Technical Report TR-535
SEL-1

May, 1977
NASA-NSG-5123

THE SOFTWARE ENGINEERING LABORATORY*

Victor R. Basili[1], Marvin V. Zelkowitz[1]
Frank E. McGarry[2], Robert W. Reiter[1]
Walter F. Truszkowski[2], David L. Weiss[1,3]

1 - Department of Computer Science, University of Maryland, College Park, Md.

2 - NASA Goddard Space Flight Center, Greenbelt, Md.

3 - Naval Research Laboratory, Washington, D.C.

*Research supported in part by National Aeronautics and Space Administration grant NSG-5123 to the University of Maryland.

THE SOFTWARE ENGINEERING LABORATORY

Victor R. Basili
Marvin V. Zelkowitz
Frank E. McGarry
Robert W. Reiter
Walter F. Truszkowski
David L. Weiss

Abstract

The development of techniques to produce cost-effective reliable software first requires the collection of quantitative and qualitative data on the development process. Towards this end, the Software Engineering Laboratory has been organized in conjunction with NASA Goddard Space Flight Center. The purpose of the Software Engineering Laboratory is to monitor existing software methodologies and develop and measure the effectiveness of alternate methodologies.

Initially, three aspects of the software development life cycle are to be investigated. These are: (1) management aspects in estimating team organization, resource requirements, schedules and reliability factors in the finished software product, (2) error characteristics and their causes, and (3) program structure and its relation to well-developed software.

I. Introduction

A great deal of time and money has been and will continue to be spent in developing software. Much effort has gone into the generation of various software development methodologies that are meant to improve both the process and the product ([MYER, 75], [BAKE, 75]). Unfortunately, it has not always been clear what the underlying principles involved in the software development process are and what effect the methodologies have; it is not always clear what constitutes a better product. Thus progress in finding techniques that produce better, cheaper software depends on developing new deeper understandings of good software and the software development process through studying the underlying principles involved in software and the development process. At the same time we must continue to produce software.

To gain a better knowledge of what is good in the current methodologies and what is still needed, and to help understand the underlying principles of the software development process, we must analyze current techniques, understand what we are doing right, understand what we are doing wrong, and understand what we can change.

There are several ways of doing this. One way is to analyze the development process and the product at various stages of development. Unfortunately, such analysis is a tedious process. But it must be performed if we are to gain any real insight into the problems of software development and make improvements in the process. We need to study carefully the effect of various changes in the development process or the product to determine whether or not a particular methodology has any real effect, and more importantly, what kind of effect ([THAY, 76][WALS, 77]).

This requires measures of all kinds, quantifiable and nonquantifiable. Nonquantifiable measures, although subjective, reveal a great deal of information about the product. We can "see" good design and code that meets the prob-

lem requirements in a clear, understandable, effective way and is easy to modify and maintain in unforeseen circumstances. This kind of understanding is clearly needed, and clearly fruitful; it is accomplished by reading and understanding the design and code. Unfortunately, these judgements are not easy to quantify. They require a great deal of time to analyze and measure each product, or class of products.

A secondary approach is to develop a set of measures that attempt to quantify these qualitative characteristics of good software design and development. Although there is currently no mechanical way of evaluating design, the development of quantitative measures that correlate well with subjective judgements of quality can aid in the understanding and evaluation of the product and process. For example, the "goodness" of a product is related to the time it takes to modify it and the aspects of its organizational structure that permit ease of modification and ease of finding and correcting errors where ease is measured in terms of the time required, number of places code needs to be changed, etc. The "goodness" of the development methodology is related to the "goodness" of the product it produces, e.g., the number and difficulty of finding errors in the product it produces.

It is important to understand what characterizes classes of problems and products, what kinds of problems are encountered and errors made in the development of a particular class of products, whether or not a particular methodology helps in exposing or minimizing the number or effect of a class of errors, what the relationship is between methodology and management control, estimating, etc. A better understanding of the factors that affect the development of software and their interrelationships is required in order to gain better insights into the underlying principles. The Software Engineering Laboratory has been established at NASA Goddard Space Flight Center in coopera-

tion with the University of Maryland to promote such understanding. The goals of the laboratory are to analyze the software development process and the software produced in order to understand the development process, the software product itself, the effect of various "improvements" on the process with respect to the methodology, and to develop quantitative measures that correlate well with intuitive notions of good software.

The next section gives an overview of the research objectives and experiments being performed at the Laboratory. Section III contains the current list of factors that affect the software development process or product and are to be studied or neutralized. The data collection and data management activities are discussed in Section IV and Section V discusses the research objectives in greater detail. The last section contains information on the current status and future plans for the Laboratory.

Appendix 1 contains a list of personnel associated with the Software Engineering Laboratory. Appendix 2 contains the data collection forms and their associated instructions that are used for all projects being studied. Appendix 3 contains a description of the structure of the data base of information collected on the projects and Appendix 4 describes the development environment at NASA Goddard Space Fligth Center and the details of all the projects currently under study.

II. Activities

It is clear that many kinds of data can be gathered and analyzed to develop quantitative information about the software process and the product that it leads to. The laboratory has limited funding and personnel and for this reason has limited its scope to studying three very specific areas related to reliability, management, and complexity. It is expected that the scope will eventually expand as we learn more about the collection of valid data and what can be done with it. The data collection methods and the storage and retrieval system are described in Section IV. In this section we discuss the research activities and the two classes of experiments to be run.

The research objectives can be divided into three basic areas: reliability, management and complexity. Because error-free software is as yet an unattainable goal, the reliability study will provide insight into the nature and causes of software errors. We would like to classify errors, expose techniques that reduce the total number or classes of errors, and detect the effect or lifetime of these errors. We expect to detect the point at which errors enter the process and the relative costs of finding and fixing them.

Management of the software development process is as poorly understood as the technology involved. We believe that a major effort should be expended on this area. The management aspect of the Software Engineering Laboratory involves the analysis of the management process, the classification of projects from a management point of view and the development of reasonable management measures for estimating time, cost, and productivity. We will study the effect of various factors, such as time, money, size, computer access, techniques, tools, organization, standards, milestones, etc. We would like to understand at what point in the development process, estimates become reasonably accurate, how one can measure good visibility and management control and

under what conditions certain methodologies help provide management control.

Lastly, there is a relation between the development methodology and the product it produces. A good methodology should help produce a less complex product than a "bad" one. We are trying to discover whether the complexity of a software system can be measured by the structure of the resulting programs. Do various techniques create a more systematic structure, one that is easier to read and maintain, where data and function are localized with a minimal amount of interaction between modules? The rleationship between various complexity measures of program structure will be examined throughout the development process and such measures as error rate, development time, the accuracy and speed of modification, etc., will be correlated with these complexity measures.

Two kinds of experiments will be conducted: screening experiments and controlled experiments. In the screening experiments, we are collecting data on a large assortment of projects of varying sizes and types. The impact on the development process is manifested by the requirement that the developers fill out a set of data collection forms (see Section IV). The purpose of the screening experiments is to determine how software is developed now. We are organizing a data bank of information to classify projects for future reference and public availability, analyze what methodologies are being used as opposed to what methodologies are supposed to be used, demonstrate how carefully the actual implementation of a methodology can be monitored, discover what parameters can be validly isolated, expose the parameters that appear to be causing major problems, and discover the appropriate milestones and techniques that show success under certain conditions. While the data collected in the screening experiments may not be complete or totally accurate, it will provide input for the more strictly monitored controlled experiments.

The purpose of the controlled experiments is to discover the effect of various factors on the software development process and product in a reasonably controlled environment. A set of duplicate developments will be performed and detailed data collected for all of them. A carefully chosen set of techniques will be taught to and used by one of the development groups, denoted as the "impacted" group. We will then analyze the effect of the introduced factors by comparing the impacted development process and product in a reasonably controlled environment.

The experiment must be designed in such a way as to insure that we are testing the real hypothesis, i.e., to guarantee that we are measuring what we think we are measuring. It is important that all the contributing factors be well understood and the factors that we are not studying be neutralized [CAMP, 66]. Our approach is first to develop a particular experimental design, analyze its ability to neutralize potential interfering factors, (i.e., individual programmer capability) and perform one experiment. Based on this experience, the design will be modified and experiments repeated until we have arrived at a reasonable standard.

One current experimental design is to have two groups, Group 1 and Group 2, each develop a product, A. We will then impact Group 2 with a set of factors by teaching them the use of certain development techniques. Both groups will then develop a second small project B to give Group 2 some experience with the techniques in an operating environment. Then both groups will develop product C, Group 2 using the new approach. This gives us several points of comparison. We can discover any difference in personnel by comparing project A for both groups; the two groups can then be more honestly compared in project C by factoring out differences from project A. The measures developed for the areas of interest will be used to compare the two processes and products.

In a second controlled experiment, several large scale projects (5 to 10 man years each) are to be carefully monitored with some of the personnel given a training course and set methodology to use. Using the notation above, these will be a set of C projects with no A and B. While the projects are not identical, they are highly similar and should yield information about differences in techniques. In Section VI, both of these controlled experiments will be described in greater detail.

III. Factors

There are a large number of factors that affect the software development process and software product. Initially, we are interested in a list of potential factors to establish the kind of data that needs to be collected. Next, we are interested in the kinds of factors that we can reliably measure. From this measurable set of factors, we would like to isolate those that appear to have a major inpact on the development process and product, i.e., those whose use or non-use show large variation in our measures. Finally, when we have a better understanding of the factors affecting the software development process, we want to quantify them in some way by perturbing them to study their effects or neutralizing them to make sure they are not affecting factors that are under study.

