristics as viewed by knowledgeable managers. Here, one describes such characteristics as the quality of the product.

8. Level 2 of Process Data. This information details the models, tools, and methodologies used in developing the product. Each phase of the software life cycle must be characterized with some selected rating for each of the applied methodologies or tools. As opposed to the general description (for Level 1) of the environment and basic development philosophy, here each of the detailed methodologies (as listed in Section 2.2) must be itemized. This information is quite difficult to normalize and is quite vulnerable to bias and outright error.

9. Level 2 of Product Data. Once a general description of the software product is attained (size, amount of documentation, and so on), one should then attempt to characterize on the component level (such as size and complexity of each component). This information is generally obtained at the end of a project, but the application of this information has been found to be quite academic to date.

In addition, one could attempt to capture, for each component, the estimated characteristics of the component before it is developed and again after development is completed. This information should provide insight into which types of components we can estimate and which types of components are most difficult to characterize until they are completely developed. In the SEL, this particular information has been found to be relatively expensive to collect and relatively difficult to utilize effectively.

10. Level 2 of Change/Error Data. Once all of the error characteristics have been provided, the detailed level of data for the change/error information would include detailed descriptions and histories of changes made to the software. This data must be captured each time a modification is made to design, specification, or code. This data has been found to be quite difficult and expensive to retrieve, and the useful application of it to date seems somewhat limited.

SECTION 5 - DATA COLLECTION PROCEDURES

The heart of any software engineering research program is data collection: the continuing process of collecting, validating, preparing, and furnishing the data required for the intended analyses. The major shortcoming of most published models, predictions, and hypotheses in the software engineering field is the lack of reliable data to provide validation. Only with a rigorous, aggressive program of data collection can software engineering efforts provide a demonstrable payoff to a software development organization. Without substantiation by way of data interpretation, all of the pronouncements and pontifications of software engineering theorists are merely unsupported opinion.

The cost and complexity of this collection process is largely responsible for the paucity of data. Clearly identified goals, careful planning, and systematic implementation are essential, along with adequate monitoring to ensure that data collection goals are met. This requires a high-level commitment of support for the activity, and implies the establishment of some central group or organization with long-term responsibility and resources sufficient for the task.

This section describes the actual data collection procedures that are necessary to support practical and useful software engineering research. Planning for data collection is discussed in Section 5.1. Section 5.2 describes the design of the process. The mechanics of data collection, for each of the sources of data identified in Section 3, are described in Section 5.3. Section 5.4 deals with the management of the data from collection through availability for research. Costs and priorities of data collection (a major concern in real-world endeavors) are discussed in Section 4.

5.1 PLANNING OVERVIEW

The planning process for data collection, as noted in Figure 1-1, includes the following:

- Define the goals of the effort ("identify productive methodologies"...)
- Identify the analyses required to achieve those goals
- Determine what data are needed to perform those analyses
- Determine where the data are, how to collect them, how much they cost
- Specify priorities--what to collect first, what to defer, what to ignore

This procedure, driven by availability of data and available resources, should produce a list of what data are to be collected and whence to collect it.

The initial steps in planning for data collection consist of defining the goals and requirements of the software engineering effort. Because the types of analyses desired may require specific classes and level of detail of data, some focusing of effort should be performed early in the research effort. Some published models, for example, have precise and detailed input data requirements. A high-level description of the types of analyses supportable through these efforts is given in Section 6; an in-depth discussion of the potential of software engineering is beyond the scope of this document. Once the specific goals of the data collection activity have been defined, the mechanics of data collection and management can be planned in detail.

Planning the actual data collection is straightforward once the data requirements and availability have been defined.

Although a number of different functions can be defined, the actual implementation is not significantly different than managing any data base function. Details on data management are provided in Reference 13.

The major activities in data collection are shown in Figure 5-1. The planning for these activities must identify and define responsibility for:

- Implementation
- Data collection and support
- Data management
- Project management

These responsibilities are detailed in Table 5-1.

Implementation functions include design of procedures, forms, data flows, and protocols and implementation of the data base; these are essentially one-time startup activities. Data collection and support functions include supervision and monitoring of the data collection process, ongoing quality assurance at the point of collection, and data entry and validation. Data management functions include definition and maintenance of the data storage, access, and retrieval procedures. The functions of project management of the activity can only be defined with respect to the organization involved, but will certainly include monitoring to ensure adequate, reliable data. Consistent and valid information--especially when collected by forms--cannot be obtained without active management support to ensure compliance and cooperation from development personnel.

The major non-staff resource required is some medium for archiving and validating the collected data. For small-scale data collection activities, this may simply require a file cabinet, data entry/process/correction log, and dissemination procedure. For larger operations, a computerized data base is (considering the subject matter) the obvious

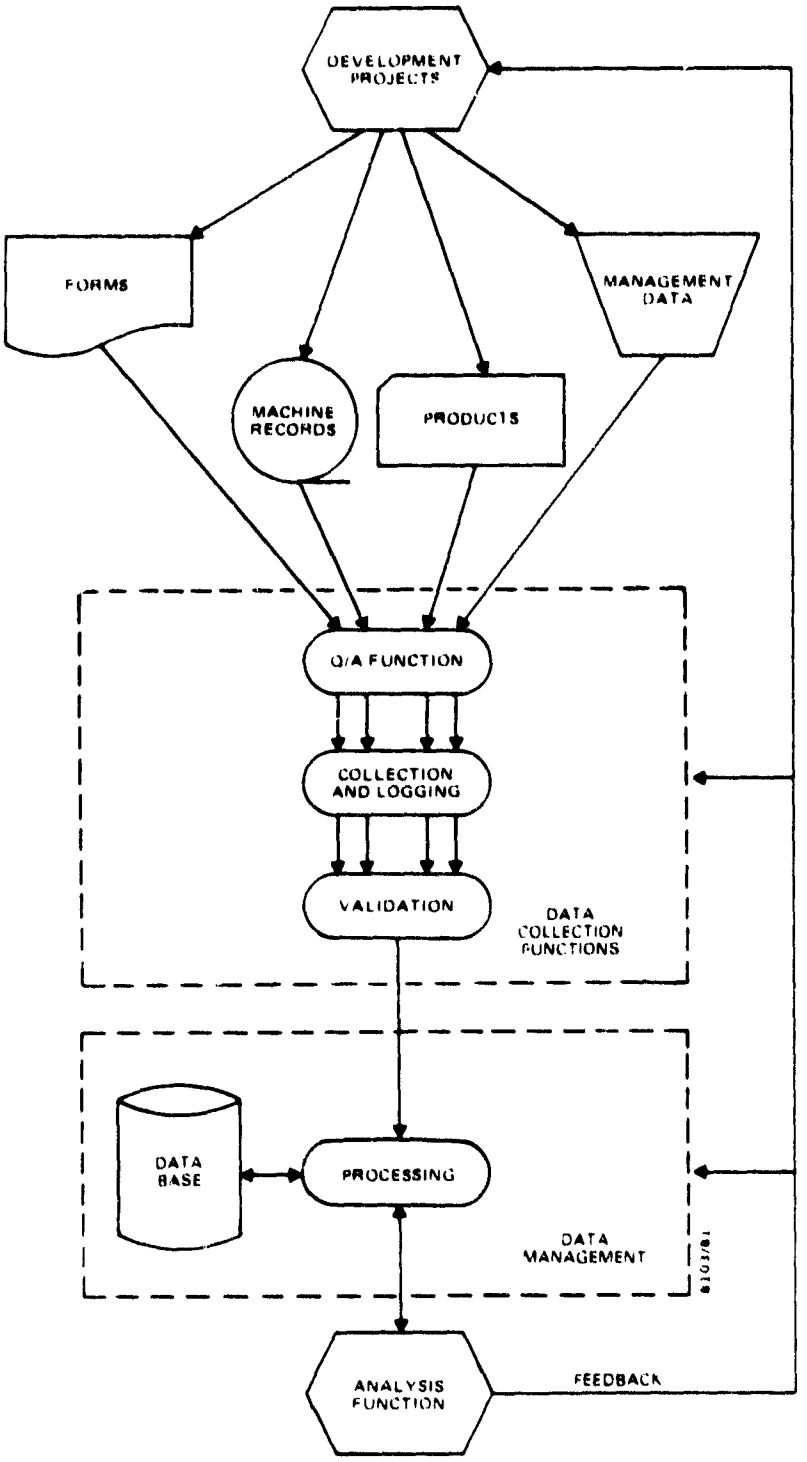

Figure 5-1. Major Functions in Data Collection

Table 5-1. Data Collection Functions (1 of 2)

Implementation

- Design/specify data flow for each type of data and source, QA, monitoring
- Design and validate data forms
- Specify data log format and protocol
- Write procedures guides, instructions

Data Management

- Design files, access mechanisms
- Build data entry and validation mechanisms
- Design/implement maintenance procedures
- Develop data access, reporting procedures
- Provide training and documentation
- Oversee data entry, validation
- Record, evaluate, direct correction of problems and errors
- Provide status reports
- Identify/request/allocate resources
- Provide training for data entry personnel

Data Collection and Support

- Perform data coding, checking, entry
- Perform maintenance under data base administrator direction
- Generate routine reports
- Perform consistency checks on data at point of collection

Table 5-1. Data Collection Functions (2 of 2)

- Perform data cross-checking and validation under direction of data base administrator or data collection supervisor.

Project Management

- Establishes data collection procedures with development projects
- Obtains, allocates resources
- Directs activity

alternative. These functions (whether or not computerized) are discussed in Section 5.3.

5.1.1 IMPLEMENTATION FUNCTIONS

The responsibilities for devising and establishing the collection and monitoring procedures must be defined in the planning stage. Each step of the data collection process must be identified, and monitoring and QA procedures specified. Data collection forms must be designed, and forms-logging protocols established. A manual or protocol of data flow, data descriptions, instruction and responsibilities must be written. Where machine records are to be collected (e.g., accounting data), some regular procedure should be described to minimize delays and loss of data and to fix responsibilities. Procedures for data review at the point of collection (e.g., the staff person filling out forms) must be developed and responsibility assigned.

5.1.2 DATA COLLECTION AND SUPPORT FUNCTIONS

The data collection process must be conducted and monitored on a continuing basis; responsibility for this support function should be established during the planning activity. Machine-supplied records must be requested, acquired and processed. Forms must be collected and logged in a timely fashion. Point-of-collection quality assurance must be established as a regular activity. Attempts to recover missing data must be made and followed up. Particularly important when several projects are active at once, a single central collection and monitoring function will smooth and simplify the data collection process.

QA functions should be the responsibility of someone associated with each development project and trained in the requirements of the software engineering effort. In this way, consistency of data within a project and compatibility across projects can be ensured.

Data entry personnel should be trained to become familiar with the format and typical content of data entry forms. This will minimize errors in data entry and provide an additional check on information content.

5.1.3 DATA MANAGEMENT

The data which have been collected, reviewed, logged, quality assured and assembled must be managed to facilitate access for analysis purposes. This is essentially a data base problem rather than a software engineering problem, and is not treated in depth in this guide. Some notes, however, are appropriate here.

