(NASA-TM-89372) MEASURING SOFTWARE DESIGN N87-23189
(NASA) 77 p Avail: NTIS HC A05/MF A01;
single copies available from NASA/GSFC, Code
552, Greenbelt, Md. 20771 CSCL 09B Unclas
G3/61 0076836

SOFTWARE ENGINEERING LABORATORY SERIES SEL-86-005

MEASURING SOFTWARE DESIGN

NOVEMBER 1986

NASA

National Aeronautics and Space Administration

Goddard Space Flight Center
Greenbelt, Maryland 20771

FOREWORD

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

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

The goals of the SEL are (1) to understand the software development process in the GSFC environment; (2) to measure the effect of various methodologies, tools, and models on this process; and (3) to identify and then to apply successful development practices. The activities, findings, and recommendations of the SEL are recorded in the Software Engineering Laboratory Series, a continuing series of reports that includes this document. A version of this document was also issued as Computer Sciences Corporation document CSC/TM-86/6053.

Single copies of this document can be obtained by writing to

Frank E. McGarry
Code 552
NASA/GSFC
Greenbelt, Maryland 20771

AUTHORS

The principal author of this document is

Dave Card

Additional materials were contributed by

Bill Agresti
Vic Basili
Vic Church
Bill Decker
Leon Jordan
Frank McGarry
Jerry Page
Ed Seidewitz
Rick Selby
Mike Stark

Original sources are fully identified in the references.

ABSTRACT

An extensive series of studies of software design measures conducted by the Software Engineering Laboratory is described. Included are the objectives and results of the studies, the method used to perform the studies, and the problems encountered. The document should be useful to researchers planning similar studies as well as to managers and designers concerned with applying quantitative design measures.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

LIST OF TABLES (Cont'd)

SECTION 1 - INTRODUCTION

This document describes an extensive series of studies of software design measures. It presents the study objectives and explains how the studies were conducted, what results were achieved, and where problems were encountered. The document should be useful to researchers planning similar studies as well as to managers and designers concerned with applying quantitative design measures.

These studies were conducted by the Software Engineering Laboratory (SEL) (Reference 1) as part of a continuing program of software measurement (Reference 2) and technology evaluation (Reference 3). The overall goal of the measurement part of this program is to provide effective measures for every phase of the software life cycle. Code measures have already been extensively analyzed (Reference 4), and a study of specification measures is currently underway (Reference 5).

This document is organized into five major sections. Section 1 describes the background and objectives of the design studies. Section 2 outlines the measurement approach and problems encountered. Section 3 presents the measurement results. Section 4 discusses some studies of new design methods for which design measures are being developed. Section 5 summarizes the SEL design measurement experience.

1.1 THE SOFTWARE ENGINEERING LABORATORY

The SEL (Reference 1) is a research project sponsored by the National Aeronautics and Space Administration/Goddard Space Flight Center (NASA/GSFC) and supported by Computer Sciences Corporation (CSC)/System Sciences Division and the University of Maryland/Computer Sciences Department (Figure 1-1). The objectives of the SEL are to measure the process of software development in the GSFC flight dynamics environment, identify technology improvements, and transfer this technology to flight dynamics software practitioners.

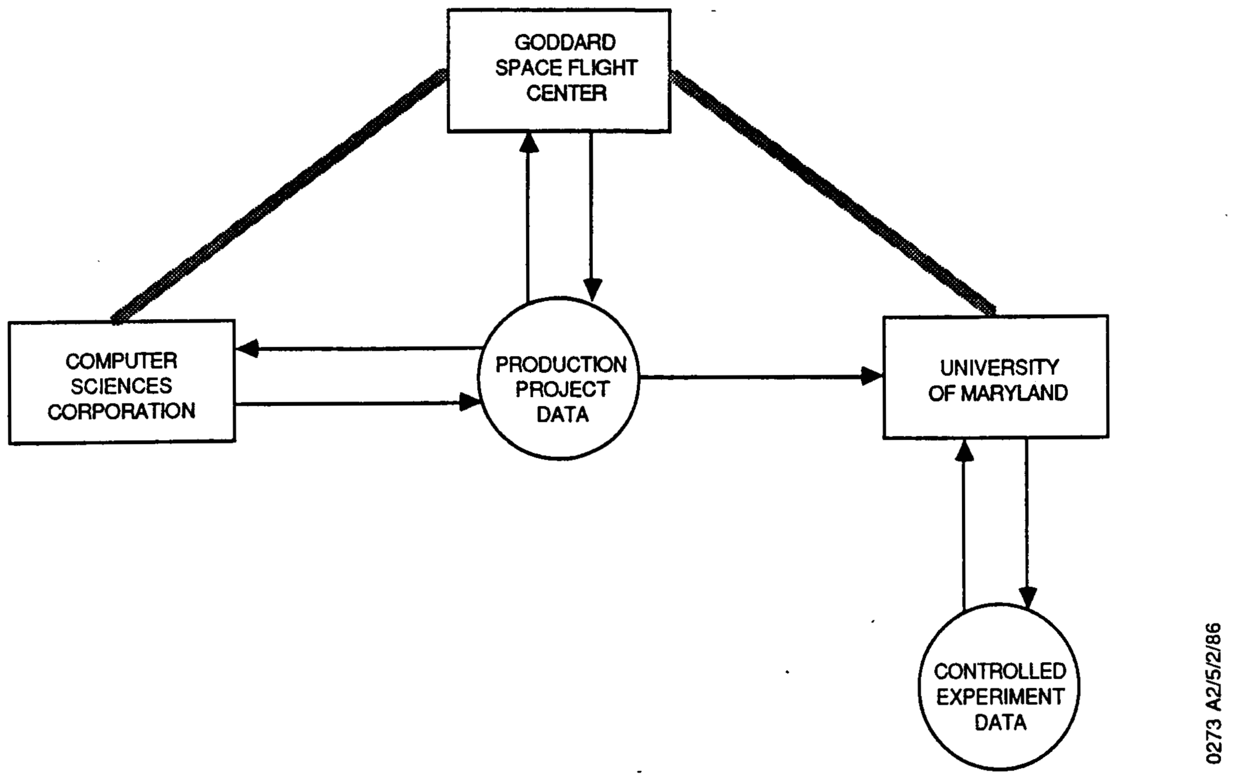

GODDARD SPACE FLIGHT CENTER
COMPUTER SCIENCES CORPORATION
PRODUCTION PROJECT DATA
UNIVERSITY OF MARYLAND
CONTROLLED EXPERIMENT DATA
0273 A2/5/2/86

Figure 1-1. Software Engineering Laboratory Organization

The SEL monitors software developed for spacecraft flight dynamics applications at GSFC. During the past 10 years, the SEL has collected data from more that 50 software development projects. Most flight dynamics projects are developed on a group of IBM mainframe computers using FORTRAN and assembler programming languages. Specific software applications include attitude determination, attitude control, maneuver planning, orbit adjustment, and general mission analysis. The attitude systems, in particular, form a large and homogeneous group of software that has been studied extensively.

Data are collected both manually and automatically during and after development (Reference 6). For example, errors detected and hours charged are reported on forms by the development team. Computer-use data are collected automatically as part of the accounting procedure. In addition, the completed software is processed through a source analyzer to obtain detailed size and structure measures. All of this information is stored in a computer data base accessible to SEL participants.