Our procedure is to start with as complete a list of factors and categories of factors as possible. We expect continally to build, iterate, and refine this list through the activities of the laboratory. The development of reporting forms and automated tools have helped define the list of factors that we can isolate. The screening experiments will help further isolate those factors which we can measure and those that appear to be contributing strongly to the various measures associated with errors, complexity of program structure, management difficulties, etc. The controlled experiments will be used to demonstrate the effect of the various factors that have been shown worth isolated study.

A list of factors is given below, categorized by their association to the problem, the people, the process, the product, the resources, and the tools. Some factors may fit in more than one category but are listed only once.

A. People Factors: These include all the individuals involved in the software development process including managers, analysts, designers, pro-

grammers, librarians, etc. People related factors that can affect the development process include:

number of people involved

level of expertise of the individual members

organization of the group

previous experience with the problem

previous experience with the methodology

previous experience with working with other members of the group

ability to communicate

morale of the individuals

capability of each individual

B. Problem Factors: The problem is the application or task for which a software system is being developed. Problem related factors include:

type of problem (mathematical, database manipulation, etc.)

relative newness to state of the art requirements

magnitude of the problem

susceptibility to change

new start or modification of an existing system

final product required, e.g., object code, source, documentation, etc.

state of the problem definition, e.g., rough requirements vs. formal specification

importance of the problem

constraints placed on the solution

C. Process Factors: The process consists of the particular methodologies, techniques, and standards used in each area of the software development. Process factors include:

Programming Languages

Process Design Language ([VANL, 76])

Specification Language

Use of librarian ([BAKE, 75])

Walk-throughs ([BAKE, 75])

Test Plan

Code Reading

Top Down Design

Top Down Development (stubs)

Iterative Enhancement ([BASI, 76])

Chief Programmer Team ([BAKE, 75])

Chapin Charts

HIPO Charts ([STAY, 76])

Data Flow Diagrams

Reporting Mechanisms

Structured Programming ([MILL, 72], [DAHL, 72])

HOS Techniques ([HAMI, 76])

Milestones

D. Product Factors: The product of a software development effort is the software system itself. Product factors include:

deliverables

size in lines of code, words of memory, etc.

efficiency tests

real-time requriements

correctness

portability

structure of control

in-line documentation

structure of data

number of modules

size of modules

connectivity of modules

target machine architecture

overlay sizes

E. Resource Factors: The resources are the nonhuman elements allocated and expanded to accomplish the software development. Resource factors include:

Target machine system

Development machine system

Development software

Deadlines

Budget

Response times and turnaround times

(Note there is a relationship between resource and product factors in that the resources define a set of limits within which the product must perform. Sometimes these external constraints can be a dominating force on the product and sometimes they are only a minor factor, e.g., it is easy to get the product to perform well within the set of constraints.)

F. Tool Factors: The tools, although also a resource factor, are listed separately due to the important impact they have on development. Tools are the various supportive automated aids used during the various phases of the development process. Tool factors include ([REIF, 75],[BOEH, 75], BROW, 73]):

Requirements analyzers (e.g., PSL/PSA) [TEIC, 77]

System design analyzers

Source code analyzers (e.g., FACES [RAMA, 74])

Database systems (e.g., DOMONIC [DOMO, 75])

PDL processors

Automatic flowcharters

Automated development libraries

Implementation languages

Analysis facilities

Testing tools ([RAMA, 75], [MILL, 75])

Maintenance tools

IV. Data Collection

Data collection occurs as four components - reporting forms, interviews, automatic collection of data by the computer system, and use of automated data analysis routines.

A. Forms: The seven forms that appear in Appendix 2 were defined to obtain information on several of the factors given in Section III. These forms are filled out by various members of the project development team and are used to gather information at various states of the development process. They reveal the resource estimates at inception, the overall layout of the system, the updating of the estimates and the achievement of milestones, the time spent in various activities, the expenditures of resources, and an audit of all changes to the system. Several redundancy checks have been included to validate the accuracy of the information obtained.

Briefly, the seven forms are as follows:

1. The General Project Summary - 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 is filled out by the project manager at the beginning of the project, at each major milestone, and at the end. The final report should accurately describe the system development life cycle.
2. The Programmer/Analyst Survey - This form is to classify the background of the personnel on each project. It is filled out once at the start of the project by all personnel.
3. 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 is 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 is made. It is filled out by the person responsible for that component.

4. The Component Status Report - This form is used to keep track of the development of each component in the system. The form is turned in at the end of each week and for each component lists the number of hours spent on it. This form is filled out by persons working on the project.
5. The Resource Summary - This form keeps track of the project costs on a weekly basis. It is filled out by the project manager every week of the project duration. It should correlate closely with the component status report.
6. Change Report - The change report form is filled out every time the system changes because of change or error in design, code, specifications or requirements. The form identifies the error, its cause and other facets of the project that are affected.
7. Computer Program Run Analysis - This form is used to monitor the computer activities used in the project. An entry is made every time the computer is used by the person initiating the run.

B. Interviews: Interviews are used to validate the accuracy of the forms and to supplement the information contained on them in areas where it is impossible to expect reasonably accurate information in a form format. In the first case spot check interviews are conducted with individuals filling out the forms to check that they have given correct information

as interpreted by an independent observer. This would include agreement about such things as the cause of an error or at what point in the development process the error was caused or detected.

In the second case, interviews will be held to gather information in depth on several management decisions, e.g., why a particular personnel organization was chosen, why a particular set of people was picked, etc. These are the kinds of questions that often require discussion rather than a simple answer on a form.

C. Automatic Data Collection: The easiest and most accurate way to gather information is through an automated system. Throughout the history of the project, more and more emphasis will be placed on the automatic collection of data as we become more aware what data we want to collect, i.e., what data is the most valuable and what data we can or need to get, etc. More energy will be expended in the development or procurement of automatic collection tools as the laboratory continues.

The most basic information gathering device is the program development library. The librarian will automatically record data and alleviate the clerical burden from the manager and the programmers. Copies of the current state of affairs of the development library will be periodically archived to preserve the history of the developing product.

A second technique for gathering data automatically is to analyze the product itself, gathering information about its structure using a program analyzer system. A set of modifications to the FACES system is currently underway to and will progress as the laboratory gains more experience. These modifications are geared at getting more of the kind of information about the product requires for our measures.

D. Database analysis: The above data collected on the project will be stored in a computerized database. Data analysis routines will be written to

collect derived data from the raw data in the database. The data that is being collected is to be processed by PDP11-based system that is nearing completion. For ease of implementation, it utilizes the INGRES relational database system [HELD, 75] which runs under the UNIX operating system.

The data that is collected on the reporting forms will either be encoded onto magnetic tape via CRT terminals at NASA/GSFC or else entered into a file on the Univac 1100/40 at the University of Maryland. The magnetic tape (or Univac file) will then be moved to the PDP11 in the Department of Computer Science at the University of Maryland and stored as a file under UNIX.

After archiving, the next stage is to validate the encoded data. A table-driven program, written in the systems programming language C, has been developed that reads in the raw data, validates the entries, and if correct, enters it into an INGRES file. The data describing the forms is read by the program as a set of tables. These tables are direct analogs of the data on the paper forms and thus allow for easy modification should the forms need to be altered. (See Appendix 3)

Software to perform all of the above has been written and is now under test. The next level of software is now being designed. Initially, English queries can be posed to INGRES in order to retrieve certain fields within the database, and these retrieved records can be converted back into a UNIX file. This UNIX file can then be printed or read by a C or FORTRAN program in order to be processed.

The goal of the database implementation, however, is to automate this process as much as possible. The QUEL query language exists to interface between a C program and INGRES. A system is now being designed that will utilize this language so that data can be retrieved directly by the analysis programs without the need for operator intervention.

V. Current Investigations

The Software Engineering Laboratory is currently emphasizing three general areas in software development. These three areas are as follows:

1. Management

Management may be defined as the deliberate and judicious use of resources to accomplish an objective. In the software development context the resources refer to such entities as people, time, dollars, computer systems and peripherals, programming methodologies and standards, languages, and the like. The manager, given a project or a problem to be solved, manipulates these resources within a management structure and according to some master stratagem, to evolve the ultimate objective - a piece of quality software.

The goal of this management study within the context of the Software Engineering Laboratory is two-fold:

a. to investigate the various management techniques, currently in vogue, which are the driving forces behind the projects studied under the Lab's auspices and, by careful analysis of the principles in play, come to a fuller appreciation of them noting both their successes and failures.

b. to help evolve and develop a software engineering methodology (or methodologies as the case may be) for effective management of software projects. The overall goal, in this respect, is to devise a set of management practices and principles which highlights and maximizes the successes while minimizing the failures of currently used management techniques.

In order to achieve the goals of this investigation the following plan will be followed. The management of a project will be viewed as a management function which relates resources and quality

software products. The internal workings of this function will be deciphered in terms of its composing sub-functions, i.e., planning, organizing, staffing, coordinating, directing, and controlling. Each of these sub-functions will be given an unambiguous meaning in the framework of the software development process and will be discussed in terms of the three basic levels of management, i.e., strategic, tactical, and operational - with emphasis being placed on the operational level. The definition or modelling of these sub-functions of the mamagement function will be done in such a way that quantitative measures for each of them can be devised. These measures will be used in the evaluation of the management techniques which are applied to the projects studied by the Lab.

The measures will deal with such, often elusive, quantities as time/cost tradeoffs, milestone prediction, resource estimation and allocation, project status and visibility, and productivity.

Several sets of management measures have appeared in the open literature. As examples of the types of quantitative measures that can be used, aspects of the work of Tausworthe [TAUS, 76] and Baumgartner [BAUM, 63] will be reviewed.