- The organizing (indexing) principles should reflect the data analysis requirements
- The data management system should facilitate identification of missing or incomplete data
- Access procedures should protect any sensitive (proprietary; personnel;...) data without blocking access to the data base
- Because data requirements change with increased understanding, the data management system must be flexible and amenable to re-organization

5.2 DESIGN OF THE DATA COLLECTION PROCESS

As noted above, the data collection process involves collecting, validating, storing, and making available data regarding software development efforts. The overall design of this process involves numerous elements, as shown in Figure 5-1. The logical and physical organization of data must be defined; validation procedures must be specified; data storage and retrieval mechanisms (whether or not computerized) must be identified. The "what" of data collection is driven by the analysis requirements; the "how" depends on the environment and the available resources. The procedural

design (organization, validation, storage, and retrieval) follows from the answers to the "what" and the "how." As illustrated in Figure 1-1, the entire planning and design activity is inherently iterative.

5.2.1 DATA ORGANIZATION

The organizing principles chosen for the data base should simplify the collection and/or analysis activities. The SEL, for example, organizes its data by project and by form-type. Most SEL analyses are related to comparisons of different projects (to identify differences and evaluate methodologies) and are made easier by the project-oriented organization. Data validation procedures and manipulations are typically tied to the original forms (for verification purposes), so the forms-type division is a useful one. Other organizations might be chosen (e.g., by type of data) as long as the storage method used adequately supports retrieval for purposes of editing and analysis.

The SEL data base organization was influenced somewhat by the limits of the computer system used for data base maintenance. As noted in Section 5.2.3 below, different storage systems may impose different requirements or, conversely, provide different opportunities.

5.2.2 DATA VALIDATION

It is assumed here that some quality assurance and validation takes place at the data collection point (Section 5.1.2). To ensure that the data stored is usable, accurate information, an additional validation stage is required in the data processing activity. The SEL data base is computerized, and can therefore make use of automatic validation programs to ensure completeness and consistency. Noncomputerized systems would use manual techniques to ensure that all data is carefully accepted, logged, summarized, and stored for retrieval.

This requirement for validation is, in fact, a significant argument for computerization of the data. Because all items are checked as they are entered, sporadic errors (which might slip past a spot-checking system) are discovered. When found, systematic errors (e.g., consistently entering an incorrect code for a field) can often be corrected en masse on a computerized system.

5.2.3 STORAGE AND RETRIEVAL

Design of the storage and retrieval system greatly depends on the available resources. Possible methods range from simple file folders to elaborate data base systems. The chosen system will greatly impact the types and difficulties of analyses to be performed, with the cost of data entry (highest for computerized systems) balanced against the cost of repeated access and summation (highest for linearly organized manual systems). The intent of this section is merely to emphasize the importance of this design decision. More detail on how to design a system is given in Reference 12.

5.3 COLLECTING THE DATA

As part of the design and planning activity, specific mechanisms of data collection must be identified. Collection methods have been devised for each of the sources described in Section 3. Table 5-2 lists these methods and the data sources for which each is suited. Advantages and drawbacks of each method are discussed below.

In collecting (as in planning) the data, it should be kept in mind that accurate, complete data--even if collected at only a high level--is more useful than a large amount of data of uneven coverage and consistency. The data collection process should be directed at the information that is likely to be available, rather than at collecting a little of everything.

Table 5-2. Data Collection Methods

Source	Collection Methods
Development staff	Data forms Interviews Automated collection
Machine records	Data reduction and cross-correlation of automatically collected information
Development products	Specifically designed analysis programs
Managers	Data forms Interviews Consensus-forming

5.3.1 FORMS

The bane of every programmer's existence (at least, for programmers on monitored projects), data collection forms are perhaps the easiest-to-implement method of collecting data from development personnel.

Properly designed and managed, data forms can provide a wealth of information concerning the development process at both summary and component levels of detail. Forms can also serve the purpose of providing archival storage of ephemeral data (such as the purpose of a test run), permitting data collection to be uncoupled from the data processing function. The iterative nature of software engineering research, of course, implies that this uncoupling could allow inconsistencies to occur in the data base. When new forms are designed with new questions, problems of compatibility may arise.

There may be (as yet) no good alternatives to forms for collecting some types of data. There are, however, some serious drawbacks to the use of forms for extensive data collection. In addition to the impact on the development process and schedule, and the potential morale and compliance problems resulting from the drudgery and boredom of filling out forms, the design of forms is almost an art.

It is the experience of the SEL that development personnel will complete progress/status/exception/etc. forms only with reluctance and with continuing prompting and exhortation from management. When such encouragement has occurred, detailed reports containing a wealth of data have been collected. Despite some grumbling and some often justifiable complaints about unnecessary duplication of effort, forms were completed usefully and consistently. With lukewarm or sporadic management support, forms were completed cursorily or not at all.

Table 5-3 lists some of the desirable characteristics of data collection forms. Although specific data requirements may contravene some of these guidelines, they serve as a target for the art of forms design.

The data collection forms included in Appendix B have been developed and used by the SEL at GSFC. They are included as suggestions and as examples produced by a second-iteration forms design process. These forms reflect the research goals of the SEL and are included not as a prescription but as a guideline.

5.3.2 MACHINE RECORDS

Automatically collected records such as charge accounting data and source library update data form a major and generally reliable source of information about the software development process. A major attraction is the fact that, by definition, these data are collected independently of the software engineering effort. The cost of using them is usually limited to the cost of data reduction and correlation. The specifics of the data available will vary from one installation to another because of differences in systems, accounting software, and chargeback philosophy, and in source code library control systems. In general, however, these data can support at least summary-level analyses, and in many cases (and with greater effort), component-level detail.

Data reduction and cross-correlation efforts typically involve correlating account numbers, users, and job or component names to specific projects or systems and compressing masses of data into a usable synopsis. This may be accomplished by the accounting software itself or may require development of analysis programs. Data security may also be a concern, depending on the facility. The coverage and

Table 5-3. Desirable Forms Characteristics

- Keep Brief (1 side of 1 page)
- Use checkoff or unambiguous short answer
- Provide space for comments
- Ensure that all requested data can be justified
- Use terms which are familiar to the specific environment
- Provide professional looking, quality reproduced forms (rather than typed and xeroxed)
- Use the same forms for all projects
- For Repeated-use forms (e.g., status reports):
  - limit requests to weekly
  - do not ask for the same information repeatedly
- Integrate forms into the development reporting process

accuracy of the data will generally justify the investment in data reduction software.

It would be desirable to use computer-collected records to profile all the parameters of software development. Impact on the development process would be minimized, coverage and accuracy would be improved, consistency and comparability would be maximized. But the front-end investment in tools and procedures that would make this feasible has not yet been made. Computerized accounting records and source code control systems (see Section 5.3.2) provide some such data, but the potential for future initiatives in this area is large. Not all data can be collected automatically, but the limits of what is possible are presently unknown. More research and investment is required.

5.3.3 AUTOMATED DATA ANALYSIS

Much of the output of a software development effort is readily amenable to analysis by appropriately designed software tools. The software product itself is clearly the most reliable source of data on product characteristics, although some condensation and interpretation of data may be necessary for application. Where automated design and/or documentation tools are employed, these products also can be analyzed with minimal investment.

Typically, the analysis programs to provide these data will need to be developed specifically for the software engineering effort. Each type of product (source code, program design language, documents) will require a tailored analysis program. The fact, however, that these sources are fixed, definitive, and machine readable makes this approach highly attractive.

5.3.4 INTERVIEWS AND CONSENSUS

Some types of information are not easily collected on data forms because of the subjective or imprecise nature of the data. For example, motivation for a particular design decision may be useful in understanding a project but not easily recorded on a checklist. Constraints are frequently defined in a complex fashion suitable for the development staff but not for data recording. Judgments on the acceptability, clarity, or maintainability of software are essential to software engineering analysis but difficult to quantify in a consistent manner.

For these types of data, as well as for monitoring of the data collection process in general, interviews or meetings are the most direct and useful procedure. Responses that are not easily collected on checklists can be amplified in discussions between developers and data analysis staff. Consistent scales of evaluation can be developed in meetings of project managers with the software engineering team. The cost of such procedures will limit the extent of use, but some such activities should be anticipated and planned.

5.4 DATA MANAGEMENT

Managing the data after it has been collected and verified involves data entry, data editing, validation, and an interface to the analysts who will use the data. SEL experience shows that collection of accurate and complete data requires effort and experience. The data management system should be designed to support this process by identifying gaps and inconsistencies in the data (e.g., by using sequential form numbers). SEL experience also shows that data requirements change with improved understanding of the development process. The data management system must therefore permit

reorganizations data as new items are added and existing data proves unusable or valueless.

The design and implementation of the SEL data base is described in Reference 12.

SECTION 6 - APPLICATIONS

The software development data base will facilitate many applications of interest to managers and researchers. The manager would like to have monitoring, estimating, and evaluating tools to examine ongoing software development tasks. The substantial variation in the results of software engineering experiments by different researchers suggests that the effect of the local environment on the development process is a powerful one. This can best be understood by assembly of a historical record of development efforts that can be used to "tune" the general models to local conditions (Reference 14). Sections 6.1 through 6.3 discuss the classes of tools that can be devised to take advantage of this historical data base. Some of the software engineering research questions which can be addressed with the data are outlined in Section 6.4. The necessary data for all desired analyses should be included in the initial data collection plan.

6.1 MONITORING

A manager is likely to find the collected data very useful in keeping track of the status of ongoing software development projects. The three types of reports described below would be especially valuable.

Resource utilization (computer time, staff-hours) can be tracked and displayed in tables and/or graphs. Comparison of these values with budgeted values can help monitor development costs.

The progress towards completion of the development effort can also be tracked. The number of modules designed, number of modules coded, number of modules tested, and lines of code developed to date can be determined and report periodically.

Review of design and code measurements made during the development process can be used to detect potential problems such as unmanageably complex code, incomplete design, and low testability. These can be the basis of a concurrent quality assurance program.

6.2 LIFE CYCLE MODELING

The goal of life cycle modeling is to relate the costs of a software development effort to its products. A great many models with different emphases have been devised (References 4, 15, 16). However, as mentioned earlier, all such models should be calibrated in the user's environment with historical data. Thus, one of the important applications of the data base is in determining appropriate model constants.

The utility of life cycle models is as an estimation tool. They provide a method of estimating the cost and product size of a development project.

6.3 METHODOLOGY EVALUATION

Another application of software development data is in the evaluation of methodologies and environmental factors. Development performance information can be used to identify the effect of various development approaches (top-down design, structured programming, etc.) in the user environment. Nonmethodological environmental factors (travel time, group size, etc.) can also be considered in these analyses. Thus, the user can develop a procedure for evaluating and ameliorating his/her software development process.