1.2 STUDY OBJECTIVES

The software product is first defined in the design phase of the software life cycle. The basic objective of these studies was to develop design measures that effectively predicted eventual software quality. The target quality measures were productivity, reliability, and maintainability. Figure 1-2 shows the life-cycle phases in which these qualities are measured in the flight dynamics environment. Unit cost (Figure 1-2) is related to productivity. Measures corresponding to these qualities are defined as follows:

- Productivity (for project)
  - Lines of code per staff hour
- Cost (by unit or module)
  - Staff hours per executable statement
- Reliability
  - Errors per line of code (for project)
  - Faults per line of code (for unit/module)
- Maintainability (for project)
  - Modules or units affected per change
  - Staff hours to implement change

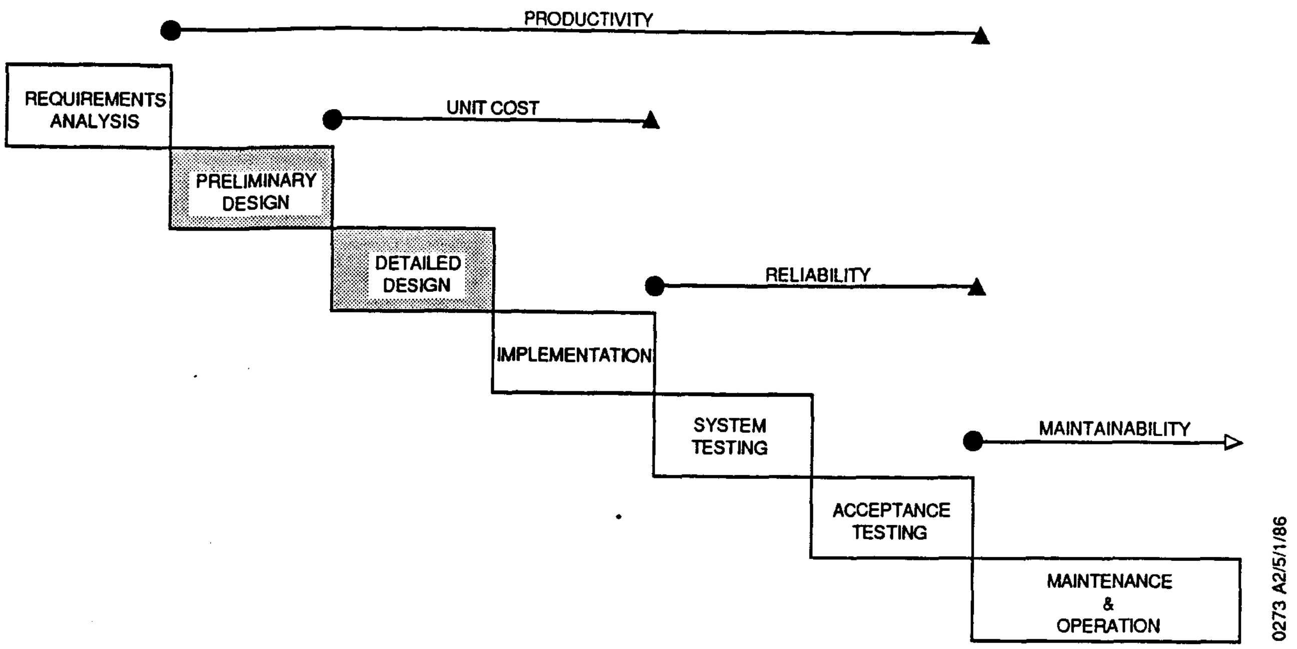


Figure 1-2. Quality Measurement Across the Software Life Cycle

Because these studies were undertaken in the context of a specific production environment, it was necessary to work with the existing design practices and representation techniques (Reference 7). These include the development of tree charts and pseudocode. The common objective of the studies can therefore be stated as follows: given the flight dynamics design representation, practices, and problem, develop design measures for predicting software quality.

SECTION 2 - MEASUREMENT APPROACH

The approach to data collection was basically to gather as much and as varied design information as possible and then to reduce it to its essential elements. This section describes this approach, discusses the problems encountered, and summarizes the data collected for analysis.

2.1 DATA COLLECTION

To support the wide range of studies undertaken, it was necessary to collect detailed structure, cost, and error data at both the project and module levels. A project is a set of functional subsystems implemented as one or more programs that support a common mission. A module is the smallest unit of independently compilable code. In FORTRAN, a module corresponds to a subroutine.

Cost and error data were collected via the standard SEL forms (Reference 6). Design data were extracted from original design materials and from the implemented code. This necessitated the establishment of a design materials library and the development of procedures for analyzing software source code (Reference 8).

The design materials library includes formal documents, notebooks, and review materials. Source analysis is performed by a specially modified version of the FORTRAN Static Source Code Analyzer Program (Reference 9).

The design data extracted from the code by the analyzer program include calling trees and counts of statements, variables, decisions, etc. Table 2-1 lists the targeted <u>module-level</u> design measures, most of which are provided by the source analyzer program. Other design-related data provided by SEL data collection mechanisms (Reference 6) were also studied.

Table 2-1. Design Analyzer Output

ITEM	NAME	LOC	FORM	SOURCE[a]	DESCRIPTION
1	PROJ CODE	01-02	I2	CIF	PROJECT CODE FROM ENCODE.HDR
2	COMP NAME	03-10	A8	CIF	NAME OF COMPONENT
3	COMP CODE	11-13	A3	CIF	COMPONENT CODE (NUMERIC HEX)
4	PAN LEVEL	14-15	I2	CIF	PANVALET LEVEL NUMBER
5	MOD FUNCTION	16-17	I2	CIF	MODULE FUNCTION 1:
6	SUBSYST FUNC	18-19	I2	CIF	SUBSYSTEM FUNCTION 1:
7	ORIGIN	20	A1	CIF	ORIGIN OF COMPONENT 1: NEW 2: EXTENSIVELY MODIFIED 3: SLIGHTLY MODIFIED 4: OLD (UNCHANGED)
8	NEXECSTMT	21-24	I4	CIF	EXECUTABLE STATEMENTS (COUNT)
9	NLINESCMTS	25-28	I4	CIF	SOURCE LINES, INC. COMMENTS
10	NCMTLINES	29-32	I4	CIF	NUMBER OF COMMENT LINES
11	TOTCHANGES	33-34	I2	CRF	CRFs FOR THIS COMPONENT
12	TOTERRORS	35-36	I2	CRF	CRFs–ERROR–FOR THIS COMPONENT
13	DSNHRS	37-44	F8.1	CSR	TOTAL HOURS, DESIGN PHASE
14	CODEHRS	45-52	F8.1	CSR	TOTAL HOURS, CODE PHASE
15	TESTHOURS	53-60	F8.1	CSR	TOTAL HOURS, TEST PHASE
16	CALLINGCOMPS	61-62	I2	DSF	COMPONENTS CALLING THIS ONE (FANIN)
17	CALLEDCOMPS	63-64	I2	DCF	COMPONENTS CALLED BY THIS (FANOUT)
18	CALLEDNSCOMPS	65-66	I2	DSF	NONSYSTEM COMPS CALLED (FANOUT EXCLUDING SYSTEM ROUTINES)
19	DEPTHNSYS	67-68	I2	DSF	DEPTH OF NONSYSTEM CALLS (TO TERMINAL NODE–TREE DEPTH)
20	AVGLEVEL	69-72	F4.1	DSF	AVERAGE LEVEL OF THIS COMPONENT (ROOT NODE = 1)
21	NARGSPARAMS	73-75	I3	DCF	ARGUMENTS IN PARAMETER LIST
22	NARGSCALLS	76-78	I3	DCF	ARGUMENTS USED IN CALL STATEMENTS
23	TOTVARSKNOWN	79-81	I3	DCF	TOTAL VARIABLES KNOWN TO COMPONENT
24	NVARSEXECSTMT	82-84	I3	DCF	VARIABLES USED IN EXECUTABLE STATEMENTS
25	NCOMMONVARS	85-87	I3	DCF	TOTAL VARIABLES IN COMMON
26	NUNREFCOMVARS	88-90	I3	DCF	UNREFERENCED VARIABLES IN COMMON
27	CLUSTER	91-92	I2		CLUSTER NUMBER (ANALYTIC)

0273 S7/3/3

NOTE: THE DDF IS A SEQUENTIAL FILE OF LENGTH 92; NO KEYS ARE DEFINED.

[a] CIF – COMPONENT INFORMATION FORM
CRF – CHANGE REPORT FORM
CSR – COMPONENT SUMMARY REPORT
DSF – DESIGN STRUCTURE FILE
DCF – DESIGN COMPONENT FILE

Because a complete set of design materials could not be recovered for all of the projects studied, most of the analysis reported in this document relies on design information extracted from the software source. Most design decision results are directly reflected in the code, although some design information, especially process data, cannot be captured in this manner. These experiences suggest that design-unique data can be captured only at design time.

Table 2-2 lists a characteristic set of project-level design parameters for future data collection efforts (Reference 10). The module-level measures proved to be too voluminous for easy historical reference. Individual module records must be summarized to project records.

2.2 DATA SUMMARY

As indicated previously, the approach to data collection was to gather as much design information as possible and then reduce it to a more manageable subset based on analysis results. The amount and nature of the data obtained varied from project to project, with project summary information obtained for 29 projects. Detailed design data were extracted from only the eight projects shown in Table 2-3; only four projects were adequately covered by component summary reports, which report module function. Consequently, the full range of projects could not be employed for all studies.