Tausworthe's set revolves around the concept of 'Index of Productivity' P which is defined by $P = L/(WT)$ where L is the total number of error-free source code lines excluding comments, W is the number of programmers, and T is the average time each worker spent in the software development effort. The units of this productivity index are lines/day. Tausworthe expands on the concept T by setting $T = T_P + (W-L)\, T_{NP}$ where T_P equals average productive time, and T_{NP} represents time spent interfacing with other team members. Then an index of individual productivity becomes $P_I = L/T_P$. He notes that

if $T_{NP}/T = 1/(W-1)$ then the overall productivity P will not be achieved and the project will fail. Additionally Tausworthe developed measures for job integration.

Baumgartner's work involves the concept of Status Index. Letting P represent progress, SP represent scheduled progress, B-budget, and AE-actual expenditures, then the Status Index SI equals (P/SP)(B/AE). This index can be used to relate actual progress and costs with the budget and milestone predictions in the project plan. If SI = 1 then all is well, if SI < 1 then this indicates less than expected progress, and if SI > 1 then performance has exceeded expected progress. Careful use of this index technique can provide information to the manager regarding time/cost performance, time/cost projections, anticipated schedule slippage, and can allow him to rank critical problem areas and redirect resources.

Note that in the examples cited the following types of information were required - code length, number of programmers, time, milestones, budget and actual expenditures.

Though the exact measures and statistical tests, which will be used in this particular management study, have yet to be fully determined, the data which will be evaluated metrically and statistically to yield management information will primarily be extracted from the data collection forms established by the Lab (see Appendix 2)

All of the forms contain data which contribute to overall project visibility for the manager. However, three are of particular importance. These are General Project Summary, Resource Summary, Programmar/Analyst Survey. These forms solicit information regarding resources, time, cost, size, milestones, manpower hours, computer usage, and experience of team members.

An underlying hypothesis of this management study is that one may devise a highly quantifiable model of the management process as it relates to software development. Most managers, though acquainted with some principles and guidelines for management, more often than not resort to personalized heuristics in their attempt to achieve quality software. The management data, collected on the forms, along with more personalized interviews will help divine these 'rules of thumb'. At this point the existing management model may be managed to incorporate those with proven success. This is a very tentative area and one which poses special problems. It may be that some heuristics, which lead to qualitative judgements, cannot easily be quantified if at all. At this point only time will tell.

It is anticipated that this phase of the Laboratory's work will proceed through the process of iterative refinement. With each iteration and refinement a clearer more cohesive model relating to the management of the software development process will evolve.

2. Errors

According to the folklore of computer science, the first program ever written to run on a digital computer contained an error. Since then, an often discussed, but rarely achieved, goal of programmers is to produce programs that are error free. Although it would be of interest to know the circumstances surrounding that first error, and the reasons it occurred, such information was apparently never carefully recorded. Unfortunately, few analysts, designers, programmers, etc., ever bother to record their errors and the reasons for them. As a result, there is little agreement in the software community as to what constitutes an error, what the most common kinds of errors are, or why software errors occur.

For the purposes of the Software Engineering Laboratory, our working definition of a reliable system is one in which its requirements accurately reflect the needs of its users and the demands of its environment, and if it performs as specified by its requirements. We then separate errors into two classes: errors in requirements, i.e., those cases in which a change must be made in requirements (and consequently their implementation) for the system to operate as desired, and errors in implementation of requirements. The latter case might involve incorrect specifications, incorrect design, improper translations of design into code, or incorrect documentation.

Most previous studies on errors either involved small university projects or else studied only a small class of errors, thus there is little historical data available that permits careful study of the way that software is developed ([SHOO, 75], [THAY, 76], [ENDR, 75], [GANN, 75], [LITE, 76], [AMOR, 73]). The main goal of this study is to gain insight into the software development process. In particular the objectives of the reliability study are

- to develop methods for accurate measurement of errors,
- to discover the kinds of errors that are prevalent in the software under study,
- to discover the principal causes of errors in the software under study, and
- to evaluate potential error-prevention techniques.

Since studies of software errors depend on information furnished by human beings about themselves and the kinds of mistakes they make, it is extremely important to develop valid, consistent techniques for gathering error data and estimating its accuracy. (A valid technique yields data on the desired subject, i.e., the object mea-

sured is really the one the experimenter thinks he measured. A consistent technique yields reproducible results, i.e., the same data will be obtained in the same circumstances when the experiment is repeated.) Such techniques can be used by other investigators to repeat the measurements to provide comparable data. Only in this way can a large body of useful data be accumulated by the software engineering community.

Because software errors are unique, i.e., once an error is corrected it does not recur in the corrected system, meaningful patterns of error occurrences can only be found by placing errors with similar attributes into a single category. As an example, one might place all errors resulting from an improperly specified interface into the category of interface errors. (Note that some of these errors might also be placed into other categories, depending on what other attributes they have.) The categories selected depend on the attributes of errors that are of interest to the investigator. The initial categorizations of errors to be investigated in the laboratory studies are relatively broad, and will be refined as results are obtained from the screening and controlled experiments. A prime purpose of the screening experiments from the reliability viewpoint is to discover into which of the broad categories most errors tend to fall.

The initial error classification scheme to be used in the reliability studies is listed below. The categories are intended to be inclusive but not exclusive, i.e., some errors will fall into more than one category. Errors will be categorized according to:

- whether or not they are caused by a misunderstanding, and if so whether it was a misunderstanding of requirements, functional specifications, design, language, hardware environment, software environment, or some other factor
- the time at which they entered the system

- modifications to the system that generated them
- activities used for error detection
- time required to find the cause
- time required to design the correction
- whether or not they are clerical errors

The above list will be refined, categorizations added, and categories split into subcategories as more insight is gained into the software development process.

The approach we are taking in the reliability study is that of strong inference - hypotheses are formulated, experiements are performed to provide data to confirm or deny the hypotheses, and new hypotheses are generated. The starting point for the process consists of hypotheses taken from the software engineering literature, the experimenter's background and intuition, and the data provided by the screening experiments.

The error data collection process is based on the assumption that once past the initial design and coding stages, every event of significance to the development process is reflected in a change to the system. Changes can occur in requirements, specifications, design, code, the hardware environment, or the software environment. A form, called the change report form has been designed to provide information on each change made to the system. The form is the central instrument in the collection of data for the reliability study. It yields information concerning the reason for the change, the place(s) where the change is made, a description of the change, the time required to design the change, the cause of the change, the effects of the change, the method used to decide that the change was needed, whether or not an error is associated with the change, and, if an error is involved, information used to classify the data according to the scheme described previously.

The data, once collected, will be used to construct a historical record of the system development, including the relation of various events and changes to the system to each other and the errors that occurred, to construct distributions of errors, and to test hypotheses concerning the effects of different factors on error rates and error distributions. Some of the error distributions that will be constructed are:

- errors distributed according to project phase (requirements, specifications, design, coding, etc.) in which they occur
- errors distributed according to technique used to detect them
- errors distributed according to time required to detect and correct them
- errors distributed according to the type of misunderstanding associated with them (no misunderstanding, misunderstanding of requirements, specifications, design, programming language, hardware environment, software environment)
- errors distributed according to modification associated with them (modification to correct a previous error, modification not to correct an error)
- errors distributed according to the named component in which they occurred
- errors distributed according to the amount of time they remained in the system

Many of the above distributions are obtainable from the screening experiments. Furthermore, it is expected that the results of the screening experiments will suggest other distributions that will be of interest to construct.

Hypotheses of interest related to the effects of different factors on error distributions and rates will generally be investigated in conjunction with the controlled experiments. The controlled experiments, consisting of the planned introduction of integrated methodologies for software development during the course of a series of development projects, will be used for hypothesis testing. Most of these hypotheses will be concerned with the effects of using different methodologies on the error distributions previously described. The hypotheses will generally be of the form "The use of methodology X has a significant effect on the distribution of errors according to Y ", where "X" is a particular methodology used in a controlled experiment, and Y is a description of one of the error distributions previously listed. As error distributions are constructed for different methodologies, they will be compared to distributions obtained during other controlled experiments.

It is not possible to conduct a long term experiment measuring creative human activities without problems of data collection. Analysis and interpretation apppearing. Some of the expected problems are described in the following.

The type of investigation described here is quite sensitive to the effects of confounding variables. It is extremely important that factors such as programmer background and experience, application area of the system being developed, organization of the development team, etc., must either be specifically controlled or neutralized.

The data collected must be monitored continually throughout the project for validity and consistency because it sometimes requires subjective judgments on the part of the people completing the Charge Report forms.

It may not be possible to categorize precisely a number of the errors observed, or to associate them with previous changes to the software. This can occur if the causes of errors cannot be determined well enough to provide the necessary information, or cannot be determined at all, or if the forms are not completed in a timely way.

3. Complexity

Software complexity can be defined as the difficulty of human comprehension of a software system's organization and operation (i.e., the degree of mental effort required for this comprehension). Since the programs constitute a significant part of a software system, it is assumed that a significant component of overall software complexity may be found by isolating program complexity, the difficulty of comprehending the system's source code's organization and operation. [This research is based on the beliefs that software complexity is one of the major barriers presently precluding the achievement of quality software today (c.f., [MILL, 73]), and valid understanding of the nature of software complexity is fundamentally necessary before the barrier can be overcome.]