6.4 RESEARCH

Very few questions about the nature of the software development process have been answered definitively. Extensive work is currently being done in the areas of software metrics (Reference 7), classifications, reliability (References 17, 18), and models. Analysis in all of these areas

has been limited by the lack of substantial reliable data. The reader may wish to address him- or herself to some of these topics.

SECTION 7 - RECOMMENDATIONS

Perhaps the most important recommendation is not to expect instant results. Usable data must be collected over the life of a project, and data on a number of projects must be assembled. This does not happen overnight; as a result, the data collection plans must be laid out to allow adequate time before analysis results are expected.

Beyond that, some thoughts from the SEL are as follows:

1. Subjective management information is very important to project analyses and comparisons.
2. Do not worry about the Hawthorne effect; typical projects are too long and typical programmers are too professional for psychological effects to have a significant impact.
3. Try to provide feedback from the data collection process to the technical staff.
4. Explain the purpose of data collection to the technical staff; try to elicit active support.
5. Do not spend too much time demanding more, or more precise, or more accurate data.

APPENDIX A - SEL DATA COLLECTION EXPERIENCES

Since its formation in 1977, the SEL has monitored more than 40 development projects representing over a million lines of code. The projects include a wide variety of applications, computers, size, and complexity, but they are concentrated in the flight dynamics software area. The major source of data consists of FORTRAN software systems averaging 60,000 lines of code that perform "scientific" data processing of spacecraft telemetry data.

The SEL devised a set of forms to collect the data initially expected to be useful. These forms were revised extensively after a period of use and feedback from the programmers who filled them out. A second revision is currently being considered for the next hiatus in monitored projects.

Data have also been collected from the end-product source libraries using a static source code analyzer called SAP. This program has been modified to count and report the statistics used by various complexity models and measures and to aid in selecting a most consistent definition of "lines of code."

Software currently under development by the SEL will analyze data from accounting tapes maintained by the primary computing facility for monitored projects, an IBM S/360 facility.

The SEL has also developed an extensive system for storing this data using a PDP-11/70 computer system. The data base software, written in FORTRAN for RSX-11M, provides a standard data entry and editing capability. Data entry is performed by data entry clerks under the direction of SEL personnel.

More detail on the SEL is found in Reference 2.

APPENDIX B - SAMPLE DATA COLLECTION FORMS

The forms reproduced here are used by the SEL at the Goddard Space Flight Center to collect data on development projects. The terms used in these forms are defined in Section B.2.

B.1. SAMPLE DATA COLLECTION FORMS AND INSTRUCTIONS

This section contains sample data collection forms and instructions for their use. The instructions precede the forms. The following forms are included

1. General Project Summary (GPS)
2. Resource Summary Form (RSF)
3. Component Summary Form (CSF)
4. Component Status Report (CSR)
5. Run Analysis Form (RAF)
6. Change Report Form (CRF) and Attitude Maintenance Change Report (ATM)

INSTRUCTIONS FOR COMPLETING THE GENERAL PROJECT SUMMARY - FORM 580-1 (2/77)

This form is used to classify the project and will be used in conjunction with the other reporting forms to measure the estimated versus actual development progress. It should be filled out by the project manager at the beginning of the project, at each major milestone, and at the end. Numbers and dates used at the initiation of the project are assumed to be estimated; intermediate reports should change estimates to actuals (if known) and update estimates. The final report should accurately describe the system development life cycle.

A. PROJECT DESCRIPTION

Description. Give an overview of the project.

Inputs. Specifications and requirements (etc.) of project. Give the format of these.

Requirements. How requirements are established and changed.

Products Developed. List all items developed for the project (e.g., operational system, testing system, simulator, etc.).

Products Delivered. List all items required to be delivered (e.g., source of the operational system, object code of the operational system, design documents, etc.).

B. RESOURCES

Target Computer System. System for which software was developed.

Development Computer System. System on which software was developed.

Constraints. List any size or time constraints for the finished product. Do you anticipate any problems in meeting these constraints?

Useful Items From Similar Projects:

1. List previous projects, which will contribute various aspects to this project.
2. For each project, give the percent of the current project it makes up in each of the 3 listed aspects.
3. For each of the 3 listed aspects (specification, design, code) check what level of modifications are necessary.

C. TIME

Start Date. First date of work, including design and modification of the specifications.

End Date. Delivery date.

Estimated Lifetime. Estimate the operational life of the system.

Mission Date. Scheduled operation date of the system (write unknown if not known or undecided yet on any of these dates). Date project must be operational.

Confidence Level. Give the percent probability you think the end date is realistic. (e.g., 100% means certain delivery on that date. 0% means no chance of delivery.)

D COST

Cost. Total amount of money the project costs, including both contract and in-house costs.

Maximum Available. Maximum amount available, independent of what estimated cost is.

Confidence Level. Rate percent reliability in cost estimate.

How Determined. At initiation how is it estimated, at completion how is it calculated.

Personnel. Give the number of full time equivalent persons required at inception of the project, 1/3 of the way into the project, 2/3 of the way into the project, at the completion of the project.

Total Person Months. Give the total number of months that full time equivalent personnel (managers, designers, programmers, keypunchers, editors, secretaries, etc.) are assigned to the project. Do not include all overhead items such as vacation and sick leave.

Computer Time. Give the total number of hours on all systems normalized to one machine (e.g., the IBM 360/75) and name the machine.

E SIZE

Size of the System. Include the total amount of machine space needed for all instructions generated on the project plus the space for data, library routines (e.g., FORTRAN I/O package) and other code already available. Break down size into data space and instruction space.

Confidence Level. Rate percent reliability in size estimates.

Total Number of Source Statements. Give the number of FORTRAN, ALC, or any other language instructions generated specifically for this project.

Structure of System. Give overall structure of system. Is it a single load module, is it an overlay structure, or is it a set of independent programs? For overlay and separate programs, give the number and average size of each.

Define Your Concept of a Module. Give the criteria you are using to divide the software into modules.

Estimated Number of Modules. Include only the number of new modules to be written.

Range in Module Size. Give the number of instructions in the minimum, maximum and average module and the language in which they are written as a reference.

Number of Different I/O Formats Used. Give the number of distinct external data sets that are required for the system including card reader, printer, graphics device, and temporary files.

F COMPUTER ACCESS

A librarian is a person who can be used to perform any of the clerical functions associated with programming, including those given on the chart. Check the appropriate boxes for those persons who have access to the computer to perform the given functions. Give the percentage of time spent by each in batch and interactive access to the computer.

G. TECHNIQUES EMPLOYED

For "level," specify to what level of detail in the finished project the technique is used. (e.g., subroutine, module, segments of 1000 lines, top level, etc.)

Specifications

Functional - Components are described as a set of functions, each component performing a certain action.

Procedural - Components are specified in some algorithmic manner (e.g., using a PDL).

English - Components are specified using an English Language prose statement of the problem.

Formal - Some other formal system is used to specify the components.

Design and Development

Top Down - The implementation of the system one level at a time, with the current level and expansion of the yet to be defined subroutines at the previous higher level.

Bottom Up - The implementation of the system starting with the lowest level routines and proceeding one level at a time to the higher level routines.

Iterative Enhancement - The implementation of successive implementations, each producing a usable subset of the final product until the entire system is fully developed.

Hardest First - The implementation of the most difficult aspects of the system first.

Other - Describe the strategy used if it is not a combination of any of the above.

None Specified - No particular strategy has been specified.

Coding. The final encoding of the implementation in an executable programming language.

Structured Code With Simulated Constructs - The language does not support structured control structures (e.g., FORTRAN) but they are simulated with the existing structures, please state the structured control structures you are using (e.g., WHILE, CASE, IF).

Structured Control Constructs - The language supports structured control structures (e.g., a FORTRAN preprocessor) please list structures you are using.

Other Standard - Describe any other standard you are using.

None Specified - No particular strategy has been specified.

Validation/Verification. Testing: execution of the system, via a set of test cases.

Top Down - Stubs or dummy procedures are written to handle the yet to be implemented aspects of the system and testing begins with the top level routines and proceeds as new levels are added to the system.

Bottom Up - Check out of a module at a time using test drivers and starting at the bottom level modules first.

Structure Driven - Using structure of program to determine test date (e.g., every statement of program executed at least once).

Specification Driven - Using specifications of program to determine test data (e.g., all input output relationships hold for a set of test data).

Other - Describe any other strategy you are using.

None Specified - No testing strategy has been specified.

Validation Verification. Inspection - visual examination of the code or design.

Code Reading - Visual inspection of the code or design by other programmers.

Walk Throughs - Formal meeting sessions for the review of code and design by the various members of the project, for technical rather than management purposes.

Proofs - Formal proofs of the design or code, please specify the techniques used, e.g., axiomatic, predicate transforms, functional, etc.

None Specified - No inspection techniques have been specified.

There is some space given to permit the further explanation of any of the strategies that may be used.

H. FORMAL NOTATIONS USED AT VARIOUS LEVELS AND PHASES

Give the phases (e.g., design, implementation, testing, etc.) and levels (subroutine, module, segments of 1000 lines, top level, etc.) at which any type of formalism (flowchart, PDL, etc.) will be used in the development of the system.

I. AUTOMATED TOOLS USED

Name all automated tools used, including automated versions of the formalisms given above and compilers for the programming languages used, and at which phase and at what level they are used. Include any products that may be developed as part of this project (e.g., simulators).

J. ORGANIZATION

Describe how the personnel are subdivided with respect to responsibilities into teams or groups, giving titles, brief job descriptions, the number of people satisfying that title and their names and organizational affiliations if known.

K. STANDARDS

List all standards used, whether they are required or optional, and the title of the document describing the standard.

L. MILESTONES

Give the phase at which management may check on progress of the development of the system (e.g., specification, design, implementation of version 1, etc.). State also the date at which it should take place (at completion of the project), how it is to be determined that the milestone was reached, who will be responsible for reviewing the progress at that point and what the review procedure will be. Also give the resources used since the last milestone. For

size of system give the current size of the system at that milestone. Each milestone has 2 confidence levels, one for time estimates and one for resource expenditures. For estimated future milestone, the first confidence level for the probability of reaching the milestone at that date. The second is for the accuracy of the resources used. For past milestones, the first confidence level is normally 100% (actual date) while the second is an estimate on the accuracy of the accounting system.

M. DOCUMENTATION

For each time of documentation developed, state the type of documentation, its purpose, the date it should be completed, its size and list any tools used in its production. (At the beginning of the project these should be estimates, at the end of the project, they should be accurate figures.)

N. PROBLEMS

Give the three most difficult problems you expect to encounter managing this project. Please be as specific as possible.

O. QUALITY ASSURANCE

To what do you attribute your confidence in the completed system. Be as specific as possible.