Preliminary investigations of the data indicated that values for some design and quality measures did not conform to the normal distribution model upon which common regression and analysis-of-variance techniques are based. In particular, values of cost (Figure 2-1) and fault rate (Figure 2-2) were skewed to the left (toward zero). Many module-level studies therefore employed nonparametric and classification techniques instead of least squares procedures. The analysis technique used for each study is identified in Section 3.

Table 2-2. Project-Level Design Parameters

PARAMETER CLASS	PARAMETER NAME	DESCRIPTION
IDENTIFICATION	PROJECT	TWO-DIGIT PROJECT CODE FROM THE ENCODING DICTIONARY
	PHASE	LIFE-CYCLE PHASE IN WHICH THESE PARAMETERS WERE MEASURED (DETAILED DESIGN, IMPLEMENTATION, ETC.)
SYSTEM CONTEXT	ENTITY	NUMBER OF ENTITIES (PHYSICAL OBJECTS) AFFECTING OR AFFECTED BY THE SOFTWARE SYSTEM
	DATAIN	NUMBER OF DESCRETE DATA ITEMS (OR VARIABLES) OUTPUT FROM THE SYSTEM (EACH ITEM MAY OCCUR REPEATEDLY)
	DATAOUT	NUMBER OF DISCRETE DATA ITEMS (OR VARIABLES) OUTPUT FROM THE SYSTEM (EACH ITEM MAY OCCUR REPEATEDLY)
	DATALOC	NUMBER OF DISCRETE DATA ITEMS (OR VARIABLES) LOCAL OR INTERNAL TO THE SYSTEM (EACH ITEM MAY OCCUR REPEATEDLY)
ARCHITECTURAL DESIGN	UNITS	NUMBER OF DISCRETE COMPILABLE UNITS, MODULES, PROCEDURES, OR PROCESSES PRESENT IN THE SYSTEM (I.E., FUNCTIONAL UNITS)
	PROGS	NUMBER OF SEPARATELY EXECUTABLE PROGRAMS OR TASKS PRESENT IN THE SYSTEM (INCLUDES ONE OR MORE UNITS)
	LEVELS	NUMBER OF DISTINCT LEVELS PRESENT IN THE DESIGN INVOCATION HIERARCHY (I.E., HEIGHT OF DESIGN)
	NODES	MAXIMUM NUMBER OF NODES PRESENT ON ANY LEVEL OF THE DESIGN INVOCATION HIERARCHY (I.E., WIDTH OF DESIGN)
	CALLAPP	TOTAL NUMBER OF CALLS TO APPLICATION MODULES (I.E., CALLS TO UNITS DEVELOPED SPECIFICALLY FOR THIS PROGRAM)
	CALLSYS	TOTAL NUMBER OF CALLS TO SYSTEM MODULES (I.E., CALLS TO PREEXISTING UNITS SUCH AS THE GRAPHIC EXECUTIVE SUPPORT SYSTEM (GESS) AND SYSTEM UTILITIES)
	COMUTL	TOTAL NUMBER OF COMPLEX (NOT TERMINAL NODE) UTILITIES (I.E., FAN-IN > 1 AND FAN-OUT > 0)
	COMGLO	TOTAL NUMBER OF COMMONs AND OTHER GLOBAL AREAS PRESENT IN THE SYSTEM
	EXTFILE	NUMBER OF EXTERNAL FILES ACCESSED BY THE SYSTEM (LOGICAL FILES, NOT ACCESSING UNITS)
DETAILED DESIGN	PDLSTMT	TOTAL NUMBER OF PROGRAM DESIGN LANGUAGE (PDL) STATEMENTS (EXCLUDING PROLOG DESCRIPTION) FROM ALL UNITS
	PDLDEC	TOTAL NUMBER OF PDL-DEFINED DECISIONS FROM ALL UNITS
	PROLOG	TOTAL NUMBER OF LINES OF PROLOG (UNIT DESCRIPTION) FROM ALL UNITS (EXCLUDING PDL)
	REUSED	TOTAL NUMBER OF REUSED PROCESSES, MODULES, AND UNITS (WHETHER OR NOT SUBSEQUENTLY MODIFIED)
DESIGN COMPLEXITY*	SCPX	STRUCTURAL (INTERMODULE) DESIGN COMPLEXITY
	LCPX	LOCAL (INTRAMODULE) DESIGN COMPLEXITY

* SEE SECTION 3.6 FOR A DISCUSSION OF THESE MEASURES.

0273 S7/3/4

Table 2-3. Data Summary for Detailed Design Data

PROJECT	TOTAL MODULES	PERCENT REUSED[a]	SIZE (KDLOC[b])	DATA COLLECTED	
				DESIGN DATA	COMPONENT SUMMARY
A	158	11	50	X	
B	203	34	49	X	
C	338	32	106	X	
D	259	84	37	X	
E	327	24	83	X	X
F	393	47	79	X	X
G	199	49	57	X	X
H	245	43	56	X	X

[a] PERCENT OF TOTAL MODULES.
[b] THOUSANDS OF DEVELOPED LINES OF CODE.