There is however a real scarcity of knowledge (even raw data) regarding program complexity; what little information there presently is has been largely subjective and qualitative. An important method in the pursuit of scientific understanding has always been the attempt to model or measure the unknown phenomenon. The benefits of developing models are that (1) they may be validated via compariosn with the reality they represent, and (2) they often lead to new insights and understanding into the nature of the thing studied, its causes, effects, etc. The scope of this research has been consciously limited in order to place the emphasis on (a) program complexity, as one of the significant components of

software complexity, and on (b) modeling or measuring, as a first step toward complete understanding.

Quantitative measurement of program complexity is the primary goal of this research. It has been refined into two specific subgoals: (1) objective validation of quantitative measures and (2) useful application of quantitative measures. This means that the major effort will be devoted toward (1) demonstrating that complexity as manifested objectively is indeed correlated with complexity as measured directly from the program source code, and determining how dependable, accurate, and generalizable that correlation is, and (2) demonstrating ways in which it can be beneficially applied in its own right towards combating some of the problems of software development.

Quantitative measures of program complexity could be used to monitor the real-time progress of the software development process. A regular series of measurements taken throughout the life-cycle should help make software production more visible, more manageable, more predictable, etc. Direct measures on the software should be able to serve as advance (or at least timely) warning of the presence of program complexity, before it is dramatically announced later by its undesirable manifestations. Even a gross measure could be beneficial, since there currently is none. Quantitative measures of program complexity could detect software system components which are highly susceptible to the problems associated with excessive complexity. They could also act as indicators of error-locality or error-prone-ness. A set of valid quantitative measures of program complexity would be an extremely useful tool for evaluating the quality of a software system. Besides acting as acceptance criteria for contracted software, such a yardstick could help answer other research questions empirically, such as, what main effects do various software development

methodologies and techniques have on the quality of the product.

Several measures have been proposed to measure program complexity. The ones which are being studied for applicability for the Software Engineering Laboratory are given below.

Sullivan [SULL, 73] proposed several quantitative measures of program complexity as "the degree of mental effort required for comprehension" of the program. His basic conception of the problem is that a count of the number of "active concepts" that are required to understand the code at a given point in the program is "a reasonable measure of the local complexity" at that point. He assumes that "the true complexity of a program, i.e., the difficulty of understanding it, is obviously a function of the data graph as well as the control graph, and other factors that we assume are less important, such as language syntax."

Sullivan's process (control) complexity measure is motivated by the assumption that one significant set of concepts, clearly required in order to understand the program at a given point, is related entirely to the control flow graph alone, namely, the paths (their number and significance and interrelationship) by which control may have arrived at that point. It is assumed that "paths which have unconditionally joined at some previous point are no longer relevant to the complexity of the current point." He defines a certain hierarchical decomposition of the control graph to explain precisely "what it means for a part of a scheme [control flow graph] to be irrelevant to, or isolated from, some other part." His decomposition of a graph P into two subgraphs B and P-B is defined in terms of two "bottleneck" nodes such that all paths into B unconditionally join at or before the one node, and all paths out of B unconditionally join at or before the other node, hence isolating B from P-B.

These decompositions are carried out successively until the graph has been partitioned into subgraphs which further admit only trivial decompositions, known as "elementary subschemes."

The elementary subschemes in the decomposed graph provide the basis for defining his C2 control complexity measure, as follows: the complexity of a graph is the sum of complexities for each elementary subschemes, the complexity for an elementary subscheme is the complexity for its (unique) terminal node, and the complexity for a node of an elementary subscheme is one less than the number of paths from the start of that subscheme to the particular node, discounting any paths which repeat the same node more than x times. Thus he has defined a "parameterized" measure, C2(x), of the complexity of the control structure of a routine, where x is normally fixed at 2 or 3; this measure is related to the number of paths that need to be actively understood and remembered by the programmer.

For Sullivan's resource (data) complexity measure, the basic assumption is that each data item contributes to overall data complexity since there are a set of active concepts associated with each variable which must be comprehended in order to understand the role of that variable and how that role is implemented within the routine. This view of each particular data item's complexity is formalized by defining the derived control graph proper to that data item in terms of the original control graph as follows: begin with just the nodes which reference (use or set) the data item, plus the start and terminal nodes; then add arcs between these nodes in the derived graph if and only if there exists a path in the original graph which joins the two nodes without passing through any other node in the derived graph.

The PD2 measure of complexity for a particular data item, in the context of a particular routine, is then defined as the C2 measure of the

derived control graph proper to that data item, with the exception that any elementary subschemes having only use-references in nodes other than its start node is assigned the value of zero, so that a "cluster" of use-references that can be reached only from a common set-reference point do not add to the complexity. Sullivan notes that this PD2 measure is sensitive to localization of references, in that data item references spread throughout several elementary subschemes of the original control graph are bound to make the derived graph larger and more complicated and hence increase its C2 measure. Notice that this measure is not measuring data complexity alone, but actually is sensitive to both control and data structure, thus it is a more complete measure of program complexity as a whole.

One measure to be studied [REIT, 76] is similar to Sullivan's process (control) complexity measure, since its fundamental model makes several of the same assumptions, but employes (a) a different criteria for the hierarchical decomposition, (b) a different evaluation of elementary subschemes, and (c) a different formula to combine their individual evaluations. This measure grew out of attempts to identify and count nesting levels and to assess the degree of "structured-ness" of transfers of control in non-structured-programming languages (such as FORTRAN) where neither concept is explicitly defined. It has its theoretical roots in the notion of "prime program" structures as discussed by Linger and Mills [LING, 77], a concept strongly related to the theory of structured programming control structures. Informally, a prime program structure is a single-entry, single-exit control structure which does not contain other prime program substructures of more than one node.

The model is based on the premise that the use of excessive or deep nesting or large and convoluted control structures increases a program's

complexity. The model seeks to identify the underlying organization of the control flow graph according to its hierarchical decomposition into nested prime program schemes. In fact, it would be quite interesting, as a side issue, to have some empirical data on (a) what prime program schemes actually appear in production FORTRAN code and at what frequencies, (b) how many GOTO's are written other than to simulate IFTHEN's, IFTHEN-ELSE's, WHILEDO's, etc. and (c) the relative frequencies of DO-loops versus non-DO-loops (test and branch). The measure is defined as a recursive formula

$$m(H) = (\sum_{i} m(G(i)) * s(H)) + L$$

for evaluating the complexity of each composite prime scheme, $\underline{m}(H)$, where $m(G(i))$ is the complexity of each of its subschemes, $s(H)$ is the complexity of the way they are connected together, and L is the additional level of nesting induced. The primitive schemes, simple actions and decisions are assigned unit complexity values.

Like Sullivan's data complexity measure, a proposed data complexity measure is also built up from a companion measure for control structure only, is really a control-plus-data complexity measure. The same model as the control structure complexity measure above is extended by realizing that the components identified by the prime program decomposition correspond exactly to levels of functional abstraction, as in the concept of stepwise refinement. Thus each prime scheme corresponds to a certain function from one set of data items (the domain) to another (the range).

The idea is to attribute a complexity weighting to the size or significance of the function computed by a prime scheme, and to include this new factor into the recursive formula for computing the overall complexity of composite schemes. As a first cut, define the weighting be the size of the domain plus the size of the range, thus reflecting the minimum

number of items necessary to characterize the "-ary-ness" of the function, i.e., how many data items are being manipulated when the prime scheme is abstracted to a single statement. This new factor is included as an additive term, resulting in a formula such as

$$m(H) = f(H) + (\sum_{1} m(G(ii) * s(H)) + L$$

where $f(H)$ is this functionality weighting for the amount of data manipulated by the prime scheme H.

The inter-routine binding complexity measure is based on the premise that routines represent a certain lowest level of operational partitioning of the system, i.e., the system is often described in terms of these routines, each has usually been identified during the design phase with some particular subtask and are separately compiled. There exist various bindings or interactions between these routines, for example, invocation of one by another, parameters passed and returned, and global variables shared by several. This measure seeks to capture this overall interface complexity, by counting and weighting these sources of inter-routine bindings. Each global variable can be weighted by the number of routines that access it (or common blocks by the number of variables in them multiplied by the number of routines that declare that block). Each procedure of function call can be weighted by the number of parameters passed in and returned (including the function value). The assumption here is that the more interactions there are between routines and the more significant they are, the more "active concepts" at the inter-routine interface level there are to be comprehended, and hence the greater this component of program complexity.

These measures are not an exhaustive set, by any means (c.f., [HELL, 73], VANE, 70]). The measures presented above are some of the more interesting and most promising ones examined to date. The intent in this

research is to investigate as many program complexity measures as possible. It is expected that several more will be studied during the course of this research.

This research will depend on both the screening and controlled projects for data toward achieving the subgoal of objective validation, and on the controlled projects for data toward the subgoal of useful applications. The data collected from the reporting forms will be processed to provide the various objective measures of complexity; these measures are the ones with which the proposed direct measures of complexity will be correlated (e.g., reliability, productivity, maintainability, etc.).

The static analyzer, which woul compute the various proposed quantitative measures of program complexity, will be built on top of the FORTRAN Automatic Code Evaluation System (FACES) developed by Ramamoorthy [RAMA, 74]. The FACES system consists of programs (1) the FORTRAN front end analyzer, which parses FORTRAN syntax and stores an abstract representation of the program system's source code into a fairly easily manipulated, linked list data base, and (2) the diagnostic analysis and inquiry routines, which operate on this internal data base to provide the information as directed. For this study, the front end will be used almost intact, but additional analysis routines would have to be written to compute the proposed measures of program complexity.

Furthermore, a history file will be kept of successive versions of the modules in the software system; it will be updated either weekly or at every official submission of a routine to the library. This history file will provide the capability of not only measuring the complexity of the final product software system, but also tracking the complexities of modules as they undergo development, testing, and refinement. This history file will also provide the necessary basis for studying certain use-

ful applications of the quantitative measures of program complexity such as indicators of errors.