GENERAL PROJECT SUMMARY

PROJECT NAME _____ DATE _____

A. PROJECT DESCRIPTION

Description _____

_____

Form of Input _____

Requirements _____

Products Developed _____

_____

Products Delivered _____

_____

B. RESOURCES

Target Computer Systems _____ Development Computer Systems _____

Constraints: Execution Time _____ Size _____

Other _____

Any Problems in Meeting Constraints? _____

_____

Useful Items from Similar Projects:

Project	Specification				Design				Code			
	%	Major	Minor	None	%	Major	Minor	None	%	Major	Minor	None

C. TIME

Start Date _____ End Date _____ Estimated Lifetime _____ Mission Date _____

Confidence Level _____

D. Cost

Cost $ _____ Maximum Available $ _____ Confidence Level _____

How Cost Determined _____

Personnel: Inception _____ 1/3 Way _____ 2/3 Way _____ Completion _____

Total Person Months _____

Other Costs: Computer Time _____ (hrs) Documentation $ _____

Other ( ) _____ Other ( ) _____

E. SIZE

Size of System _____ Words. _____ Data Words _____ Instructions

Maximum Space Available _____ Words. Confidence Level _____

Total Number of Source Statements: FORTRAN _____ ALC _____

Other ( ) _____

Structure of System (Check One):

___ Single Overlay

___ Overlay Structure (Number of Overlays _____ Avg. Size _____)

___ Independent Programs (Number of Programs _____ Avg. Size _____)

Define Your Concept of a Module _____

_____

Number of Modules _____ Range in Module Size: Min. _____ Max. _____ Avg. _____

Number of Different I/O Formats _____

980-1 (2/77)

F. COMPUTER ACCESS (Check All That Apply. Who Has Access to What.)

	Librarian	Programmer
Keying in New Source Code		
Keying in Update of Source Code		
Inclusion of Code Into System		
Submitting Completions		
Module Testing		
Integration Testing		
Utility Runs (Tape Backup, Etc.)		

Give Percentages for Types of Access:

	Librarian	Programmer
% Batch		
% Interactive		

G. TECHNIQUES EMPLOYED (Check All That Apply and Give Level at Which Used.)

Specification:	Used	Level		Used	Level
Functional			Procedural		
English			Formal		
Design:					
Top Down			Bottom Up		
Iterative Enhance			Hardest First		
Other:			None Used		
Development:					
Top Down			Bottom Up		
Iterative Enhance			Hardest First		
Other:			None Used		
Coding:					
Simulating Construct			Structured Code		
Other			None		
Validation/Verification: Testing					
Top Down (Stubs)			Bottom Up (Drivers)		
Other:			Specification Driven		
Structure Driven			None		
Validation/Verification: Inspection					
Code Reading			Walk Through		
Proof:			None		

H. FORMALISMS USED

	Used	Level	Phases
PDL			
HIPO			
Flowcharts			
Baseline Diag. (Tree Ch.)			
NDS			
Functions			
Other			
Other			

I. AUTOMATED TOOLS USED

Name	Phases in Which Used	Level

J. ORGANIZATION

How are the Personnel Organized: _____

Project Personnel:

Title	Job Description	Number	Names and Affiliations (If Known)

K. STANDARDS

Type _____ Optional _____ Required _____
Title of Document _____

Type _____ Optional _____ Required _____
Title of Document _____

Type _____ Optional _____ Required _____
Title of Document _____

Type _____ Optional _____ Required _____
Title of Document _____

Type _____ Optional _____ Required _____
Title of Document _____

Type _____ Optional _____ Required _____
Title of Document _____

Type _____ Optional _____ Required _____
Title of Document _____

Type _____ Optional _____ Required _____
Title of Document _____

580-1 (2/77) Continuation

L. MILESTONES

Phase ______ Estimated Date ______ Confidence Level ______
How Determined ______
Reviewers ______
Reporting Procedure ______
Resource Expenditures: Cost ______ Person Months ______ Computer Time ______ hrs. ______
Size of System ______ Confidence Level ______

Phase ______ Estimated Date ______ Confidence Level ______
How Determined ______
Reviewers ______
Reporting Procedure ______
Resource Expenditures: Cost ______ Person Months ______ Computer Time ______ hrs. ______
Size of System ______ Confidence Level ______

Phase ______ Estimated Date ______ Confidence Level ______
How Determined ______
Reviewers ______
Reporting Procedure ______
Resource Expenditures: Cost ______ Person Months ______ Computer Time ______ hrs. ______
Size of System ______ Confidence Level ______

Phase ______ Estimated Date ______ Confidence Level ______
How Determined ______
Reviewers ______
Reporting Procedure ______
Resource Expenditures: Cost ______ Person Months ______ Computer Time ______ hrs. ______
Size of System ______ Confidence Level ______

Phase ______ Estimated Date ______ Confidence Level ______
How Determined ______
Reviewers ______
Reporting Procedure ______
Resource Expenditures: Cost ______ Person Months ______ Computer Time ______ hrs. ______
Size of System ______ Confidence Level ______

Phase ______ Estimated Date ______ Confidence Level ______
How Determined ______
Reviewers ______
Reporting Procedure ______
Resource Expenditures: Cost ______ Person Months ______ Computer Time ______ hrs. ______
Size of System ______ Confidence Level ______

Phase ______ Estimated Date ______ Confidence Level ______
How Determined ______
Reviewers ______
Reporting Procedure ______
Resource Expenditures: Cost ______ Person Months ______ Computer Time ______ hrs. ______
Size of System ______ Confidence Level ______

Phase ______ Estimated Date ______ Confidence Level ______
How Determined ______
Reviewers ______
Reporting Procedure ______
Resource Expenditures: Cost ______ Person Months ______ Computer Time ______ hrs. ______
Size of System ______ Confidence Level ______

580-1 (2/77) Continuation

M. DOCUMENTATION

Type _____ Purpose _____
Estimated Date _____ Estimated Size _____ Tools Used _____

Type _____ Purpose _____
Estimated Date _____ Estimated Size _____ Tools Used _____

Type _____ Purpose _____
Estimated Date _____ Estimated Size _____ Tools Used _____

Type _____ Purpose _____
Estimated Date _____ Estimated Size _____ Tools Used _____

Type _____ Purpose _____
Estimated Date _____ Estimated Size _____ Tools Used _____

Type _____ Purpose _____
Estimated Date _____ Estimated Size _____ Tools Used _____

Type _____ Purpose _____
Estimated Date _____ Estimated Size _____ Tools Used _____

N. PROBLEMS

State the three most difficult problems you expect to encounter in completing the project. (1 = most difficult)

1. _____

2. _____

3. _____

O. QUALITY ASSURANCE

State the three most important aspects of the design, development and testing of the system to which you attribute your confidence in the completed system. (1 = most important)

1. _____

2. _____

3. _____

PERSON FILLING OUT FORM _____

980-1 (2/77) Continuation

INSTRUCTIONS FOR COMPLETING THE RESOURCE SUMMARY

This form keeps track of the project costs on a weekly basis. It should be filled out by the project manager every week of the project duration.

PROJECT. Give project name.

DATE. List date form turned in.

NAME. Name of project manager.

WEEK OF. List date of each successive Friday.

MANPOWER. List all personnel on the project on separate lines. Give the number of hours each spent that week on the project.

% OF MANAGEMENT. Add the % of time this person spent managing the project during this reporting period. A new form should be used if this % changes.

COMPUTER USAGE. List all machines used on the project. For each machine give the number of runs during each week and the amount of computer time used.

OTHER. List any other charges to the project.

RESOURCE SUMMARY

PROJECT _____ DATE _____

NAME _____

WEEK OF											
MANPOWER (HOURS)											% OF MGMT.
COMPUTER USAGE (NO. RUNS/HOURS CHARGED)											
OTHER CHARGES TO PROJECT											

560-3 (6/78)

INSTRUCTIONS FOR COMPLETING THE COMPONENT SUMMARY

This form is used to keep track of the components of a system. A component is a piece of the system identified by name or common function (e.g., an entry in a tree chart or baseline diagram for the system at any point in time, or a shared section of data such as a COMMON block). With the information on this form combined with the information on the Component Status Report, the structure and status of the system and its development can be monitored.

This form should be filled out for each component at the time that the component is defined, at the time it is completed, and at any point in time when a major modification to the component is made. It should be filled out by the person responsible for the component.

PROJECT. Give project name.

DATE. Give date form filled out.

NAME OF COMPONENT. Give name (up to 8 characters) by which the component will be referred to in other forms.

BRIEF DESCRIPTION. State function of component.

TYPE OF SOFTWARE. Check all classifications that apply. All common blocks are separate components.

STATUS OF COMPONENT. Check whether this is a new component, whether it is a component under development (e.g., a previous component summary has already been submitted), or whether the component is now complete.

A. CODE SPECIFICATIONS. Give the form of design for this component and tell to what level of detail the specifications are given.

Functional—Components are described as a set of functions, each component performing a certain action.

Procedural—Components are specified in some algorithmic manner (e.g., using a PDL).

English—Components are specified using an English Language prose statement of the problem.

Formal—Some other formal system is used to specify the components.

Relative to the one developing the component, rate the precision of the specifications. Very precise means that no additional analysis on the problem is needed, precise means that only easy or trivial ideas have to be developed, and imprecise means that much work still remains in developing this component and its basic structure.

B. INTERFACES

Give the relative position of this component in the system. Give the number and list the names of all components that call this component, and are called by this component. Also, give the names of any components or other items this component shares with other components (e.g., COMMON blocks, external data). The components directly descended from this component refers to the tree chart or the system. If the interfaces are not yet complete, check "Not Fully Specified"

C. PROGRAMMING LANGUAGES

List languages (or assembly languages) to be used to implement this component. If more than one, list percentages of each (in lines of source code). If there are any constraints on the component (e.g., size, execution time) list them. Also give estimated size of finished component in terms of source statements, (estimate size with comments and without comments) and resulting machine languages (including data areas, but not COMMON blocks).

Useful Items From Similar Projects

1. List previous components and projects which contribute various aspects to this component.

2. For each such component, give the percent of each of the three listed aspects it makes up (e.g., a component may be 50% of design but only 25% of code due to changed interfaces, etc.).

3. For each of the three listed aspects, check what level of modifications are necessary.

D. COMPLEXITY

Rate your belief in the complexity of the implementation. Also approximate the number (by %) of assignment type statements (input statements are included) and control statements (those that alter the flow of control, e.g., IF, CALL, GOTO). The sum of these two may not be 100% (e.g., CONTINUE, DIMENSION and REAL statements will not be counted). I/O and declarations should be listed as other.

E. RESOURCES TO IMPLEMENT

For each of the three listed phases (Design, Code, Test), estimate computer runs, time needed, hours to implement, and estimated completion date. If not known, or no estimate can be given, write "unknown".

F. ORIGIN OF COMPONENT

If this component is independent of any other component of the system (e.g., is a low level component which is designed first, or is the root node of the tree chart) then check yes, otherwise check no.

If no is checked, then explain why the component was added. (Usually only one reason will be checked, although more may be checked, if appropriate).

A lower level elaboration of a higher level component means that an existing component was expanded to include new components (e.g., expanding tree chart). List the higher level component time.

Added as a driver or interface means that a calling program was added to call existing components. List these called components.

A redesign of an existing component means that new capabilities were added to an already existing component. Write its name.

A renaming of an older component. Give the old name.