0273 S7/3/12

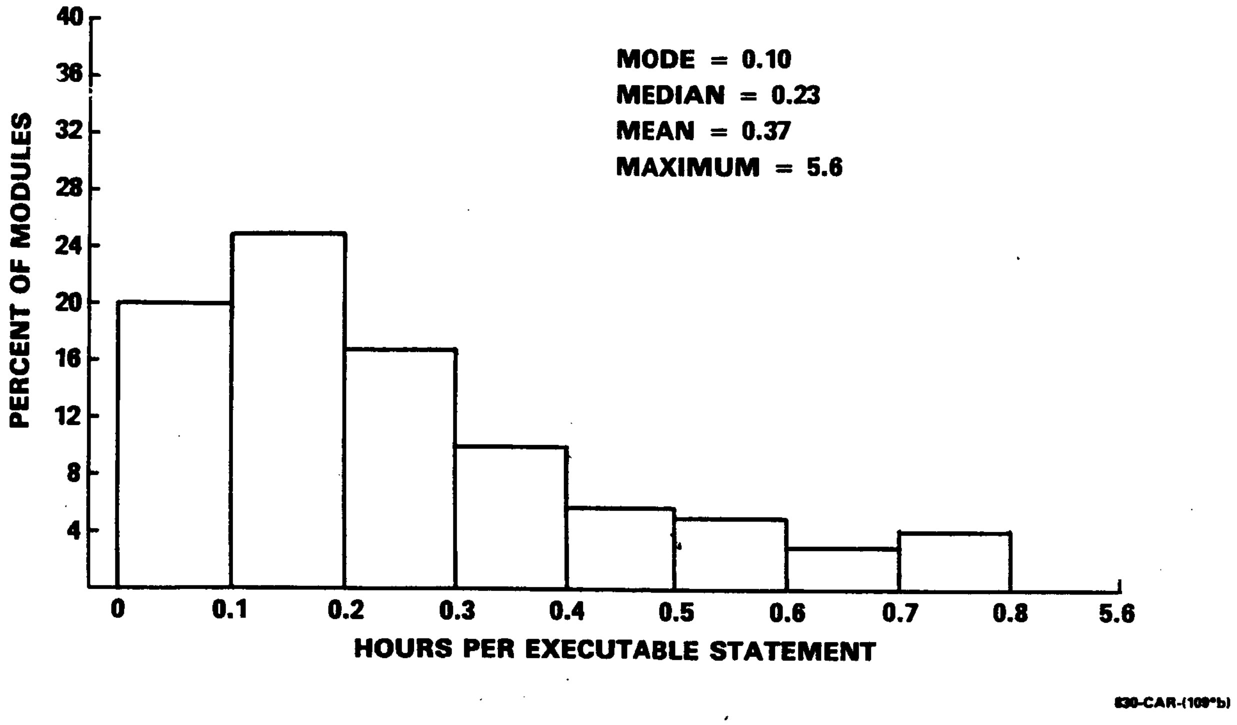


Figure 2-1. Distribution of Development Cost

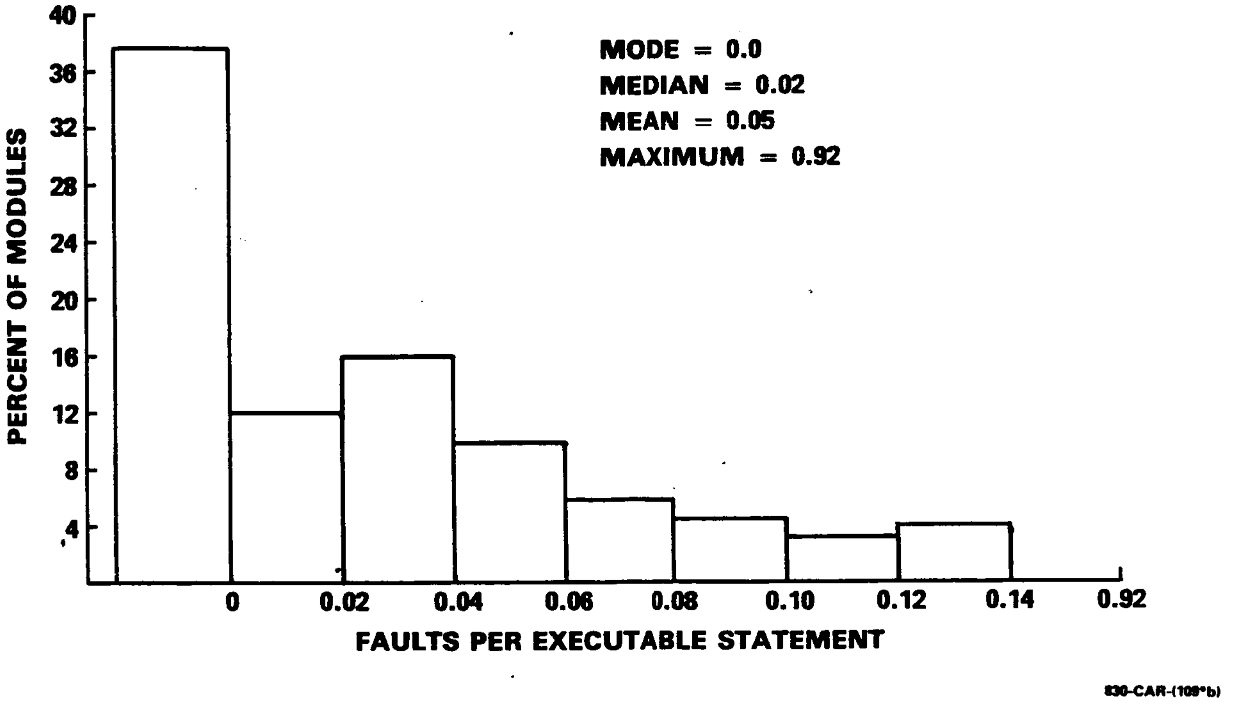


Figure 2-2. Distribution of Fault Rate

SECTION 3 - MEASUREMENT RESULTS

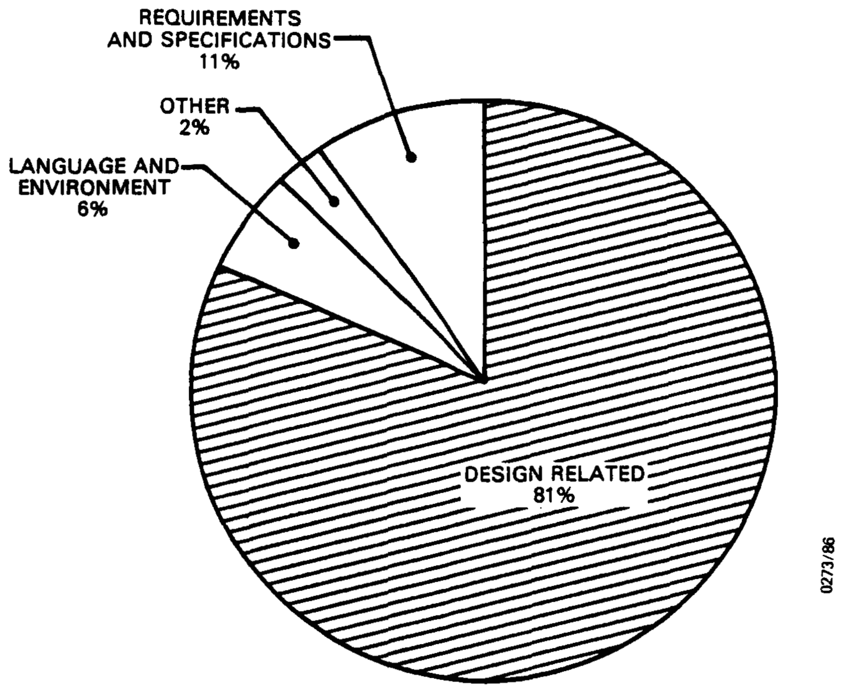
The design measure studies were conducted at both the project and module levels, using a wide range of techniques. The general areas of investigation were as follows:

- Defining design quality characteristics
- Identifying software types and patterns
- Quantifying software reuse
- Evaluating module design heuristics
  - Size/length
  - Strength/cohesion
  - Control coupling
  - Data coupling
- Assessing system design characteristics

This section describes the relevant results in each of these areas.

3.1 DESIGN QUALITY CHARACTERISTICS

One area of investigation focused on profiling the design process itself in terms of productivity, reliability, and maintainability as well as defining its effect on overall project quality. One study of three projects showed that design-related errors accounted for 74 to 82 percent of all nonclerical (nontranscription) errors reported (Reference 11). This percentage includes errors resulting from producing the design as well as those from misunderstanding the design. Figure 3-1 shows the distribution of errors for the median project from this sample. Only 6 percent were strictly programming errors (related to language or environment). Clearly, design quality is a principal factor in software reliability.



NOTE: EXCLUDES CLERICAL/TRANSCRIPTION ERRORS.

Figure 3-1. Nature of Software Errors

Another study attempted to use design characteristics to predict development, modification, and correction effort (Reference 12). Design cost and the number of variables (parameters) in the argument list (η_2^*) (Reference 13) were measured for new modules from six projects. The modules were ranked into quartiles for all five measures. Figure 3-2 shows the percent of modules in the upper (high) quartile of effort for each quartile of design cost (Figure 3-2a) and η_2^* (Figure 3-2b).

This information could be used to forecast the quality of modules in a system. At the end of the design phase, the η_2^* metric and the amount of effort spent in design (cost) are known. The modules in the upper quartile of design effort should be identified by a project manager for planning purposes, because 56 percent of these modules fell into the