Statistical techniques are to be used in achieving both of the major subgoals of this research: (a) evidence for objective validation of the measures of program complexity, and (b) demonstration of useful applications of these measures. The strategy for objective validation involves collecting data from a reasonable size sample of monitored projects, namely the screening experiments, and then analyzing the existence and degree of correlation between the complexity as directly measured and the complexity as objectively manifested. As a first cut, a "scatter graph", plotting for each project its manifested complexity rating on one axis and its measured complexity rating on the other axis, gives a visual picture of the degree of association between the two complexities. Further analysis would consist of (a) calculating some nonparametric correlation coefficients, and (b) applying the corresponding statistical significance test which determines (at the stated confidence level) whether or not the data supports the hypothesis that such an association exists in the population from which the sample was drawn.

These same correlation techniques would be used to demonstrate some of the useful applications of the quantitative measures of program complexity. For example, these proposed measures could be beneficially applied during software development if it could be shown that they correlate well with error rates on a per routine basis. Here each module would contribute one sample point, and each project development would constitute another replication of the experiment, the correlation coefficient being calculated and tested each time. Other useful applications could be made and conclusions drawn by employing some simple nonparametric

tests within the experimental design of the controlled experiments. For example, it would be nice to have evidence whether or not certain methodologies improve the complexity (measured or manifested) of the systems developed by means of them. This could be done by using the Mann-Whitney U test or the Kruskal-Wallis H test (one way analysis of variance) to determine the significance of differences with and without those methodologies [SIEG, 56]. The controlled experiment as currently planned will be replicated several times to give large enough sample sizes to apply these tests effectively.

VI. Current Status

The Software Engineering Laboratory was organized in August, 1976, with the fall of 1976 spent mostly in the designs of the seven reporting forms. The development of these forms included feedback from programmers and managers from NASA/GSFC and the offsite contractor. These forms have been reproduced as Appendix 2.

Beginning in November, 1976, most new software tasks that were assigned by the System Development Section of NASA/GSFC were given the added responsibility of filling out the forms, and thus entered our set of screening experiments. At the present time, about a dozen projects are currently involved. These projects are mostly ground support routines to various spacecraft projects. Appendix 4 describes these further.

The PDP11 database project was begun in late 1976. As explained in Section IV, this project is undergoing testing and should be operational shortly. The conversion of the collected screening data onto magnetic tape is now underway.

In June of 1977, the first of the controlled experiments will begin. Two teams (0 and 1) will be assigned tasks to be designed and developed for delivery to the Systems Development Section. The format of these tasks will be such that they will satisfy the experimental design outlined in Section II.

i.e., A_0 XB_0 C_0

A_1 YB_1 C_1

where A_i, B_i, and C_i, represent tasks to be developed by team i and X and Y are training sessions. These tasks will be developed on the PDP-11/70 at NASA/GSFC and will require approximately six months time. One team will consist of in-house NASA/GSFC personnel while the other will consist of contractor personnel. The tasks will consist of five separate subtasks with two comprising

project 'A', one project 'B', and two comprising project 'C'.

The five subtasks are as follows:

1. Human Resources Allocation and Management System
2. FORTRAN STATIC ANALYZER
3. Namelist Processor
4. Financial Management System
5. IUE Control Monitor

(See Appendix 4 for further explanations of these tasks.)

Task A will consist of subtasks 1 and 2, Task C will consist of subtasks 4 and 5, and Task B will be a subtask 3 following training sessions X or Y. Team 1 will be given a training session (Y) consisting of several techniques: PDL, Structured Programming, Walk-throughs, use of Librarians, Code Reading, and will also be given a small project B to take into account the necessary learning curve before Project C is undertaken. Team 0 will also be given a training session and a B Project, but will not be taught the above techniques.

For this first controlled experiment, there is complete control of the development process. The A projects enable us to determine the background of the personnel and the C projects enable us to determine the effects of the training sessions. The small B task enables us to filter out much of the learning curve involved in learning new techniques. Due to cost considerations, the duplicate developments must necessarily be kept small; however, the projects are large enough to require team interaction among the programmers and therefore we believe that they are generalizable to larger projects. In addition, the techniques taught in the Y training session are those most applicable to team situations.

A second, longer range, controlled experiment was begun in March, 1977. In this case, several similar large scale projects are being carefully mon-

itored. These projects can be summarized by the following table:

Project	Starting Date	Due Date	Launch Date	Personnel	Techniques Used
1. AEM-A	3/77	2/78	4/78	6	NONE
2. SEA SAT	4/77	2/78	5/78	6	Structured code, Librarian, code reading
3. ISEE-C	8/77	4/78	7/78	4	Training session Y of experiment 1
4. SMM	3/78	3/79	6/79	6	Not yet defined

In this case we are performing C-like experiments of controlled task 1. Due to budgetary restrictions, it is not possible to duplicate the development of each however, the tasks are highly similar and should give us results similar to the one strictly monitored controlled task 1.

While we realize that we have less control over this experiment, this controlled experiment does have the advantage that it is realistic in terms of NASA/GSFC's current development process. By varying the methodology, we expect to observe differences in project progress. Appendix 4 outlines these tasks in greater detail.

By the summer of 1977 the PDP11 database will be operational and data from the screening experiments and some of the A subtasks of the controlled experiments should be available. This data will first be checked for validity and then will be processed as outlined in Section V.

The next stop will be to define controlled experiment 3, based upon the preliminary results of experiments 1 and 2. It is expected that controlled experiment 3 will begin in early 1978. In this case, the techniques taught in training sessions X and Y and used in C, may be changed to reflect the new techniques to be measured. It is expected that as this process continues over several iterations, quantitative data on various products and development processes will result.

APPENDIX I

SOFTWARE ENGINEERING LABORATORY PERSONNEL

(as of May 1, 1977)

University of Maryland, Department of Computer Science

Dr. Victor R. Basili
Co-principal Investigator and Associate Professor

Dr. Marvin V. Zelkowitz
Co-principal Investigator and Associate Professor

Howard J. Larsen, Undergraduate Programmer

Robert W. Reiter, Graduate Research Assistant (prime area: complexity experiments)

David L. Weiss (Information Systems staff, Naval Research Laboratory), (prime area: errors)

Charles L. Wolf, Undergraduate Programmer

NASA/Goddard Space Flight Center, Greenbelt, Maryland

Frank E. McGarry
Contract Technical Officer and Head, Systems Development Section (code 582.1)

Robert Nelson, Member of Systems Development Section

Keiji Tasaki, Member of Systems Development Section

Walter F. Truszkowski, Member of Spacecraft Control Programming Section (prime area: management)

APPENDIX 2

REPORTING FORMS

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., simulator).

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 _____

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 Compilations		
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.)			
HOS			
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 _____

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 ______

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 _____

INSTRUCTIONS FOR COMPLETING THE CHANGE REPORT FORM 580-2 (2/77)

At the current time, there is little known about the types and causes of errors in various kinds of software. There have been many opinions expressed concerning the errors made in software development, and many "cures" have been suggested based on those opinions. The study for which the change report form will furnish the data is an attempt to do a careful, detailed investigation of the kinds of software errors that occur, 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 reducing the numbers 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. A unique number per form that will be assigned by the study group.

PROJECT. 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 at that time.

NEED FOR CHANGE DETERMINED - DATE. The date on which it was first realized that a change would be needed.

REASON. A brief description of the reason a change was needed.

What Modules/Subroutines Were Examined When It Became Evident That a Change Was Needed. The modules/subroutines that had to be looked at to determine where the change was to be made and what it affected.

CHANGE MADE - DATE. The date on which the change was started.

DESCRIPTION. A brief description of the change. 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." A listing of the changed code should be attached so that the implementation of the change can be looked at.

What Modules/Subroutines Are Changed. The names of all modules/subroutines where changes have been made.

Time Required to Design Change. The length of time required to design, not implement, the change. If the change is an error correction, enter only the length of time to design the correction.

Was the Change Made to Correct an Error?

No - If the change is not being made to correct an error, check this space and complete Sections A and C.

Yes - If the change is being made to correct an error or bug, check this space and complete Sections B and C.

SECTION A

Cause of Change and Item Affected. Most changes to code or documentation are the result of a higher level change somewhere else, e.g., a requirements change often requires a design change, which requires documentation and code changes. The purpose of this section is to discover what other changes resulted in the change now being made. Check the box(es) under the "caused by change in" column corresponding to changes in other items that resulted in the current change. Under the "affects" column, check those items that will have to be changed as a result of this change. Give the names of and references to any appropriate documents. If you are uncertain as to the cause or effect, check "can't tell" and describe in Section C the reason for the uncertainty.

SECTION B

Number of Run Analysis Form. The identifying number of the run analysis form for the run during which the error was first noticed, if the error was discovered via a computer run. That run analysis form should contain the number of this change report form.

Activities Used in Detecting the Error and 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 error detection (test, debug, etc.). Note that error messages have been divided into 2 categories: those produced by the support system (e.g., compiler, operating system), and those designed into the code for the specific purposes of this project or program. Special debug code is code that is inserted for the express purpose of finding this particular error, and which will be turned off or removed after this error is corrected. A check should be put in the second column next to the method by which the error was first detected. The third column is similar to the first, except that it refers to the methods used in trying to find the cause of the error. The fourth column refers to the activities that were successful in finding the cause.

Relation to Previous Modification

Yes – If you can determine that the error was the result of some previous change to the software, whether in the same subroutine or module as the error or not, check this space and give the number of the change report form completed for the previous change.

No – If you can determine that this error is unrelated to previous changes, check this space.