A regrouping of existing material means that several components were redesigned with a new component resulting from this redesign. Give the old component names.

Type of addition. Why was this component added to the system at this time? Check the appropriate reason. (Normally, only one should be checked, although more can be if appropriate.)

G. ADDITIONAL COMMENTS. Add any other comments that will help explain the purpose, design, and complexity of this component.

H. PERSON RESPONSIBLE. Include name of person responsible for implementing component.

I. PERSON FILLING OUT FORM. Give name of person filling out form. This normally is the same name as in H.

COMPONENT SUMMARY

PROJECT _____ DATE _____

NAME OF COMPONENT _____ CREATION DATE _____

BRIEF DESCRIPTION _____

STATUS OF COMPONENT NEW _____ UNDER DEVEL _____ COMPLETED _____

TYPE OF SOFTWARE (Check All That Apply)

- ____ I/O Processing
- ____ Algorithmic
- ____ Logic Control
- ____ Systems Related
- ____ DATA/COMMON Block
- ____ Other

A. CODE SPECIFICATIONS (Check All That Apply)

FORM OF DESIGN	LEVEL OF DETAIL				
	Component	Subcomponent	Basic Block Segment	Stmt	Other
Functional					
Procedural					
English					
Formal					
Other ( )					

Precision of Code Specification Very Precise _____ Precise _____ Imprecise _____

B INTERFACES

Number Components Called _____ Names _____ Not Fully Specified _____

Number Calling This Component _____ Names _____ Not Fully Specified _____

Number Shared Items _____ Names _____ Not Fully Specified _____

Number of Components Directly Descended from This Component _____ Names _____ Not Fully Specified _____

C. PROGRAMMING LANGUAGES

Languages Used and Percentages _____ ( ) _____ ( )

CONSTRAINT PROBLEM EXPECTED

	Constraint Present	Component Meets Constraint
Memory Space		
Execution Time		
Other ( )		

Size Source Statements (Including Comments) _____ Machine Bytes _____

Source Statements (Not Including Comments) _____

Useful Items From Similar Projects

Component	Project	Specification				Design				Code			
		%	Major	Minor	None	%	Major	Minor	None	%	Major	Minor	None

SEL-5 (6/78)

D COMPLEXITY

Complexity of Function Easy _____ Moderate _____ Hard _____

_____ % Assignment Statements _____ % Control Statements _____ % Other Statements (e.g., Data Declarations)

E RESOURCES TO IMPLEMENT

	Runs	Computer Time (min)	Effort (hrs)	Est. Completion Date
Design				
Code				
Test				

F Is this component independent of the existing components? _____ Yes _____ No

If No, describe relation of this component to the existing system

_____ inserted as a lower level elaboration of higher level components (names) _____

_____ added as a driver or interface for existing components (names) _____

_____ a redesign (to add new capability) of existing components (names) _____

_____ a renaming of existing component (name) _____

_____ regrouping of existing material from several components (names) _____

_____ other _____

Type of Addition

_____ error correction

_____ planned enhancement

_____ implementation of requirements change

_____ improvement of clarity, maintainability, or documentation

_____ other (explain below)

_____ improvement of user service

_____ utility for development purposes only

_____ optimization of time/space/accuracy

_____ adaptation to environment change

G ADDITIONAL COMMENTS

H PERSON RESPONSIBLE FOR IMPLEMENTING COMPONENT _____

I PERSON FILLING OUT FORM _____

350-5 (6/78)

INSTRUCTIONS FOR COMPLETING THE COMPONENT STATUS REPORT

This form is to be used to accurately keep track of the development of each component in the system. A Component Summary Report should exist for each component mentioned. The form is to be turned in at the end of each week. Please fill out either daily or once each week. If daily, then a given component may be listed several times during the course of a week. For each component list the number of hours spent on each of the listed activities. This form should be filled out by persons working on the project.

PROJECT Name of the project.

PROGRAMMER Name of programmer.

DATE Date report turned in. Usually the date of a Friday.

COMPONENT Name of component. Either a part of the system structure for which there is a component summary form, or one of the following:

JCL. Developing command language instructions.

Overlay. Developing system overlay structure.

User Guide. User's Guide Documentation.

System Description. System Description Documentation.

DESIGN

Create. Writing of a component design.

Read. Reading (by peer) of design to look for errors. (e.g., peer review)

Formal Review. Formal meeting of several individuals for purpose of explaining design. Also include time spent in preparing for review. All those attending review should list components discussed in their own Component Status Report for that week.

CODE/DEVELOPMENT

Code. Writing executable instructions and desk checking program.

Read. Code reading by peer. Similar to Design Read above.

Formal Review. Review of coded components. Similar to Design Review above.

TESTING

Unit. Unit testing. Test run with test data on single module.

Integ. Integration testing of several components.

Review. Review of testing status.

OTHER. Any other aspect related to a component of the project not already covered other than Design, Code Development, Test (e.g. Documentation of a specific component). List type of activity, and hours spent on that activity. A set of activities has been listed for which time may be charged to the overall project:

Travel. Time spent on official travel related to this project, including trips to and from GSFC)

Forms. Time spent on filling out reporting forms.

Meetings. Time spent in meetings which are not design or code review meetings.

Training. Training activities identified for project.

Acc Test. Acceptance Testing activities.

COMPONENT STATUS REPORT

PROJECT _____ DATE _____

PROGRAMMER _____

COMPONENT	DESIGN			CODE DEVELOPMENT			TEST			OTHER	
	CREATE	READ	FORMAL REVIEW	CODE	READ	FORMAL REVIEW	UNIT	INTEG	REVIEW	ACTIVITY	HRS
										Travel	
										Forms	
										Meetings	
										Acc Test	
										Training	
JCL											
Overlay											
User Guide											
System Desc											

INSTRUCTIONS FOR COMPLETING THE COMPUTER PROGRAM RUN ANALYSIS FORM

This form will be used to monitor the activities for which the computer is used in the course of a project life cycle. An entry should be made for each computer run—including all activities performed when the computer is used in an interactive mode.

PROGRAMMER. Write down name of person preparing computer runs. This may not necessarily be the person running the program (i.e., librarian).

PROJECT. Write down project name. Use a different form for each project.

COMPUTER. Indicate the machine on which these runs were made (e.g., S/360, PDP-11, CS).

DATE. Date form turned in.

JOB ID. Identification of job.

RUN DATE. Date run submitted in format MM-DD (month-day).

INTERACTIVE. Place an X if the run was submitted from an interactive terminal.

RUN PURPOSE. Place an X in all boxes that describe this run.

Unit Test. A purpose of the run is to test one or more components without the rest of the system being configured into the load module. A run which uses a 'test driver' would fall into this category.

System Test. This run executes a load module which contains all of the currently available system in order to test one or more components in a full system configuration.

Benchmark Test. This is a recertification type run. A run that has successfully executed in the past is now rerun to verify that certain capabilities still exist.

Maintenance/Utility. A purpose of this run is to perform a 'library-type' function. Examples are runs that update source, create backups, delete compress/copy data sets.

Compile/Assembly/Link. A purpose of the run is to check for errors in the compile, assembly and or link steps. A run which includes one or more of these steps simply as a prerequisite to a system execution would not fall into this category.

Debug Run. This run was submitted in order to investigate a known error.

Other. This run has a purpose which does not fall into one of the other categories. Examples are runs which access other systems in order to aid in the design, development and/or testing of the project under study.

COMPONENTS OF INTEREST. List all components important to this run (e.g., components being tested, compiled, copied, etc.).

FIRST RUN. Place an X here if this is the first time any of the listed components have been processed by the computer for the purpose of run specified.

MEETS OBJECTIVES. This is a subjective evaluation of whether the run satisfied your objectives. Runs that terminate in errors may be satisfactory if the objective was to locate errors or to test for correctness; runs that terminate normally may be unsatisfactory if the purpose was to locate an error known to be present. Thus this question is independent of whether the program contained any errors or not.

RUN RESULTS. Check the box that best describes the results of this run. Normally only one box is checked, although more than one may be checked if appropriate.

Good Run. Program ran to termination with no known errors.

Setup Error. Error in creating program deck.

Submit Error. Deck submitted incorrectly, resources unavailable, keypunch error, or general submission error.

JCL Error. JCL statement incorrect. (JCL cards mistyped should be listed under submit errors.)

Other Setup Error. Such as insufficient space or time specified for job step. This should not be caused by program error.

Machine Error. Errors outside of the control of the programmer.

Hardware Error. Machine malfunction.

Software Error. System crash or system program error (e.g., error in FORTRAN compiler).

Program Error. Error caused by the submitted program.

Compile Error. The source program contains an error which is found by the compiler or assembler.

Link Error. The loader or linkage editor finds an error.

Execute Error. System error messages are generated during the execution step, possibly causing an abend.

User Generated Error. The program terminates in a programmer generated error message which is not a system error.

Ran to Completion. The program terminated with no error message; however, the results are incorrect signifying that there is something wrong with the program.

COMMENTS. If you believe that your answers to these questions do not adequately characterize this run, you may add any additional comments that you wish. Also use this space to indicate if the run was lost before you had a chance to evaluate results.

COMPUTER PROGRAM RUN ANALYSIS

PROGRAMMER _____ PROJECT _____ COMPUTER _____ DATE ____/____/____

JOB ID	DATE MM DD	INTERACTIVE	RUN PURPOSE								COMPONENTS OF INTEREST				RUN RESULTS												COMMENT (e.g., Run Lost, No Results)
			Unit Test	System Test	Benchmark Test	Maintenance/Utility	Compile/Assembly/Link	Debug Run	Other		First Run	Run Met Objectives	Run Did Not Meet Objectives	Good Run	Good Run	Setup Error	Submit Error	JCL Error	Other Setup Error	Machine Error	Hardware Error	Software Error	Program Error	Compile Error	Link Error	Execute Error	User Generated Msg	Run to Completion	

SBD 7 (6/78)

INSTRUCTIONS FOR COMPLETING THE CHANGE REPORT FORM

This form is used to keep track of all changes made to a system. A change is any alteration to the design, documentation, or code generated for a project. Each change can be thought of as a step in the process of transforming the original software design into a complete working system. The initial creation of sections of fresh code or design is not a change.

One change report form should be filled out for each change. Where several changes are made simultaneously for different reasons a separate form should be completed for each reason.

NUMBER. A unique identifier per form per day consisting of initials followed by a sequence number. The initials should be those of the person filling out the form. The sequence number should be a positive integer indicating the number of forms filled out so far during the day. Number DMW01 indicates the first form of the day filled out by DMW. DMW02 is the second form that day, etc.

PROJECT NAME. The name of the development project.

CURRENT DATE. The date on which an entry is first made on the form, even if the form is not completed on that day.

SECTION A—IDENTIFICATION

REASON. Explain why the change is being made.

DESCRIPTION. Describe the change that is being made. This should not be on the variable name or bit level, but should be sufficiently abstract so that the function of the changed code can be determined, e.g., "the input buffer was cleared," rather than "array buff was set to zero."