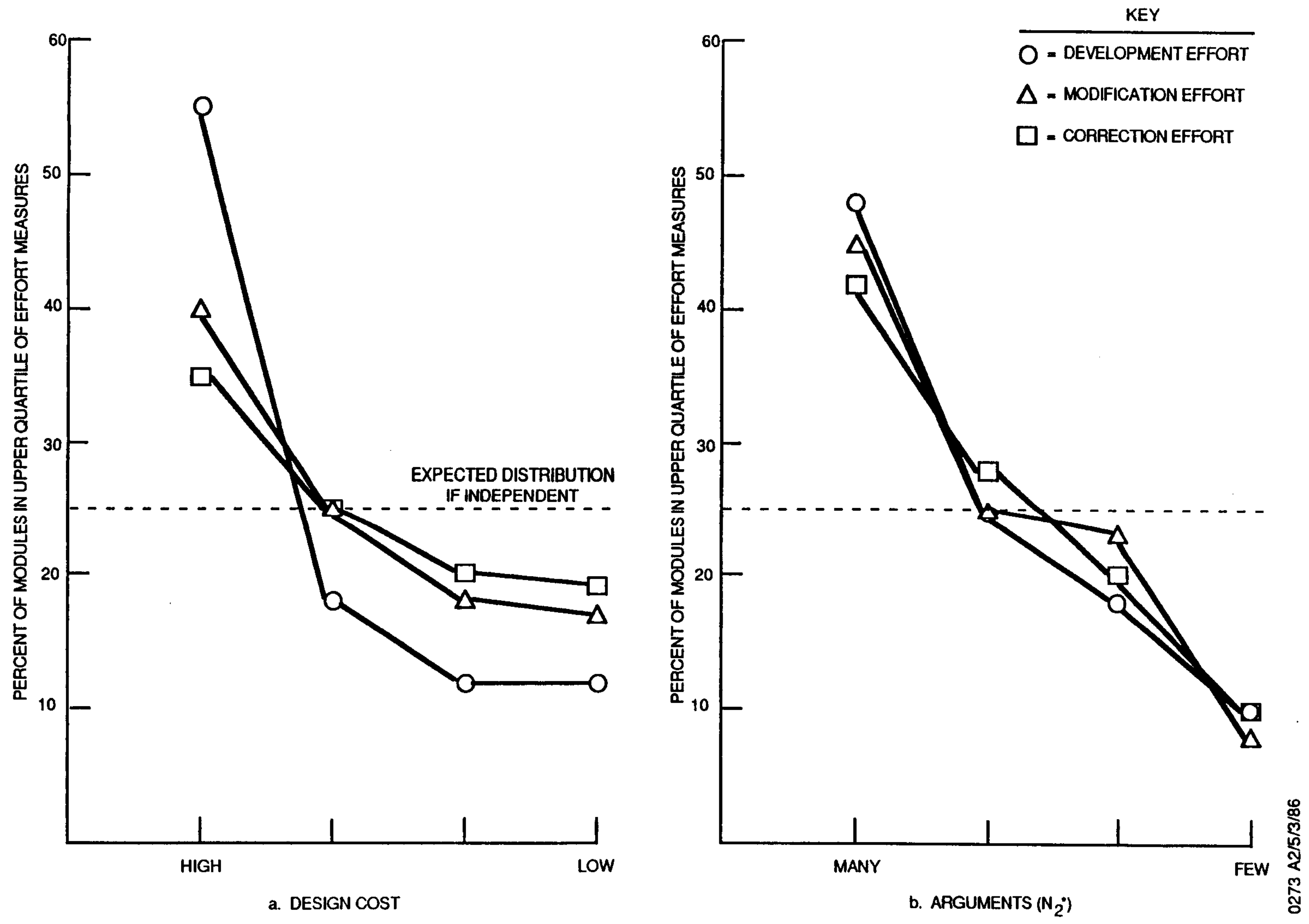


Figure 3-2. Relationships Among Design Characteristics

upper quartile of total development effort. Only 25 percent would be expected to fall into this class if assignments to classes were random. That is, in this environment, the modules in the upper quartile of design effort were more than twice (i.e., 0.56 ÷ 0.25) as likely as by chance to be the most expensive to develop overall.

Modules in the upper quartile of the η_2^* metric were almost twice as likely as by chance to require the most effort to develop, modify, and correct. It was also observed that it is easiest to identify modules that will have high development effort and most difficult to identify modules that will require little fault-correction effort. In addition, the metrics of design effort and η_2^* were seen to be reasonably similar in forecasting ability, except that η_2^* seems superior in identifying modules that will require little modification and correction effort.

An assumption must be made in order to use metric data from past projects to predict the quality of modules from a current project. The assumption is that the relationship between a module's current metric quartile and its eventual quality (i.e., development, modification, and correction effort) is the same as the relationship between the final metric quartiles of past projects' modules and their quality. This assumption is reasonable when using data from recent projects that are similar to the current project, and when predicting from metrics whose final quartiles are reasonably certain early in development. For example, the number of calling sequence arguments (η_2^*) in a module tends to remain relatively constant once specified in the design phase; consequently, the metric's value does not tend to change quartiles. It should be noted that the examples and metric data presented are from one particular environment; project data from other environments may differ.

3.2 SOFTWARE CLASSIFICATION

Another design approach looks for recurring patterns or types of software. Functionally similar systems and modules are expected to exhibit similar structure and quality characteristics. This relationship can be investigated by explicitly assigning types or seeking implicit types in the data.

3.2.1 PROJECT-LEVEL STUDIES

One set of studies (Reference 14) was based on a group of 29 projects that were explicitly classified based on a number of criteria. Table 3-1 shows the breakdown into four functional classes: scientific, data processing, support, and system software. Median values of 11 measures are reported for each class, with no statistically significant differences reported. However, the study also showed that software employing GESS (Reference 15) reused more code, contained larger modules, required more coding effort, and consumed more computer time. These results help managers plan future development projects more effectively.

A cluster analysis was performed using developed lines, average module size, computer use, percent of reused code, percent of design effort, and percent of test effort. This analysis divided the 29 projects into 2 well-defined clusters and a set of unique individual projects, as shown in Figure 3-3. Six of the original 29 projects dropped out of the cluster analysis because of missing data for at least 1 of the clustering variables.

Cluster 1 consists of attitude ground support system (AGSS) projects using GESS. Cluster 2 consists of data base, data preprocessor, data simulator, and flight software. The

Table 3-1. Characterization of Classes for Application Types

MEASURE	APPLICATION TYPE [a]			
	SCIENTIFIC	DATA PROCESSING	SUPPORT	SYSTEM
DEVELOPED LINES	49,272	26,199	15,012	30,481
AVERAGE MODULE SIZE [b]	185	81	140	91
PERCENT REUSED CODE	21.8	0.4	12.0	5.8
PERCENT DESIGN EFFORT	30.0	39.5	29.5	29.0
PERCENT CODE EFFORT	29.0	27.0	23.0	23.0
PERCENT TEST EFFORT	35.0	26.0	36.5	45.0
COMPUTER USE [c]	0.076	0.110	0.030	——
PRODUCTIVITY [d]	3.55	9.57	2.67	1.96
STABILITY [e]	0.036	0.024	0.058	0.080
MAINTAINABILITY [f]	47.0	59.5	31.0	31.0
SAMPLE SIZE	20	2	5	2

0273 S7/3/5

[a] MEDIAN VALUES FOR APPLICATION TYPE.
[b] SOURCE LINES OF CODE.
[c] COMPUTER HOURS PER DEVELOPED LINE OF CODE.
[d] DEVELOPED LINES OF CODE PER HOUR.
[e] CHANGES PER DEVELOPED LINE OF CODE.
[f] PERCENT OF CHANGES REQUIRING LESS THAN ONE HOUR.

remaining five projects demonstrated themselves to be unique in some way, as follows:

- V--Small project with large average module size
- D--Very high percent of reused code
- W--High percent of reused code and unusual distribution of effort due to research orientation
- K--Very large AGSS project
- L--Very small AGSS utility

This classification is similar to that determined by the GESS use/non-use criteria (Reference 14). Cluster 1 includes only GESS projects; however, some GESS projects also occur in Cluster 2 and among the unique projects.

3.2.2 MODULE-LEVEL STUDIES

Another set of studies (Reference 16) was based on individual modules taken from eight AGSSs. Because of the common

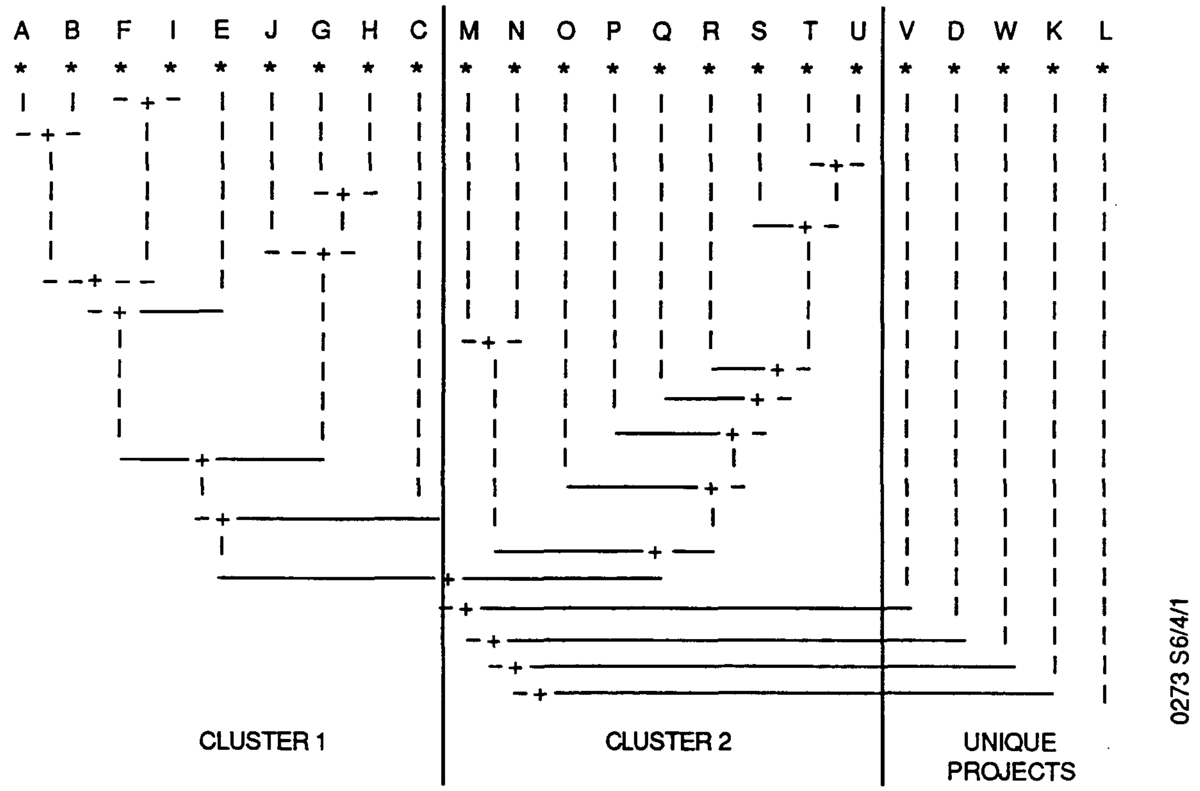


Figure 3-3. Cluster Analysis Tree Diagram

application, all projects incorporated the same four major functions: executive, telemetry processing, data handling, and attitude computation. Table 3-2 shows the percent of fault-free modules and the median cost for each class. The attitude computation class stands out as a unique group of software, exhibiting the highest percentage of fault-free modules but also the highest cost (in hours per executable statement). On the other hand, the "other" class shares that fault-free rate while demonstrating the second lowest cost. These data thus suggest that no direct relationship exists between cost and error rate.