Can't Tell – If you suspect that this error is related to some previous change, but can't be sure, check this space, and explain why you are not sure in Section C.

Time Required to Isolate the Cause of the Error. Check the space that most closely approximates the time required to isolate the cause of the error. Note that the time to design the correction has been previously requested.

Cause Not Found – If the cause of the error was never found, check this space.

Workaround Used?

Yes: If a workaround was used because the cause of the error was never found, check this space.

No: If the cause was never found and a workaround wasn't used, check this space and explain what was done to remedy the error.

Clerical Error. We define clerical errors to be inadvertent spelling errors, misplaced or omitted delimiters such as decimal points, commas, or parentheses, missing list elements such as parameters, etc., where such errors are not the result of misunderstood or incorrect specifications, documents, or programs.

Aspects of the System Misunderstood or Misinterpreted. Errors that are not clerical can be viewed as the result of an incorrect or misinterpreted (misunderstood or mistranslated) aspect of the system. The aspect of the system where the problem occurred could be a document describing requirements, functional specifications, etc., or it could be some part of the system design, such as the intended use (result of invoking) a segment of code, procedure, subroutine, or module, or the structure (position of and representation of components) of data, or the meaning of data values (e.g., 0 = False, 1 = True), or some other aspect of the design. In addition to the above design considerations, it is of particular interest whether or not an interface is involved.

Other problems could involve understanding the syntax of the programming language, the semantics (ways of using and meaning of features) of the programming language, the hardware environment (CPU, Memory, I/O, Peripherals, etc.), or the software environment (operating system, compliers, text editors, utility package, etc.). If you are uncertain about some of the items, check "can't tell," and give as much information as possible in Section C.

When Did the Error Enter the System? Check the item that indicates where the error first appeared, i.e., in requirements definition, in the functional specs, in the design stage, in the coding and test stage, or at some other point. If you are uncertain as to when the error first appeared, check the item that indicates the earliest stage at which you are certain the error appeared and check "can't tell." In Section C give any information that may be helpful in resolving when the error first appeared.

SECTION C

Additional Information. This section is intended to permit explanation of any items you feel may be significant in categorizing the error, understanding its cause, how it was found, and any effects it may have that are not fully covered in previous sections. Do not hesitate to give a full description of the error 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.

CHANGE REPORT

NUMBER _____

PROJECT _____ CURRENT DATE _____

NEED FOR CHANGE DETERMINED DATE _____

REASON _____

What modules/subroutines were examined when it became evident that a change was needed? _____

CHANGE MADE DATE _____

DESCRIPTION (Please attach listing) _____

What subroutines/modules are changed (include version and line numbers) _____

The time required to design the change was ____ one hour or less, ____ one hour to one day, ____ more than one day.

Was the change made to correct an error: No - answer questions in Sections A, C
Yes - answer questions in Sections B, C

SECTION A

What is the change caused by and what does it affect?

	Can't Tell	Caused By Change In	Affects	Name(s)/References
Requirements/Specifications				
Design				
Hardware Environment				
Software Environment				
Optimization				
Other (Specify):				
Other (Specify):				

SECTION B

Number of run analysis form for run where error first noticed _____

What were the activities used in detecting the error and its cause:

	Activities Used for Detection	Error First Detected By	Activities Tried to Isolate Cause	Activities Successful in Isolating Cause
Test Runs				
Code Reading by Programmer				
Code Reading by Other Person				
Reading Documentation (documents:)				
Proof Technique: (method:)				
Trace: (type:)				
Dump				
Cross-Reference				
Attribute List				
Special Debug Code				
Error Messages General				
Error Messages Project Specific				
Inspection of Output				
Other (Specify):				
Other (Specify):				
Other (Specify):				

Was this error related to a previous modification? ____ Yes (Change Request #:)
____ No ____ Can't Tell

The time used to isolate the cause was ____ one hour or less, ____ one hour to one day, ____ more than one day.
Cause not found _____

Was a workaround used? ____ Yes
____ No (explain:)

Was it a clerical error? ____ Yes ____ No.

If not a clerical error, which aspects of the system were incorrect or misinterpreted?

	Can't Tell	Incorrect	Misinterpreted	Name(s) References
Requirements				
Functional Specifications				
Other Documents (Specify:)				
Design — Intended Use of Segment/Proc/Module				
Design — Value or Structure of Data				
Design — Other (Specify:)				
Interface				
Programming Language — Syntax				
Programming Language — Semantics				
Hardware Environment				
Software Environment				
Other (Specify):				

When did the error enter the system? ____ Requirements ____ Functional Specs ____ Design
____ Coding & Test ____ Other ____ Can't Tell

SECTION C

Please give any information that may be helpful in categorizing the change, understanding its cause, how it was found, and its ramifications.

Person Filling Out This Form: _____

APPROVED: _____

Date: _____

INSTRUCTIONS FOR COMPLETING THE RESOURCE SUMMARY FORM 580-3 (2/77)

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.

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)											
COMPUTER USAGE (HOURS CHARGED/NUMBER RUNS)											
OTHER CHARGES TO PROJECT											

580-3 (2/77)

INSTRUCTIONS FOR COMPLETING THE COMPONENT STATUS REPORT FORM 580-4 (2/77)

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 project.

PROGRAMMER. Name of programmer.

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

DESIGN

Create. Writing of a component design.

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

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

DEVELOPMENT

Code. Writing executable instructions and desk checking program.

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

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

TESTING

Mod. Module testing. Test run with test data on single module.

Integ. Integration testing of several components.

Review. Management review of testing status.

OTHER. Any other aspect related to the project not already covered. List type of activity.

COMPONENT STATUS REPORT

PROJECT ______________________________ **DATE** ____________

PROGRAMMER ______________________________

COMPONENT	DESIGN			DEVELOPMENT			TESTING			OTHER
	Create	Read	Review	Code	Read	Review	Mod	Integ	Review	

580-4 (2/77)

INSTRUCTIONS FOR COMPLETING THE COMPONENT SUMMARY - FORM 580-5 (2/77)

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 (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 seem to apply.

A. SPECIFICATIONS

Give how component is specified and any document name that defines form of specification. Also give the level of detail used in specifying component. See instructions in General Project Summary for further information on levels.

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).

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 both in terms of source lines and resulting machine language (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).

E. RESOURCES TO IMPLEMENT

For each of the three listed phases (Design, Code, Test), estimate computer runs, time needed, man-hours to implement, and estimated completion date. Estimate man-hours as Technical/Clerical. Technical includes design, programming and testing while clerical includes keypunching, typing documentation submitting runs, etc. This is independent of whether the work is performed by a programmer, librarian or other support personnel.

F. ORIGIN. Check all that apply.

New Function. Specification of an operation not previously present in system.

Further Elaboration. New component due to more detailed specification of existing specifications (e.g., top-down design of a new module).

Reorganization. Restructuring of existing components. Check reason why.

Extraction of Common Function – A function previously duplicated in several components is centralized in one.

Optimization – A new component is defined in order to optimize some system resource. Specify which resource (e.g., size of program, execution time, etc.)

Components Affected. List all components which must be modified because of this new one. If this is a further elaboration of a previous component, be sure to include the old component's name.

G. PERSON RESPONSIBLE. Include name of person responsible for component.

H. PERSON FILLING OUT FORM. Give name of person filling out form. This should be the same name as in G.

COMPONENT SUMMARY

PROJECT _____ DATE _____

NAME OF COMPONENT _____ CREATION DATE _____

BRIEF DESCRIPTION _____

TYPE OF SOFTWARE (Check All That Apply)

___ String Processing ___ Business/Financial ___ Table Handler

___ Scientific ___ Systems Program ___ Mathematical/Numerical

___ Real-Time System ___ Data Base Application ___ Telemetry

___ Command and Control ___ On-Board Computation ___ Attitude Orbit Determin.

___ Other: _____

A. SPECIFICATIONS

Form of Specification Functional ___ (level ___) Procedural ___ (level ___)

English ___ (level ___) Formal ___ (level ___) Other _____

Specification Document Reference Number _____

Precision of Specification Very Precise ___ Precise ___ Imprecise ___

B. INTERFACES

Number Components Called _____ Names _____

Number Calling This Component _____ Names _____

Number Shared Items _____ Names _____

C. PROGRAMMING LANGUAGES

Languages Used and Percentages _____ (___) _____ (___)

Constraints: Space _____ Execution Time _____

Other _____

Size: Source Instructions _____ Machine Words _____

Useful Items From Similar Projects

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

D. COMPLEXITY

Complexity of Function Easy ___ Moderate ___ Hard ___

___% Assignment Statements ___% Control Statements

E. RESOURCES TO IMPLEMENT

	Runs	Computer Time (min)	Man-Hours	Est. Completion Date
Design				
Code				
Test				

F. ORIGIN (Check All That Apply)

___ New Function ___ Further Elaboration of Existing Component

___ Reorganization Due to:

___ Change in Specifications or Requirements

___ Extraction of Common Function

___ Optimization (How? _____)

Components Affected: _____

G. PERSON RESPONSIBLE FOR COMPONENT _____

H. PERSON FILLING OUT FORM _____

580-5 (2/77)

INSTRUCTIONS FOR COMPLETING THE PROGRAMMER/ANALYST SURVEY - FORM 580-6 (2/77)

The purpose of this form is to classify the background of the personnel on each project. It should be filled out once at the start of the project by all personnel.

PROJECT. Which project are you currently assigned to.

DATE. Today's date.

NAME. Your name.

AGE. Current age.

NEW/REVISED. Check whether this is a new form or an update of a Programmer/Analyst form filled out earlier.

PRESENT JOB TITLE. Give current title.