EFFECT. What components (or documents) are changed? List the names of all components and documents modified as part of the change, including version numbers.

EFFORT. What additional components (or documents) were examined in determining what change was needed? List all components and documents that were examined, but were not actually changed, in deciding what change to make, how to make it, and where to make it. This list should not overlap with the list of components and documents actually changed.

DATES OF CHANGE. Need for change determined on. Give the date on which it was first realized that a change was needed.

Change started on. Give the date on which the change was started.

What was the effort in person-time required to understand and implement the change?

Give the best available estimate of the total time needed to understand what change had to be made and how to make it, including the implementation time. This should include the time of all persons involved in making the change. As an example, if two people each worked 6 hours on the change, the space marked "one day to 3 days" should be checked.

SECTION B—TYPE OF CHANGE

Check the one box that best describes the change. If none of the change descriptions seem to fit, check other and give a detailed description of the change in Section E. If several of the descriptions seem equally appropriate, more than one box may be checked.

Error Correction. A change made to correct an error in previous work. If this box is checked Sections C and D of the change report form should be completed.

Planned Enhancement. The insertion of a body of code into a program stub that was initially created as a dummy for testing purposes or adding capability to an already existing component as part of a planned incremental development.

Implementation of Requirements Change. Altering the system to conform to a change in requirements imposed by the customer.

Improvement of Clarity, Maintainability, or Documentation. Changes made to improve code quality, such as improving indentation of code, resequencing labels for readability, adding or updating documentation or correcting literary errors in it, suppressing redundant information or replacing multiply-occurring sections of code with procedure calls. Corrections of violations of programming standards and design improvements that should have been visible in the functional specifications of components of the system are to be treated as error corrections. Documentation updates made concomitantly with a change should be treated as a part of that change and classified with the primary cause of the change.

Improvement of User Services. During system development, individual programmers may find that with very little extra work they can provide the user with additional facilities on top of the functional requirements of the system. Such changes are classed as improvements to user services.

Insertion/Deletion of Debug Code. Changes made to the program text specifically to provide additional information during test runs so that errors can be isolated.

Optimize Time/Space/Accuracy. An optimization is a localized adjustment of the program whose main purpose is to reduce its execution time or memory requirements, or to obtain results of greater numerical accuracy by tuning the algorithms used to the specific problem being solved.

Adaptation to Environment Change. The "boundary" of a software system is defined to include just those programs whose development and maintenance is being monitored as part of the software engineering laboratory project. A change whose cause lies outside this boundary (e.g., in response to an operating system, compiler, or hardware change) is regarded as environmentally caused.

Was more than one component affected by the change? A component is defined to be directly involved in a change if it contains subroutines that are changed and it contains no subcomponents containing those subroutines. Check yes if the change directly involves more than one component of the system, no otherwise. It may be the case that a change to one subroutine/component will require some future adjustment in other components (these components may not even have been coded yet, or their adaptation may be postponed). In such cases, the effects of the change involve more than one component even though only one module was noted as changed on this form.

SECTION C--TYPE OF ERROR

Check the one box that best describes the error. If none of the error descriptions seem to fit, check other and give a detailed description of the error in Section E.

Requirements Incorrect or Misinterpreted. Requirements may be incorrect (inconsistent or ambiguous), or their meaning may be misinterpreted. In either case, an error of this type, if undetected early, may propagate through design and into code. Even if undetected until acceptance testing (or maintenance), errors resulting from incorrect or misinterpreted requirements should be classified in the requirements error category.

Functional Specifications Incorrect or Misinterpreted. Functional specifications are taken to be a specification of a component as a set of functions defining the output for any input. Similar to requirements, specifications may be either incorrect or misinterpreted. Errors in the specifications that occur as a result of misunderstandings of requirements are classified as misinterpreted requirements errors and not incorrect specifications. Specification errors that result from misunderstandings among those writing the specifications are classified as incorrect specifications. Errors in code or design or documents resulting from incorrect or misinterpreted specifications should be classified in the specifications error category.

Design Error Involving Several Components. A design decision is a choice of organization of a component into subcomponents, including the specification of the interfaces among the subcomponents. A design error is a design decision that results in one of the following:

- interfaces that contain insufficient, unnecessary, or redundant information;
- a set of subcomponents that do not satisfy the specifications of the component (i.e., one or more of the subcomponents do not have the capabilities needed to satisfy the use intended for the component).

Note that a design error may result from incorrect or misinterpreted requirements or specifications. In such cases, the error should not be classified as a design error, but as a requirements or specification error.

Error in the Design or Implementation of a Single Component. Most simple, localized programming mistakes fall into this category. It contains those cases where the organization of the system into components and their interfaces is correct, but a particular component does not behave according to its intended use (i.e., does not correspond to its specification). This may occur because the algorithm used in designing the component is incorrect, or because the implementation of the algorithm is incorrect. If the algorithm has a written specification prior to code generation, and the specification is incorrect or misinterpreted, the error is not classified as a design or implementation error, but as a specification error. If the erroneous algorithm has no written specification, or if the implementation of the algorithm has errors not attributable to any other category, then the error is classified as an error in the design or implementation of a single component.

Misunderstanding of External Environment, Except Language. Check this box if the error resulted from mistaken assumptions about the hardware or software environment in which the program operates (i.e., that software outside the "boundary" of the project--see "adaptation to environment change" in Section B). Included here are mistaken assumptions about how the operating system works, about how the hardware is controlled, about response of peripherals to various commands, about the operation of the library system, about the interface to special display hardware or software, etc.

Error in Use of Programming Language/Compiler. Errors in the use of the language/compiler are those errors that result from some misunderstanding of how the compiler works, how the language provided run-time support system operates, or some misunderstanding of particular language features. Not included in this category are clerical errors (e.g., typos) that lead to compilation errors.

Clerical Error. Clerical errors are those errors that occur in the mechanical translation of an item from one format to another (e.g., one coding sheet to another), or from one medium to another (e.g., coding sheets to cards). No interpretation or semantic translation is involved in such a process.

FOR DESIGN OR IMPLEMENTATION ERRORS ONLY

This section should be filled out only if the error was a design error, involving several components, or if it was an error in the design or implementation of a single component. Errors that occur in the design of a system, subsystem, set of components, or single component, or in the implementation of a single component, may be categorized in one of two ways. Either there was an error in the use of data, or there was an error in the function of a component (such as an algorithmic or computational error resulting in program behavior not corresponding to the intended use of the program). Data use errors can be characterized as either incorrect values for data items or improper assumptions about the structure of data items (e.g., array sizes or dimensions, or ordering of items in a list). Errors involving the function of a component include control and computational errors, such as incorrect sequencing of statements, omitted statements (where such are not clerical errors), improperly computed expressions, omitted capabilities of the component(s), etc.

SECTION D—VALIDATION AND REPAIR

What were the activities used to validate the program, to detect the error, and find its cause?

The purpose of this section is to discover how it became known that an error existed and how the cause of the error was determined. A check should be put in the first column for each method used for validating the component(s) where the error was found. A check should be put in the second column on the same line as the method by which the symptoms of this particular error was first noted. The third and fourth columns refer to activities used to find the cause of the error, once it was known that the error existed. In the third column, check all techniques used in trying to find the cause of the error. In the fourth column, check those techniques that yielded the information needed to find the cause. In some cases, such as some errors found by code reading, the technique(s) used to find the error and discover its cause will be the same. Note that error messages have been divided into two categories: those produced by the support system (e.g., compiler, operating system), and those designed into the code for the specific purposes of the project. Testing has also been divided into two categories: test runs made prior to acceptance testing (pre-acceptance test runs), and acceptance tests. If activities other than those listed in the table were used in finding the error or discovering its cause, check other in the appropriate column, and describe the activities used in Section E. This table inevitably has some redundancy: a check in column 2 must always have a corresponding check in column 1, similarly with columns 4 and 3.

What was the time used to isolate the cause?

Check the space that most closely approximates the time required to isolate the cause of the error. This should be the total of the time that was spent in the activities tried to find the cause. If the cause of the error was never found, and a workaround was used, check the appropriate box. If the cause was never found and a workaround was not used, explain the circumstances in Section E.

Was this error related to a previous change?

Changes to software may result in errors because of one or more of several reasons:

- the change was incorrectly implemented, i.e., did not conform to its specification;
- the change invalidated an assumption made elsewhere in the software;
- an assumption made about the rest of the software in the design of the change was incorrect.

An error is related to a previous change if it results from one of the above three conditions. Errors that are uncovered by changes, i.e., an error masked by another that is revealed when the latter is corrected, do not belong in this category. If the error is related to a previous change, give the number and date of the change report form of the related change. When did the error enter the system?

Check the box that most closely represents the phase in the erroneous components' development in which the error was introduced.

SECTION E—ADDITIONAL INFORMATION

This section is intended to permit further explanation of any items you feel may be significant in categorizing the change (including error corrections). If the "other" category was checked in any of the previous sections of the form, a fuller explanation should be given here. Do not hesitate to give a full description of the error or change or any doubts you may have in classifying it. The accuracy of our analysis is dependent on the amount and accuracy of the data you provide for us. The study we are performing is an attempt to do a careful, detailed investigation of the processes that go on during software development, the kinds of changes and errors that occur during development, and the reasons for their occurrence. With your help, we hope to gain enough insight into the design, coding, and testing of programs so that proposed techniques for coping with software changes and reducing the number of errors can be evaluated. Your cooperation and patience in completing the change report form each time you make a change to a document or program are needed and appreciated.

NUMBER _____

CHANGE REPORT FORM

PROJECT NAME _____ CURRENT DATE _____

SECTION A - IDENTIFICATION

REASON: Why was the change made? _____

DESCRIPTION: What change was made? _____

EFFECT: What components (or documents) are changed? (Include version) _____

EFFORT: What additional components (or documents) were examined in determining what change was needed? _____

	(Month)	(Day)	(Year)
Need for change determined on			
Change started on			

What was the effort in person time required to understand and implement the change?

_____1 hour or less, _____1 hour to 1 day, _____1 day to 3 days, _____more than 3 days

SECTION B - TYPE OF CHANGE (How is this change best characterized?)

- ☐ Error correction
- ☐ Planned enhancement
- ☐ Implementation of requirements change
- ☐ Improvement of clarity, maintainability, or documentation
- ☐ Improvement of user services
- ☐ Insertion/deletion of debug code
- ☐ Optimization of time/space/accuracy
- ☐ Adaptation to environment change
- ☐ Other (Explain in E)

Was more than one component affected by the change? Yes _____ No _____

FOR ERROR CORRECTIONS ONLY

SECTION C - TYPE OF ERROR (How is this error best characterized?)

- ☐ Requirements incorrect or misinterpreted
- ☐ Functional specifications incorrect or misinterpreted
- ☐ Design error, involving several components
- ☐ Error in the design or implementation of a single component
- ☐ Misunderstanding of external environment, except language
- ☐ Error in use of programming language/compiler
- ☐ Clerical error
- ☐ Other (Explain in E)

FOR DESIGN OR IMPLEMENTATION ERRORS ONLY

→ If the error was in design or implementation:

The error was a mistaken assumption about the value or structure of data _____

The error was a mistake in control logic or computation of an expression _____

580-2 (6/78)

FOR ERROR CORRECTIONS ONLY

SECTION D - VALIDATION AND REPAIR

What activities were used to validate the program, detect the error, and find its cause?