The functions of individual modules (within subsystems) could be determined for some modules in four projects. The functional classes defined were input/output, algorithmic, and control. Table 3-3 shows that the three classes did not differ noticeably with respect to the fault-free rate. However, marked differences appear with respect to cost. Control modules required the most effort to develop; algorithmic modules required the least. These results are consistent with the intuition that logical decisions are harder to develop than assignment statements.

Another approach (Reference 16) taken to these data involved defining quality classes and then attempting to identify structural characteristics associated with the quality classes. Partitioning the modules on the basis of effort per executable statement (productivity) and faults per executable statement (reliability) produced two sets of three quality classes. Table 3-4 shows the median values for measures for each of these classes. Input/output (I/O) variables per executable statement and percent of assignment statements demonstrated the greatest variation corresponding to productivity. It was shown in Section 3.1 that the subset of I/O variables, η^*, could be used to predict module

Table 3-2. Quality by Subsystem Function

TYPE	NUMBER	FAULT-FREE (%)	MEDIAN HOURS PER EXECUTABLE STATEMENT
EXECUTIVE	63	62	0.27
TELEMETRY PROCESSING	424	58	0.25
DATA HANDLING	338	59	0.18
ATTITUDE COMPUTATION	472	77[a]	0.30[a]
OTHER	200	77[a]	0.22

[a] HIGHEST VALUE IN COLUMN.

0273 S7/3/6

Table 3-3. Quality by Module Function

TYPE	NUMBER	FAULT-FREE (%)	MEDIAN HOURS PER EXECUTABLE STATEMENT
INPUT/OUTPUT	19	58	0.20
ALGORITHMIC	99	58	0.16
CONTROL	96	57	0.24[a]

[a] HIGHEST VALUE IN COLUMN.

0273 S7/3/7

Table 3-4. Structural Characteristics of Quality Classes

QUALITY	CLASS	I/O VARIABLES[a]	DECISIONS[a]	ASSIGNMENT STATEMENTS[a]	CALL STATEMENTS[a]
PRODUCTIVITY	LOW	0.58	0.32	0.33	0.15
	MEDIUM	0.42	0.31	0.45	0.10
	HIGH	0.32	0.27	0.50	0.10
RELIABILITY	LOW	0.59	0.32	0.37	0.17
	MEDIUM	0.33	0.30	0.46	0.14
	HIGH[b]	0.39	0.29	0.43	0.09

[a] PER EXECUTABLE STATEMENT.
[b] ZERO FAULTS REPORTED.

0273 S7/3/8

cost. Only the percent of call statements appeared to fluctuate substantially with reliability.

3.3 SOFTWARE REUSE

Although not always recognized as such, an important design decision involves the reuse of existing software. Some good ideas have been developed on how to write reusable software; however, opportunities to reuse software must be recognized during the design activity. The goals of this analysis (Reference 17) were to identify the types of software that are reused in the flight dynamics environment and to quantify the benefits of software reuse.

Table 3-5 lists some of the characteristics of software reuse. Executable statements measure module size. Mean decisions measure module complexity. This table indicates that the modules that are reused without modification (old) tend to be small and simple (exhibiting a relatively low decision rate). A more detailed cross-classification (not shown) revealed that 55 percent of all old modules are high-strength algorithmic modules, the type likely to be found in mathematical software libraries. Table 3-5 shows that extensively modified modules tended to be the largest in terms of the number of executable statements.

Tables 3-6 and 3-7 clearly demonstrate the quality benefits of software reuse. Fully 98 percent of old modules proved to be fault free, and 82 percent of them fell into the lowest cost (per executable statement) category. Significant nonparametric (γ) correlations are associated with both of these relationships. (Percentages do not add to exactly 100 due to rounding.) These results are consistent with previous SEL studies of reused code (Reference 4), which indicated that reusing a line of code costs only 20 percent of the cost of developing it new. Because these four classes of software differ substantially with respect to structure and

Table 3-5. Characteristics of Reused Software

SOFTWARE TYPE	NUMBER OF MODULES	EXECUTABLE STATEMENTS	MEAN DECISIONS PER EXECUTABLE STATEMENT [a]
NEW	532	55	0.30
EXTENSIVELY MODIFIED	132	83	0.29
SLIGHTLY MODIFIED	163	51	0.25
OLD (UNCHANGED)	60	28	0.20

0273 S7/3/18

[a]NUMBER OF LOGICAL DECISIONS DIVIDED BY NUMBER OF EXECUTABLE STATEMENTS.

Table 3-6. Software Reuse and Development Cost

SOFTWARE TYPE	DEVELOPMENT COST (PERCENT)		
	LOW	MEDIUM	HIGH
NEW	42	28	30
EXTENSIVELY MODIFIED	52	27	22
SLIGHTLY MODIFIED	63	21	17
OLD (UNCHANGED)	82	8	10

0273 S7/3/19

NOTE: GAMMA (γ) = −0.33; PROBABILITY THAT $\gamma = 0$ IS LESS THAN 0.001.

quality measures, the subsequent module-level analyses are based on new modules only.

Table 3-7. Software Reuse and Fault Rate

SOFTWARE TYPE	FAULT RATE (PERCENT)		
	LOW	MEDIUM	HIGH
NEW	44	29	27
EXTENSIVELY MODIFIED	52	33	15
SLIGHTLY MODIFIED	69	17	14
OLD (UNCHANGED)	98	0	2

0273 S7/3/20

NOTE: GAMMA (γ) = −0.43; PROBABILITY THAT $\gamma = 0$ IS LESS THAN 0.001.

3.4 MODULE SIZE

Design modularization criteria based on limiting module size in terms of executable statements or lines of code have been proposed and used (Reference 18). For this analysis, 453 modules were classified into 3 approximately equal ordered groups on the basis of the number of executable statements in each module. Table 3-8 shows the results of this classification. The largest module in the sample contained 267 executable statements.

The dividing line of 31 executable statements (Table 3-8) is significant because, in the environment studied, it corresponds to about 60 source lines of code. Many programming standards limit module size to one page (or 50 to 60 source lines of code). The informal guideline used in this environment is that no module should exceed 2 pages (about 64 executable statements). Military standards on module size range from 50 to 200 executable statements (Reference 18). One purpose of the study was to test the validity of such standards, in general, and, in particular, to determine if the local guideline should be strengthened.

Table 3-8. Module Size Distribution

MODULE SIZE	NUMBER OF FORTRAN MODULES	EXECUTABLE STATEMENTS	MEAN DECISIONS PER EXECUTABLE STATEMENT
SMALL	154	1 TO 31	0.31
MEDIUM	148	32 TO 64	0.31
LARGE	151	65 OR MORE	0.32

0273 S7/3/10

A cross-tabulation of module size with development cost showed a correlation (γ) of -0.31. The probability of this correlation being due to random factors is less than 0.001. Although the magnitude of this correlation seems small, Figure 3-4 provides a better illustration of its importance. As the figure indicates, fully 46 percent of large modules fell into the lowest cost class, whereas just 22 percent of small modules were rated as low cost.