EMPLOYER. Give current employer.

A. **EDUCATION**

Degrees. Fill out educational background.

Courses. Fill in number of university and in-house computer science courses.

B. **WORK EXPERIENCE.** Give years involved with computers and percent time in each listed activity

C. **SPECIFIC EXPERIENCE**

Structured Programming. Writing programs using only a limited set of control structures (e.g., if-then-else, do-while).

PDL. A Process design language. An algorithmic specification of a program.

HIPO. Hierarchical Input Process Output. A graphical technique describing a program as a function of its input and output data.

Top Down Development. A technique where high level modules are developed before the modules that are called by these high level routines.

Stubs. A top down technique where each undeveloped function is simulated by a short testing routine.

Stepwise Refinement. A top down technique where each line of code is replaced by an expansion of its definition in greater detail.

Chief Programmer. A technique where an individual programmer writes top level code and major interfaces and delegates responsibility to others to complete it. A librarian manages all source code and documentation.

Operating System. Give system name and amount of experience.

Types of Programming Experiences. Give level of experience for each type of programming.

PREFERRED METHOD OF ACCESS

Batch. Remote submission of runs by card decks.

Interactive. Using a terminal to create and immediately test programs.

Hands On. Physically running the computer, as with a minicomputer.

TSO. IBM Time Sharing Option. Using terminals to create programs and submission for execution.

PREFERRED PROGRAMMING LANGUAGES. In which languages do you prefer to program (e.g., FORTRAN, ALC)?

PREVIOUS EXPERIENCE. List the names and positions you held on previous projects which were similar to this current one.

EVALUATION. Give any additional information that you think is relevant.

PROGRAMMER/ANALYST SURVEY

PROJECT _____ DATE _____

NAME _____ AGE _____ NEW____ REVISED____

PRESENT JOB TITLE _____ EMPLOYER _____

A. EDUCATION

Degrees (include date, location, and major)

1. _____
2. _____
3. _____

Approximately how many college credit computer science courses did you take? _____

Approximate grade point average in such courses (A = 4.0) _____

Approximately how many in-house or training seminars in computer science have you attended? _____

B. WORK EXPERIENCE

Total Number of years involved with computers _____

Estimate fraction of total experience spent:

_____ % in individual effort, _____ % in cooperation with others (teams)

_____% supervising others (describe _____)

If any team experience, describe _____

_____

C. SPECIFIC EXPERIENCE

For the following four sections, rate each item using the following code: 1 = never used, 2 = used occasionally, 3 = used often.

1. Techniques

   Structured Programming____ PDL____ HIPO____ Top Down Development____ Stubs____ Stepwise Refinement____

   Chief Programmer Team____ Other_____ ____ Other _____ ____

2. Programming Languages

   Assembler (360/370)____ Assembler (_____)____ FORTRAN____ COBOL____ PL/1____ SIMPL____ ALGOL____

   SNOBOL4____ PASCAL____ Other (_____)____ Other (_____)____

3. Operating Systems and Command Languages

   System (360/370 OS)____ System (_____)____ System (_____)____

4. Types of Programming Experiences

   String Processing____ Business Financial____ Scientific____ Table Drivers____ Systems Programming____

   Real-Time Systems____ Mathematical/Numerical____ Data Base Applications____ Telemetry____

   Command and Control____ On-Board Computation____ Attitude Orbit Determination____ Other (_____)____

   Other (_____)____

PREFERRED METHOD OF ACCESS: Batch____ Interactive____ TSO (Remote Batch)____ Hands On____

PREFERRED PROGRAMMING LANGUAGES _____

PREVIOUS EXPERIENCE IN RELATED PROJECTS _____

_____

EVALUATION _____

580-6 (2/77)

INSTRUCTIONS FOR COMPLETING THE COMPUTER PROGRAM RUN ANALYSIS - FORM 580-7 (2/77)

This form will be used to monitor the activities for which the computer is used in the course of the project life cycle. An entry should be made every time the computer is used by the person initiating the run. This form should be turned in weekly, or sooner if it is complete.

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

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

NUMBER. This is the unique number for this form. The run number will be this number with the run line number (00 to 15) added to it.

DATE. Date form turned in.

JOB ID. Name of run from job submittal card.

RUN DATE. Date run submitted in format month-day-year.

INTERACTIVE. Check if this is an interactive run, and write system used (e.g., TSO) as job id. Leave this blank if a batch run.

CHARGE TIME. Give the cost of the run (in minutes-seconds) in terms of cpu minutes.

PURPOSE OF RUN. What is the purpose of this run. (e.g., system test, find bug in module XYZ, clean compile of module ABC, back-up libraries onto tape, etc.)

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

CATEGORY OF WORK.

Write N if this is a new function added to a component,
M if it is a modification to the specifications of a component,
C if it is a correction of an error to a component, or
O if it is for some other reason.

If this is a test run, write:
M if this is a test of a single module,
P if is a partial integration test, or
F if it is a full integration test.

Check all of the following boxes, if they apply:
Compilation - some component is being compiled (or assembled)
Load - a set of modules is being loaded by the system loader (link editor)
Execution - a set of modules are being executed
Utility - a utility program is being used (e.g., listing of file, file copy)

RESULTS. Give results of run. If program terminated with an error, give error message and component that generated message or component that caused operating system to generate message.

COMPUTER PROGRAM RUN ANALYSIS

PROJECT _____ NUMBER _____

PROGRAMMER _____ DATE _____

	JOB ID	RUN DATE (M M D D Y Y)	X-INTERACTIVE	CHARGE TIME (M M S S)	PURPOSE OF RUN	COMPONENTS OF INTEREST	N-NEW FUNCTION M-MODIFICATION C-CORRECTION O-OTHER	M-MODULE TEST P-PARTIAL INTEG F-FULL INTEG	X-COMPILATION	X-LOAD	X-EXECUTION	X-UTILITY	RESULTS (e.g., Error Messages)
01													
02													
03													
04													
05													
06													
07													
08													

67

580-7 (2/77)

APPENDIX 3
DATABASE FORMAT

The conversion of the data on the forms into the database is controlled by a program running on a PDP-11. This program is table driven where the table describes the keypunched format of the paper forms. The table is used to specify how the verify and edit routines handle the incoming data. The following is a brief description of these table entries. A forthcoming technical report "The Software Engineering Laboratory: Database Description" will describe these in greater detail, and will give the translation of the reporting forms (appendix 2) into these entries.

As mentioned previously in Section IV, the raw data will be stored as a file under UNIX. Each file will contain a common set of keypunched forms, each stored as a set of records. The program will read in the first record and the first table entry for that particular form.

A given form will be described via a set of such table entries. The table entries will describe the contents of those records, what conversions to perform on the data, and where to place the output. The following discussion describes the table entries in greater detail.

Each record of data making up one form is coded with a letter or digit. This character is used to determine where in the set of tables to continue processing.

There are five different table entries - four of the entries describe internal control for the program (e. g. specifying how many times a given field may be repeated) while the fifth describes the format of the input

data. Each table entry is a fixed seven word entry in the PDP-11, but all seven words may not be used in all cases.

When the program is run, the table describing the form and the name of the output file is read. Thus one program suffices for all forms. The program reads the UNIX file containing the raw data, validates it and produces three outputs: (1) an INGRES file for correct data, (2) an error file containing records in error and (3) a listing of the data pointing out all errors. Options on the program exist to supress some of these outputs.

Each seven word entry has the following format:

Word 1 - TYPE (8 bits)
CHAR (8 bits)

Word 2 - INTA

Word 3 - INTB

Word 4 - INTC

Word 5 - INTD

Word 6 - INTE

Word 7 - INTF

The appropriate type of record (1 to 5) is determined by the TYPE field of Word 1. The 5 types are as follows:

Type 1 - Data Format Entries. These describe data from the forms.

CHAR - error level, with following values:

E - field on form must be exactly as stated in validity code or else form will be rejected.

1 - field may be blank. If nonblank, it must be correct.

2 - field is always acceptable whether blank, incorrect or correct.

For invalid fields, '****' is printed in output listing for E errors and '++++' is printed for 2 errors.

INTA - Number of characters in input record for field.

INTB - Format code - type of data in field (Component name, date, text, number, etc.)

INTC - Type conversion of field. (move as is, convert to integer, convert date to internal format, etc.)

INTD - Character position of converted field in output record

INTE, INTF - Used with specific format codes.

Type 2 - While loop. Controls arbitrary repitions of certain fields.

CHAR - Control character

All table entries until the EXIT (type 4) record are processed. If the next record of data is of type CHAR, then the loop is reprocessed. This is used to handle arbitrary numbers of certain data items, such as the number of milestones reported on the General Project Summary.

Type 3 - Repeat Loop. Controls a fixed repition of certain fields.

INTA - repeat count.

This is similar to the While loop except that INTA specifies the number of times to repeat. It is used where a fixed number of fields must appear, such as Language Experiences in the programmer analyst survey.

Type 4 - EXIT. End repeat and while loops. This specifies the end of repeated information.

CHAR - 'Y' - output information

'N' - no output of information

If N is specified no data is output at this time and information is output at the end of the form.

If Y is specified, data is output. For example, each line in the Resource Summary is controlled by a while loop. Even though one line may be incorrect. Other correct lines may be entered into the database.

Type 5 - Stop processing; end of form.