	Activities Used for Program Validation	Activities Successful in Detecting Error Symptoms	Activities Tried to Find Cause	Activities Successful in Finding Cause
Pre-acceptance test runs				
Acceptance testing				
Post-acceptance use				
Inspection of output				
Code reading by programmer				
Code reading by other person				
Talks with other programmers				
Special debug code				
System error messages				
Project specific error messages				
Reading documentation				
Trace				
Dump				
Cross-reference/attribute list				
Proof technique				
Other (Explain in E)				

What was the time used to isolate the cause?

____one hour or less, ____one hour to one day, ____more than one day, ____never found

If never found, was a workaround used?_____Yes_____No (Explain in E)

Was this error related to a previous change?

_____Yes (Change Report #/Date_____) ____No ____Can't tell

When did the error enter the system?

____requirements ____functional specs ____design ____coding and test ____other ____can't tell

SECTION E - ADDITIONAL INFORMATION

Please give any information that may be helpful in categorizing the error or change, and understanding its cause and its ramifications.

Name: _____ _____Authorized: _____ Date: _____

SEL-82 (6/78)

Current Date _____

_____Attitude System Maintenance Report

Project Name _____Need for Change determined on (Mo., Day, Yr.)_____

Describe Change _____

_____

_____

_____

What components/subroutines/modules are changed _____

_____

_____

CHANGE (NON-ERROR) (Fill out this section if change is NOT an error correction)
This change is being made because of a change in: (Check all that apply)

_____ requirements
_____ new information/data
_____ specification
_____ design
_____ other (specify): _____

_____ hardware environment
_____ software environment
_____ optimization

ERROR ONLY (Fill out this section if change IS an error correction)
The following activities were used in error detection or isolation: (Check all that apply) (Put D for detection, I for isolation)

_____ normal use
_____ test runs
_____ code reading
_____ reading documentation
_____ other (Specify): _____

_____ trace/dump
_____ cross reference/attitude list
_____ system error messages
_____ project specific error messages

Which of the following best describes the error:

_____ requirements error
_____ design error
_____ error in translating design or specification to code
_____ other: Describe_____

_____ specification error
_____ clerical error

Was this error related to a previous maintenance change _____yes _____no ____ can't tell

Please give any information that may be helpful in categorizing and understanding the change on the reverse side of this form.

Person filling out this form _____
Approved _____Date _____

Change started on date (month, day, year) _____

Time spent on this change:
_____ less than 1 day _____1 day to a week _____more than a week

B.2. SEL GLOSSARY OF TERMS USED WITH DATA COLLECTION FORMS

This section defines the terms used in the software engineering data collection forms reproduced in Section B.1. A more extensive glossary (based substantially on this one) is found in Reference 19.

assignment statements	All statements that change the value of a variable as their main purpose (e.g., assignment or READ statements, but the assignment of the DO loop variable in a DO statement should not be included).
attitude/orbit	Any component that is directly related to either the attitude determination (or control) task or to the orbit determination (or control) task falls into this category. This should include full systems in general (such as GTDS or ISEE-B Attitude) as well as specific modules such as Deterministic Attitude or DCCONES.
attribute list	A compiler-generated list of the identifiers used by a program that describes the characteristics of those identifiers and shows the source statements where they are first defined (or first used) and, for variables, their (relative) storage locations.
automated tools	Any programs whose purpose is to aid in software development (e.g., compiler, text editor, or dump or trace facility). This includes compilers but not standard operating system software (e.g., linkage editor).
baseline diagram	A structured chart listing all components in a system in which a connection from a higher component to a lower one indicates that the higher component calls the lower one.
batch	Use of a computer in which the entire job is read into the machine before the processing begins and in which there is no provision for interaction with the submitter during execution of the job. (Interactive usage is always via a terminal; batch usage may be via a terminal or a card deck.)

bottom-up	The design (or implementation) of the system starting with the lowest level routines and proceeding to the higher level routines that use the lower levels.
business/ financial	The second of the four major categories applies to components related to some accounting task, financial data formatting, business data retrieval or reporting, or possibly personnel data management. Very few of the components being studied will fall into this class.
change	A modification to design, code, or documentation. A change might be made to correct an error, to improve system performance, to add capability, to improve appearance, or to implement a requirements change, for example.
clerical	The process of copying an item from one format to another or from one medium to another, which involves no interpretation or semantic translation.
code reading	Visual inspection of the source code by persons other than the creator of the code.
command/ control	This class of components includes those used either to generate vehicle commands or to transmit these commands from the control center.
complexity	Measures the difficulty of implementing a component, independent of the implementer's experience. Easy (or simple) means that any good programmer can write down the correct code with little thought. Hard (or complex) means that much thought is involved in the design. (Compare this with "precise;" e.g., easy and imprecise may mean a vague specification, but once the approach is decided upon, the code is easy to write.)
component	A piece of the system identified by name or common function (e.g., separately compilable function, an entry in a tree chart or baseline diagram for the system at any point in time, or a shared section of data such as a COMMON block).

computer time	For batch usage, this is the billable time for all runs. For interactive usage, it is the number of hours spent at a terminal.
confidence level	Percentage probability that a given number is correct: 100 percent means that the number is absolute certainty; 0 percent means that the number must be incorrect.
constraints	Restrictions on resource availability (execution time, memory allocation) imposed by specifications.
constraints, space	All restrictions caused by space problems. On the Component Summary Report form, list each restriction separately, e.g., maximum number of words that component may occupy at one time or maximum disk space available during execution time or for program storage.
constraints, time	All restrictions caused by various machine and calendar time problems. On the Component Summary Report form, list each restriction separately, e.g., maximum execution time for component to process and respond to some input condition or time to complete a component or milestone.
control statements	All statements that potentially alter the sequence of executed instructions (e.g., GOTO, IF, RETURN, or DO).
correction	A change made to correct an error.
cosmetic	Changes in the source program that have little effect on the performance of program, e.g., correct comments, move code around as long as it does not alter the algorithm implemented, or change the name of a local variable.
create	The creation and recording of the idea.
creation date	Date that the component was first named (e.g., date it first appeared on a tree chart).
cross-reference	A list of the identifiers used by a program showing (by means of indices or statement numbers) which statements of the program define and reference those identifiers.

data base applications	This category is to include components that retrieve, write to, or format information for a well-defined formatted bank of information available to the system. The user must decide whether the data set is to be considered a data base or not. An example of an acceptable data base would be the ADL file, SLP file, or Geodetics file, whereas a sequential telemetry file or tape would not be.
design	A description of what the system must do, its components, the interfaces among those components, and the system's interface(s) to the external environment.
design phase	The creation and recording of the design, including discussion about strategy with peers. This phase does not include the development of any code at the programming language level. It does include the creation of specifications for subcomponents of the current component.
design reading	Visual inspection of the design by persons other than the creator of the design.
development phase	The development and recording of code and inline comments based on the design. This phase includes the modification of code caused by design changes or errors found in testing. It does not include any time spent in entering the code into the computer.
documentation	Written material, other than source code statements, that describes a system or any of its components.
dump	A record of the state of the memory space used by a program at some point in its execution. A dump may include all or part of the program's memory space (including registers).
end date	Date that a project is scheduled to be completed.
English (or informal) specifications	Specifications given as readable English text, as opposed to some formal notation.

error	A discrepancy between a specification and its implementation. The specification might be requirements, design specifications, or coding specifications.
external environment	The combination of hardware and software used to maintain and execute the software, including the computer on which the software executes, the operating system for that computer, support libraries, text editors, and compilers.
formal specifications	Some specification technique based upon a strict set of rules for describing the specification and usually involving the use of an unambiguously defined notation (e.g., mathematical functions or formal PDL).
function	A mathematical notation used to specify the set of input, the set of output, and the relationship between input and output.
functional specifications	A specification of a component as a set of functions defining the output for any input. The specification emphasizes what the program is to do rather than how to do it. However, an algorithmic specification can be considered functional if it is not used to dictate the actual algorithm to be used. (See procedural specifications.)
hardest first	The design (or implementation) of the most difficult aspects of the system first.
HIPO (Hierarchical Input Process Output)	A graphical technique that defines each component by its transformation on its input data sets to its output data sets.
implementation	The implementation of a program is either a machine-executable form of the program, or a form of the program that can be automatically translated (e.g., by compiler or assembler) into machine-executable form.
integration test	A test of several modules to check that the interfaces are defined correctly.
integration test, full	Test of the entire system (i.e., top-level component).

integration test, partial	Test of any set of modules but not the entire system.
intended use of	The result of invoking a program or segment of a program, including the actions performed by that program when invoked. Invocation may be by subroutine or function call or by a branch to a segment of code.
interface	The set of data passed between two or more programs or segments of programs and the assumptions made by each program about how the others operate.
interactive	Use of a computer via a terminal in which each line of input is immediately processed by the computer.
iterative enhancement	The design (or implementation) of successive versions, each producing a usable subset of the final product until the entire system is fully developed.
level	A unit corresponding to some partitioning of the final product (e.g., a single line of code, 10 lines of code, 25 lines of code, subroutine, or module). If the system is hierarchically structured, each component is at a higher level than its subcomponents, and the system may be described as the highest level component (the component at level 1), the component at level 2, or the lowest level component.
level, lowest	Smallest unit identified by the activity (e.g., code reading to the single statement, top-down design to the module level, or top-down design to level 3).
librarian	A clerk whose responsibilities include processing source statements but not writing them, (e.g., maintaining libraries, updating code, or producing tape backups).
machine words	Number of words in a main memory that a component occupies at one time.
manpower	The sum, over the number of people, of the number of hours per person charged to the contract.

mathematical/ numerical	This category is meant to be a more specific category than the scientific class. It contains those components that reflect a specific algebraic expression or mathematical algorithm. Such components as a dot product routine or a numerical integrator are in this category.
maximum space	Total number of machine words that the system may occupy at one time.
mission date	Date that system must be operational.
module test	Test of a single module.
none used	No explicit technique was specified to be used.
on-board processing	All components that are built for the purpose of satisfying some on-board processing need belong to this class. Although the component may be built and tested on a computer that is not the real flight computer, it should be classified as onboard if the final destination is the OBC (onboard computer).
optimization	Changes in the source code to improve program performance, e.g., run faster or use less space. Optimization changes are not error corrections; however, if a change is made to use less space to conform to the specified space constraint, then the term "error" applies.
PDL	A program design language (often called pseudocode). Used in the design and coding phases of a project, PDL is a language that contains a fixed set of control statements and a formal or informal way of defining and operating on data structures. PDL code may or may not be machine-readable, and for this study it is not considered as documentation, but as an integral part of the finished source program.