No significant relationship was found between module size and fault rate. Two recent studies (References 19 and 20) concluded that smaller modules were more fault prone. These studies, however, adopted parametric approaches to the statistical analysis. Sixty percent of the small modules in the sample contained no faults (Table 3-9). Nevertheless, this size class exhibited the highest average fault rate because a small module with even a single fault will show a very high fault rate (Figure 3-5). That may be the phenomenon detected by Basili and Perricone (Reference 19) and Shen et al. (Reference 20).

The effects of programmer performance and the possibility of an interaction between module size and module strength were subsequently considered in a more detailed analysis of these data (Reference 17) (Section 3.5). That consideration did not change the conclusion that larger modules cost less to

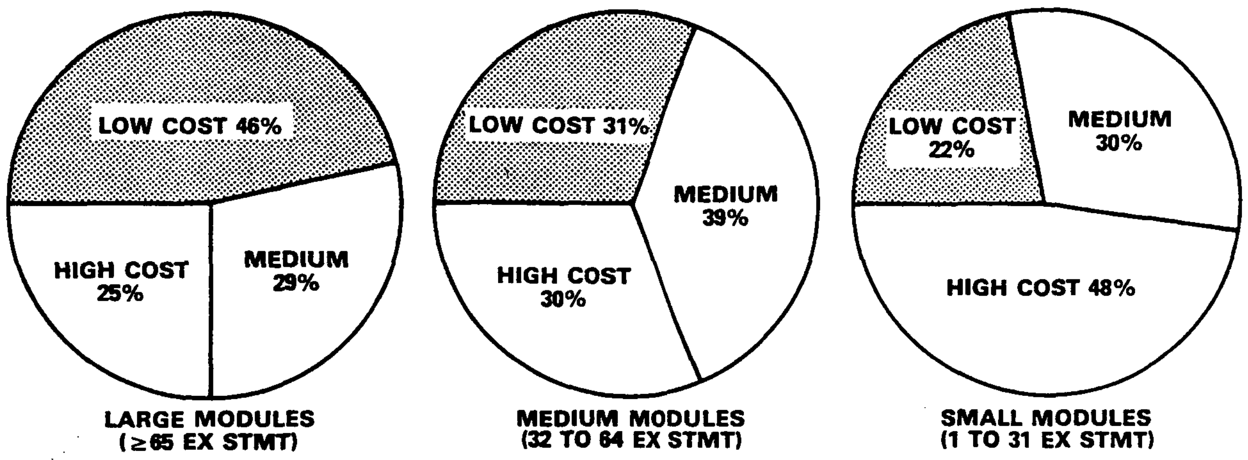


Figure 3-4. Development Cost for Classes of Module Size

Table 3-9. Two Fault Rate Statistics

SIZE	MODULES	FAULT RATE	
		MEAN	MEDIAN
SMALL	154	0.063	0.000
MEDIUM	148	0.051	0.024
LARGE	151	0.038	0.021

0273 S7313

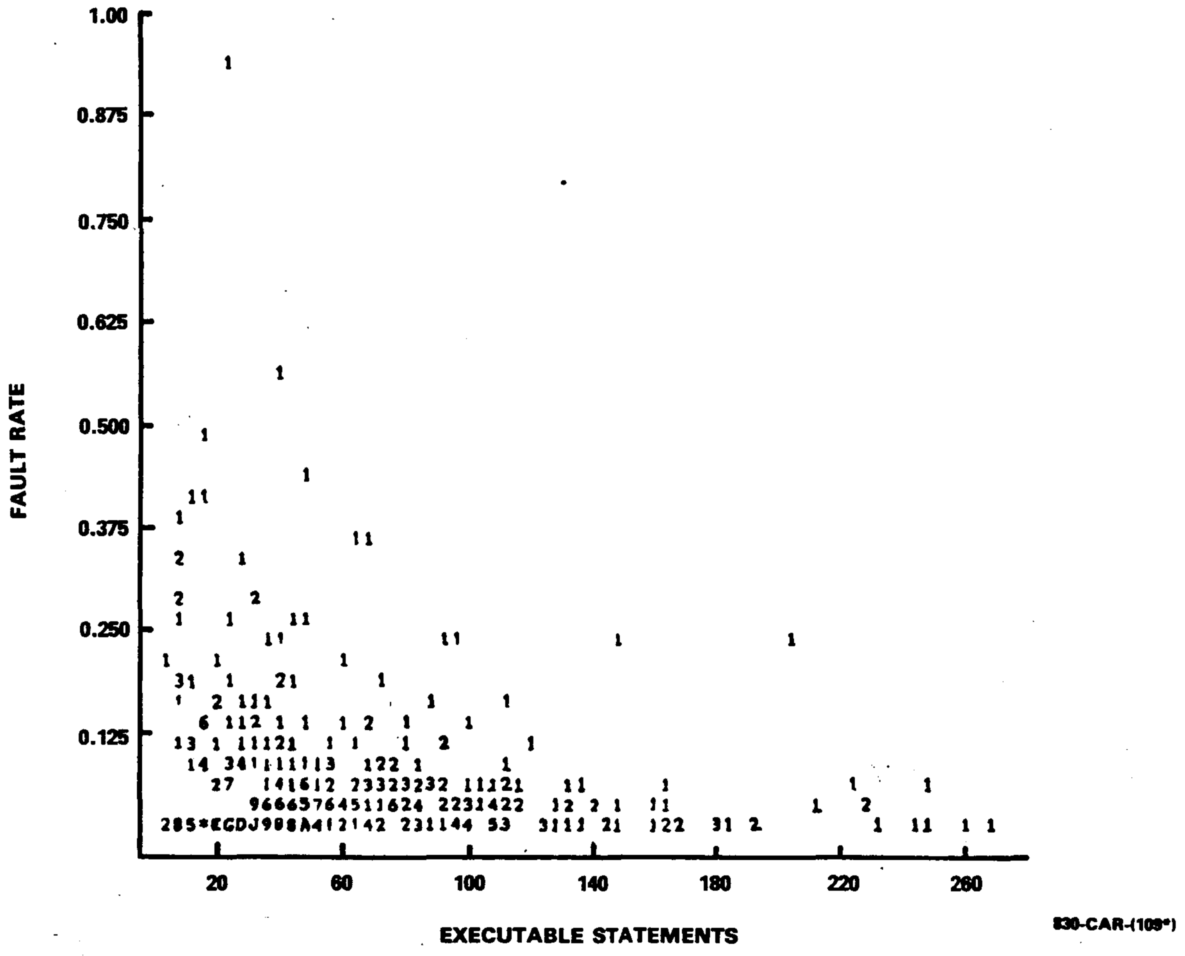


Figure 3-5. Module Size and Fault Rate

develop (per executable statement) than small ones. Module size does not appear to affect fault rate. A similar result has been reported for another class of software (Reference 21).

Other module size criteria have been proposed based on the concepts of software science (Reference 13) and cyclomatic complexity (Reference 22). A recent study evaluated the ability of these measures to predict development hours and errors (Reference 23). Ostensibly, size should be limited to reduce cost and errors. Table 3-10 shows that neither software science measures nor cyclomatic complexity performed as well as the simpler measure, source lines of code. That these measures are not strongly related to effort or errors suggests that they are not relevant criteria for defining appropriate module size.

Table 3-10. Software Science and Cyclomatic Complexity

MEASURE	CORRELATION	
	HOURS	ERRORS
SOFTWARE SCIENCE EFFORT	0.35	0.08
SOFTWARE SCIENCE BUGS	0.45	0.17
CYCLOMATIC COMPLEXITY	0.46	0.20
SOURCE LINES OF CODE	0.52	0.26

0273 S7/3/15

3.5 STRENGTH/COHESION

Myers (Reference 24) defines seven levels of module strength. In descending order, these are functional, informational, communicational, procedural, classical, logical, and coincidental. A high- (functional) strength module performs a single well-defined function. Myers contends that high-strength modules are superior to low-strength modules. Although it was not possible to test this theory exactly, a reasonable approximation was made. Some recent attempts to develop objective measures of module strength (References 25 and 26) seem promising, but are not (in their present forms) easily applied; consequently, they were not employed in this study (Reference 17).

Programmers determined the strength of a module via a checklist, rating each module they developed as performing one or more of the following functions: input/output, logic/control, and/or algorithmic processing. Distinguishing the types of functions seemed to be a less ambiguous task than identifying the number of functions, because the number of functions depends on the level of decomposition recognized by the respondent. Those modules described as having only one type of function were classified as high strength; those having two types, medium strength; and those having three types, low strength. Table 3-11 summarizes the results of this classification process.