APPENDIX 4
NASA SOFTWARE

NASA Environment

The Software Engineering Laboratory has been created to function within the constraints and service of one segment of the NASA/GSFC computational facility. This facility, which is under the control of the Mission and Data Operations Directorate (Code 500), consists of two primary hardware systems:

1.) A series of S/360's
2.) PDP-11/70

S/360 Environment

The S/360 is the prime development and operations machine for all work in the Mission Support Computing and Analysis Division (Code 580). Although NASA/GSFC has nine IBM 360's, only two handle most of all the work related to the Laboratory. The first is a S/360 model 95 running under OS and complimented with Asynchronous Support Processor (ASP) and the standard IBM Time Sharing Option (TSO). The second is a S/360 model 75 running under OS and supporting TSO.

The workhorse for this user community is the model 95, but most of the software in question can make use of the 75 with essentially no modifications. Although the 95 has 5 million bytes of memory available, special requirements and daily operational support activities reduce the available memory to about 2 million bytes to the general user. This machine has nearly 1000 registered users contending for this available storage.

Three TSO regions support the numerous terminal activities for the 360. Each of these require 270K of main memory to service both editing type operations as well as foreground jobs executing under TSO. Graphics work

is also supported via IBM 2250's, 2260's and a series of Anagraph 6600 devices.

Both the 95 and 75 are primarily batch oriented. While the 75 is card deck oriented, the 95 receives the bulk of its work via remote submittal. The remote jobs are transmitted by the user by way of various remote entry terminals such as the IBM 2780, IBM 1050 or Anagraph 6600.

There are various devices available by which users may store software libraries. Disk and magnetic tape are available to store source code, load modules, and data in general. No drum space is available to the general user on the S/360 and the available on-line disk space is very limited.

The primary language used by the software community is FORTRAN with some small usage of Assembler language and other languages. No particular language standards are rigorously imposed at the center.

During development of software systems users can expect turn around time to vary from one or two hours for small, half minute jobs, to one day for medium jobs (3 to 5 minutes, less than 600K), to several days for longer running and larger size jobs. In order to support the testing of interactive graphics work, users are normally required to schedule time at the S/360 model 75. The time must be scheduled one week in advance, and normally is only available at night and on weekends. This is to minimize the impact on daily operational support required on that machine. For most of the tasks involved in the Laboratory which require graphics development, the programming team would normally schedule an average of 2 three hour sessions per week for a period of three months for a 1 year development effort.

PDP-11/70 Environment

This machine is directly under the control of the Systems Development and Analysis Branch (Code 582) and is a more accessible machine than the S/360. Naturally, there are greater restrictions on type of software development supported due to the limited memory and peripheral equipment. The 11/70 contains 128K 16 bit words with 2 nine track tape drives, no card reader, 2 RK05 disk drives, one RP04 (88 megabyte) disk, 1 Versatec electrostatic printer and 4 V50 DEC CRTs.

There is no machine operator, which necessitates each programmer becoming familiar enough with the machine to operate it. Each programmer or team of programmers must schedule time on the machine for testing and validating software. This time is normally made available in 2 or 3 hour blocks during the day.

Again, the primary language is FORTRAN with some minor application of assembler language.

Although the 11/70 has 128K words of memory, the word size limits the addressable space to 32K forcing programmers to break larger programs into executable 'tasks' occupying less than 32K of memory. Ample direct access storage is provided to requesting tasks to store source libraries, load modules and related data.

Due to the unavailability of a Card Reader, all source code must be entered via CRT or by magnetic tape generated on some other machine.

Software Development

The software developed by the Systems Development Section of NASA is usually ground support software used to control spacecraft operations. This usually consists of attitude orbit determinations, telemetry decom-

mutation and other control functions. The software that is produced generally takes from six months to two years to produce, is written by three to six programmers most of whom are working on several such projects simultaneously, and consists of six man-months to ten man-years of effort. Projects are supervised by NASA/GSFC employees (of Section 582.1) and the personnel are either NASA personnel or outside contractors. Resource estimates are in new code for the project since most projects use some routines written for earlier projects.

Screening Experiments

The following pages list the set of screening experiments that have been started to date. These projects are not impacted in any way, but are required to fill out the various reporting forms.

Screening Task #1

Title: PAS Attitude Determination System

Objective: The Panoramic Attitude Sensor (PAS) Attitude system is a software system created to support both the ISEE-A and the IUE satellites. The software's main purpose is to read telemetered information from the spacecraft, then from pertinent sensor date - determine the current attitude (right ascension and declination) to some specified accuracy.

The software is to run interactively to support real time requirements as well as to support voluminous editing of telemetry information during the actual attitude determination process. The software must support the mission initially at launch then continue through the lifetime of the satellite (from 1 to 3 years). During the reorientation process (turning the vehicle into the final desired attitude), which takes place several hours after launch, the software must perform in a real-time mode displaying the data during the maneuver so that analysts may determine if the maneuver is proceeding as scheduled or whether there may be some difficulty (e.g., the spin axis is being readjusted in the wrong direction).

The computed attitudes during the lifetime of the mission are transmitted to experimenters around the world along with pertinent experimental information from the satellite. Experimenters must know not only where the satellite was, but which way it was pointed when certain scientific measurements were made.

The system is somewhat I/O bound with the voluminous amounts of Telemetry data being read and massaged.

Environment: Target Computer - S/360
Development Computer - S/360
Language - FORTRAN
Level of Effort - 9 Man Years
Program Size - 450 Modules
80000 Lines of Code
(30% reusable from other projects)

Methodology: Some walk-throughs
Some Code reading
Librarian Used
Top Down (Somewhat)
Strict Design requirements

PERTINENT INFORMATION: Information was made available by way of the following forms:

1. General Project Summary
2. Component Summary (Approximately 400*)
3. Component Status
4. Change Reports (Approximately 150)
5. Resource Summary
6. Run Analysis (Approximately 500)

*Before and after code completion

There were the standard Laboratory forms which were generally filled out faithfully. There are some shortcoming in the change report forms due to early misinterpretations and unavailability of forms early in the project.

Screening Task #2

Title: ISEE-B Attitude Determination System

Objective: The International Sun-Earth Explorer-B (ISEE-B) Attitude Determination System processes Earth and Sun Attitude data received from the spacecraft and computes definitive attitude and spin rate for the three-year lifetime of the mission.

Environment:
- Target Computer - S/360
- Development - S/360
- Language - FORTRAN
- Level of Effort - 6 1/4 years
- Program Size - 350 modules
  12,000 lines of code

Methodology:
- Functional Specifications
- Top down design, development and testing by subsystem
- Some iterative enhancement
- Code reading
- Librarian used

PERTINENT INFORMATION: Information is being collected through the use of the following forms:

1. General Project Summary
2. Component Summary
3. Component Status
4. Change Reports
5. Resource Summary

Screening Task #3

Title: Goddard Mission Analysis System (GMAS)

Objective: GMAS is designed to perform general mission analysis support for both pre-flight and in-flight operations for near-earth as well as interplanetary missions. The system consists of four major entities which are:

1. Executive
2. Library of utility load modules
3. Dynamic arrays
4. Automatic sequence

The parsing of the system in the manner supports 2 major requirements of the system:

1. Dynamic configuration and reconfiguration of the executing software to suit a stated problem by way of user input.
2. Support of the ability to add or change computational load modules into the system without perturbing any other part of the GMAS system.

Such problems as launch window studies, lifetime predictions, shadowing computations, monte carlo studies and various trajectory targeting problems are typical for this system.

It is a highly computational driven system with no extensive I/O problems. It has interactive graphics requirements which are supported by way of the Graphics Executive Software System (GESS).

Environment: Target Computer - S/360
Development Computer - S/360
Language - FORTRAN
Level of Effort - 10 Man Years
Program Size - 400 Modules
45000 Lines of Code (20% reusable from other systems)

Methodology: Walk Throughs
Code Reading
Librarian

PERTINENT INFORMATION: Data was made available for this project only through a modified version of the change report form. Since this project was well under way before the Software Engineering Lab was created, no additional requirements were placed on this task.

The modified version of the change report form was used from the beginning of the project and has information recorded which will only be applicable to the investigation into software errors - which is one of the primary areas of concern in the Systems Engineering project.

Screening Task #4

Title: Averaging Orbit Propagator

Objective: This program is designed to service the GMAS program (Task #3). It is a completely separate, independent load module which is connected to GMAS by two small assembly language utility routines.

The propagator is a highly computational piece of software designed to advance a vehicle state (position and velocity) over some period of time in an extremely rapid fashion. As input, the program would receive some initial state and be asked to advance this set of elements to some later (or earlier) point of time, then provide this updated state and time to the requesting module.

Environment: Target Computer - S/360
Development Computer - S/360
Language - FORTRAN
Level of Effort - 6 Man Months
Program Size - 60 modules
7000 Lines of Code (20% reusable)

Methodology: Walk Throughs
Code Reading
Libraries

PERTINENT INFORMATION: Data was made available by way of the following forms:

1. General Project Summary
2. Component Summary
3. Component Status
4. Change Reports
5. Resource Summary
6. Run Analysis

Screening Task #5

Title: General Parameter Program (GPARM)

Objective: This program is designed to service the GMAS program (Task #3). Like task #4, it is also a completely independent load module which could certainly be used as a separate program from GMAS.

The primary purpose of the program is to convert some standard set of input elements (position and velocity) to a variety of additional parameters in various coordinate systems and then to transmit the information to an on-line printer. The program is a computational one with a limited amount of I/O.

Environment: Target Computer - S/360
Development Computer - S/360
Language - FORTRAN
Level of Effort - 5 Man Months
Program Size - 20 modules
2500 Lines of Code

Methodology: Walk Throughs
Code Reading
Librarian

PERTINENT INFORMATION: Data was made available by way of the following forms:

1. General Project Summary
2. Component Summary
3. Component Status
4. Change Reports
5. Resource Summary
6. Run Analysis