procedural specifications	A specification of a component in some algorithmic manner (e.g., using PDL or a flowchart). The specification says how the program is to work. (See functional specifications.)
proof technique	A method for formally demonstrating that a piece of software performs according to its specifications. Proof techniques usually use some form of mathematical notation to describe the result of executing a program.
range in module size	The number of source statements in a module, including comments.
read	The reading by peers of the recordings of the current phase to look for errors, invent tests, and so on.
real-time	This class includes components that are a direct function of events occurring at, or near, the current time. Typical components would be the Attitude Control Monitors. Since parts of most of the telemetry processors are required to process data as it is received, they too may be considered real-time components.
requirements	A system specification written by the user to define a system to a developer. The developer uses these specifications in designing, implementing, and testing the system.
review	A formal meeting of several individuals for the purpose of explaining design (management review). Also includes the time spent in preparing for the review. All those attending a review should list the components discussed in their own Component Summary Report for that week.
scientific	A component may be in this category if it is related to some mathematical algorithm, engineering problem, law of physics, or celestial mechanics problem. Most of the full systems developed will fall into this category, whereas the various pieces of modules may fall into some of the other classes.

segment	A contiguous piece of code that is unnamed and, hence, cannot be referred to as a single entity in a program statement. A segment could be one or several lines of a subroutine, part of a data area, or an arbitrary contiguous section of memory.
shared items	Data and programs, accessible by several components, such as COMMON blocks, external files, and library subroutines.
simulating constructs	Statements that are used to simulate structured control structures when the language to be used does not contain structured control structures.
source instructions	See source statements.
source statements	All statements readable by and read by the compiler. This includes executable statements (e.g., assignment, IF, and GO TO); nonexecutable statements (e.g., DIMENSION, REAL, and END); and comments.
specification	A description of the input, output, and essential function(s) to be performed by a component of the system. The specification is produced by the organization that is to develop the system; that is, at the top level, it can be thought of as the contractor's interpretation of the requirements.
specification, imprecise	The input, output, and function of the component are loosely defined. Much of what is required is assumed rather than specified. The specification relies heavily on programmer experience and verbal communication to get an unambiguous interpretation and a full understanding of what is needed.
specification, precise	The input, output, and function of the component are well defined. There are underlying assumptions not specified, but it is assumed that any programmer working on the project, with experience on a similar project, will understand these assumptions. It is possible to arrive at an ambiguous interpretation or misunderstanding

specification, precise (Cont'd)	of the specifications if the reader does not have enough experience with the problem or does not obtain further verbal communication.
specification, very precise	A completely defined description of the input, output, and function of a component. The implementer of a very precise specification need make few, if any, assumptions. It is almost impossible to arrive at an ambiguous interpretation or misunderstanding of the specifications.
specification-driven	Using the specifications of the program to determine test data (e.g., test data is generated by examining the input/output requirements and specifications).
standards	Any specifications that refer to the method of development of the source program itself, and not to the problem to be implemented (e.g., using structured code, at most 100-line subroutines, or all names prefixed with subsystem name).
start date	Date on which initial work on a project began.
string processing	This includes components that perform operations on lists of characters. Normally, this class is assumed to include functions of compilers, hash code string hook-up, and array comparisons.
structure-driven	Using the structure of the program to determine test data (e.g., generating data to ensure that each branch of a program is executed at least once).
structure of data	The organization of a composite data item consisting of several variables or other array items. Examples of such composite data items are arrays (both singly- and multiply-dimensioned), strings, complex variables and constants, records on a disk file (each record containing several words), and multiple-word entries in a table.
structured code	The language supports structured control structures (e.g., a FORTRAN preprocessor).

systems	By system-related software, one includes any package designed to affect, modify, extend, or change the normal available processing procedure of the operating system. This could include such components as error tracing or extended I/O such as DAIO.
system size	Total number of machine words needed for all instructions generated on the project plus space for data, library routines, and other code. This is the total size of the system without using any overlay structure.
table handler	Includes components that are specifically designed to generate or interpret information in a table format such as the Generalized Telemetry Processor.
telemetry/ tracking	Includes all components that are specifically required to interface (either read, write, or format) with telemetry or tracking data.
testing phase	The design of tests, testing strategies, and the running of such tests. This phase does not include the writing of any code (even for debugging purposes), which should be recorded under coding.
top-down	The design (or implementation) of the system, starting with a single component, one level at a time, by expanding each component reference as an algorithm possibly calling other new components.
trace	A record of program execution showing the sequence of subroutine and function calls and, sometimes, the value of selected variables. Code used in producing a trace is automatically inserted into a program, usually by the compiler, sometimes by other support software.
type of software	The four major classifications of most of the applicable software being developed are: scientific, business/financial, systems, and utility. These classifications may be refined into the categories of: string processing, data base applications, real time, and table

type of software (Cont'd)	handler. A further refinement includes the catagories of: attitude/orbit, telemetry/tracking, command/control, mathematical, and numerical on-board.
utility	Any component that is generated to satisfy some general support function required by other applications software may be considered a utility. One thinks of this class of components as containing software that does not fit into any of the other three categories. Although components can fall into two of the primary categories (e.g., scientific and utility), it will be easier to use only the more descriptive of the categories (e.g., vector cross product--scientific; data unpacking--utility).
value of data	The number and kind of number (e.g., integer, floating-point, or ASCII-encoded character) stored in a local variable or data area, parameter, common variable, or system-wide data item.
walkthrough	Formal meeting sessions for the review of source code and design by the various members of the project for technical rather than management purposes. The purpose is for error detection and not correction.
workaround	The method used to counteract the effects of an error in a program when the cause of the error and, consequently, the location of the statements containing the error is not known or is inaccessible (e.g., a compiler error).

REFERENCES

1. P. Naur, B. Randell, and J. N. Buxton (eds.), Software Engineering: Concepts and Techniques. New York: Petrocelli/Charter, 1976

2. V. R. Basili, M. V. Zelkowitz, F. E. McGarry, R. W. Reiter, W. F. Truszkowski, and D. L. Weiss, The Software Engineering Laboratory SEL-1, TR-535, University of Maryland, 1977

3. V. R. Basili, "Data Collection, Validation, and Analysis," IEEE Tutorial on Software Engineering and Management, IEEE Computer Society, Fall 1980

4. Data and Analysis Center for Software, SRR-1, Quantitative Software Models, 1979

5. M. H. Halstead, Elements of Software Science. North Holland: Elsevier, 1977

6. A. Fitzsimmons and T. Love, "A Review and Evaluation of Software Science," Computing Surveys, March 1978

7. T. J. McCabe, "A Complexity Measure," IEEE Transactions on Software Engineering, June 1975

8. Stanford University, BMDP User's Guide

9. SAS Institute, Statistical Analysis System (SAS) User's Guide

10. SPSS Inc., SPSS-11 User's Guide

11. Computer Sciences Corporation, CSC/SD-81/6079, Software Engineering Laboratory (SEL) Data Base Maintenance System (DBAM) User's Guide and System Description, D. N. Card, September 1981

12. --, CSC/SD-81/6011UD1, Software Engineering Laboratory (SEL) Data Base Organization and User's Guide, D. C. Wyckoff, September 1981

13. QED Information Sciences, Data Base Systems: A Practical Reference, R. Palmer

14. Goddard Space Flight Center, Software Engineering Laboratory (SEL), A Meta-Model of Software Development Resource Expenditures (SEL internal report), V. R. Basili, and N. Bailey; also Proceedings, Fifth International Conference on Software Engineering, IEEE, 1981

15. RCA, Price S (system description), 1979

16. L. H. Putnam, "A General Empirical Solution to the Macro Software Sizing and Estimating Problem," IEEE Transactions on Software Engineering, July 1978

17. A. B. Endres, "An Analysis of Errors and Their Causes in System Programs," IEEE Transactions on Software Engineering, June 1975

18. B. Littlewood, "Theories of Software Reliability," IEEE Transactions on Software Engineering, September 1980

19. Data and Analysis Center for Software, GLOS-1, The DACS Glossary, A Bibliography of Software Engineering Terms, October 1979

BIBLIOGRAPHY OF SEL LITERATURE

Anderson, L., "SEL Library Software User's Guide," Computer Sciences-Technicolor Associates, Technical Memorandum, June 1980

Bailey, J. W., and V. R. Basili, "A Meta-Model for Software Development for 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

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

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 1980

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

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

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

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

Basili, V. R., and T. Phillips, "Validating Metrics on Project Data," University of Maryland, Technical Memorandum, December 1981

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

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

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

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

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

Church, V. E., "User's Guides for SEL PDP-11/70 Programs," Computer Sciences Corporation, Technical Memorandum, March 1980

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)

Mapp, T. E., "Applicability of the Rayleigh Curve to the SEL Environment" (paper prepared for the University of Maryland, December 1978)

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

Perricone, B. T., "Relationships Between Computer Software and Associated Errors: Empirical Investigation" (paper prepared for the University of Maryland, December 1981)

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

Software Engineering Laboratory, 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, D. S. Wilson, and R. Beard, 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, January 1978

--, 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-78-003, 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, B. Chu, and G. Page, 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, November 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, S. R. Waligora, and A. L. Green, 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 F. E. McGarry, September 1979

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

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

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

--, SEL-80-003, Multimission Modular Spacecraft Ground Support System (MMS/GSS) State-of-the-Art Computer System/ Compatibility Study, T. Weldon, M. McClellan, P. Liebertz, et al., May 1980

--, 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-80-005, 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

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

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

--, SEL-81-003, Software Engineering Laboratory (SEL) Data Base Maintenance System (DBAM) User's Guide and System Description, D. N. Card, D. C. Wyckoff, G. Page, et al., September 1981

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

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

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

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

--, SEL-81-008, 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, A. L. Green, and F. E. McGarry, March 1981

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

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

--, SEL-81-012, Software Engineering Laboratory, 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

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

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

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

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

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

specification, precise (Cont'd)	of the specifications if the reader does not have enough experience with the problem or does not obtain further verbal communication.
specification, very precise	A completely defined description of the input, output, and function of a component. The implementer of a very precise specification need make few, if any, assumptions. It is almost impossible to arrive at an ambiguous interpretation or misunderstanding of the specifications.
specification-driven	Using the specifications of the program to determine test data (e.g., test data is generated by examining the input/output requirements and specifications).
standards	Any specifications that refer to the method of development of the source program itself, and not to the problem to be implemented (e.g., using structured code, at most 100-line subroutines, or all names prefixed with subsystem name).
start date	Date on which initial work on a project began.
string processing	This includes components that perform operations on lists of characters. Normally, this class is assumed to include functions of compilers, hash code string hook-up, and array comparisons.
structure-driven	Using the structure of the program to determine test data (e.g., generating data to ensure that each branch of a program is executed at least once).
structure of data	The organization of a composite data item consisting of several variables or other array items. Examples of such composite data items are arrays (both singly- and multiply-dimensioned), strings, complex variables and constants, records on a disk file (each record containing several words), and multiple-word entries in a table.
structured code	The language supports structured control structures (e.g., a FORTRAN preprocessor).