A cross-tabulation of module strength with fault rate showed a correlation (γ) of -0.35. The probability that this correlation is due to random factors is less than 0.001. Again, a figure (Figure 3-6) provides a better indication of the magnitude of this correlation. Fifty percent of high-strength modules were fault free, whereas only 18 percent of low-strength modules were fault free.

No significant relationship was discovered between module strength and development cost. The effects of programmer performance and the possibility of an interaction between module size and module strength were subsequently considered in a more detailed analysis (Reference 27) of these data (Table 3-12).

Table 3-11. Module Strength Distribution

MODULE STRENGTH	NUMBER OF FORTRAN MODULES	MEAN EXECUTABLE STATEMENTS	MEAN DECISIONS PER EXECUTABLE STATEMENT
LOW	90	77	0.29
MEDIUM	176	60	0.32
HIGH	187	48	0.32

0273 S7/3/9

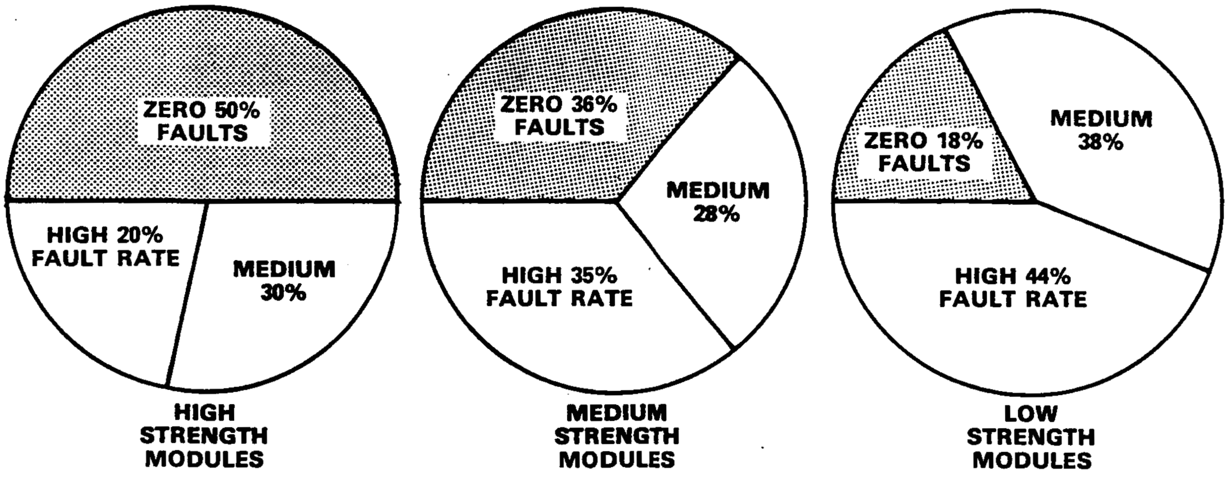


Figure 3-6. Fault Rate for Classes of Module Strength

Table 3-12. Contingency Table Results

CRITERIA	EFFECT CONTROLLED	CORRELATIONS[a]	
		FAULT RATE	COST RATE
MODULE STRENGTH	NONE	−0.35[b]	−0.19
	SIZE	−0.32[b]	−0.27[b]
	PROGRAMMER	−0.21	0.10
MODULE SIZE	NONE	0.20	−0.31[b]
	STRENGTH	0.19	−0.38[b]
	PROGRAMMER	0.27[b]	−0.41[b]

[a] GAMMA (γ) STATISTIC.
[b] PROBABILITY IS LESS THAN 0.001 THAT CORRELATION IS ACTUALLY ZERO.

Controlling for module size, the correlation between module strength and cost increases from -0.19 to -0.27 and becomes significant. Controlling for module strength, the correlation between module size and cost increases from -0.31 to -0.38. These results imply that, overall, high-strength modules (usually small) tend to be low cost but that large modules also tend to be low cost (independent of module strength).

Thus far, the potential effects of programmer performance were ignored. Lines 3 and 6 of Table 3-12 show the correlations between the modularization criteria and quality measures obtained while controlling for the effect of programmer performance. (The interaction of module size and strength is, however, no longer controlled.) The large changes from the initial correlations demonstrate that programmer performance interacts with both module size and strength. The disappearance of the significance of the relationships between module strength and module cost and fault rate indicates that these relationships exist because high-strength modules are associated with programmers who produce modules that cost less and have low module fault rates.

The effect of programmer performance was also examined in a subsequent analysis. Of the 26 programmers in the sample, 16 developed 9 or more modules. Together these programmers accounted for 413 of the total 453 modules. The performance of these programmers was reanalyzed using nonparametric correlation to better define the relationship of programmer performance to modularization criteria.

For each of these programmers, the percent of zero-fault and low-cost modules was computed. The correlation (by programmer) between module strength and percent of zero-fault modules was -0.53 (probability less than 0.05 that this correlation is due to random factors). Figure 3-7 shows two clusters of programmers. Those who produce low-fault-rate modules (i.e., good programmers) tend to produce high-strength modules. Good programmers do not, however, appear to have any preference for a particular module size. The lower significance levels associated with the correlation coefficients result from the reduction in sample size produced by studying 16 programmers instead of 453 modules.

Finding that programmer performance accounts for some of the strength of these relationships does not affect their validity. This result does, however, highlight the difficulty of separating the effects of programmer performance from those of technology or methodology (Reference 28). Furthermore, it enables us to learn about software development as Soloway (Reference 29) prescribes, by observing what good programmers do.

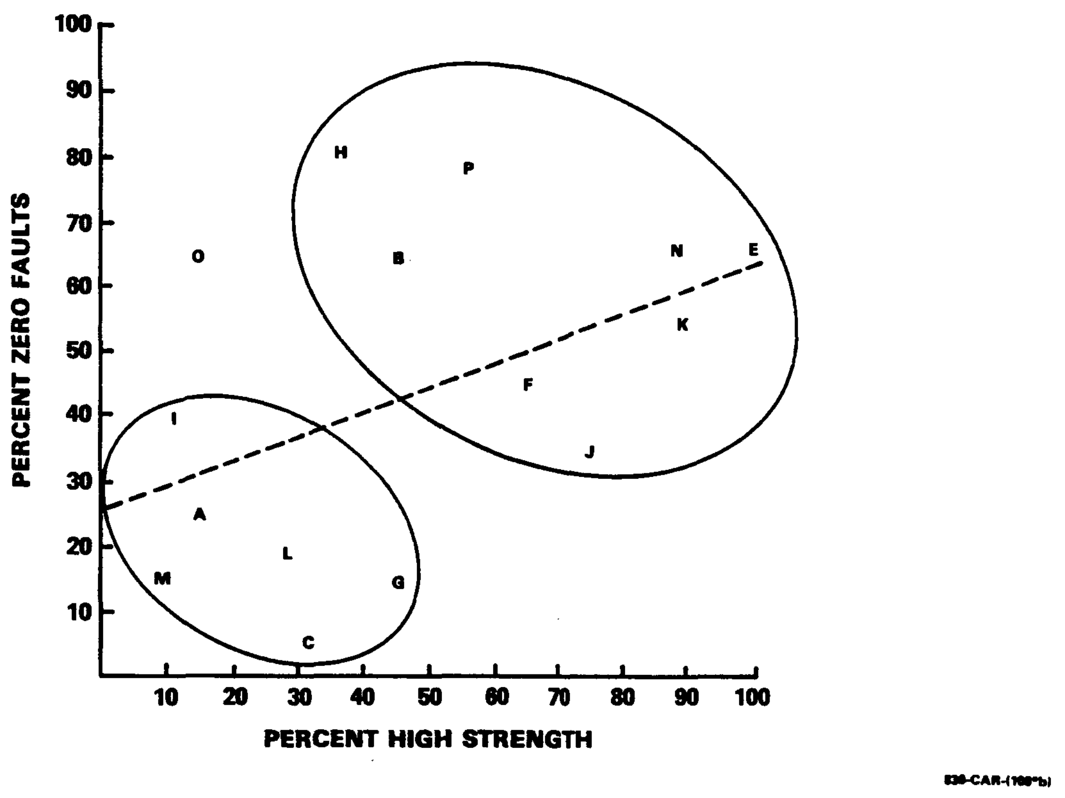


Figure 3-7. Module Strength and Faults by Programmer

3.6 CONTROL COUPLING

Two modules have a control coupling when one invokes the other. One commonly accepted design rule is that no module should call too many (i.e., more than seven) other modules. Furthermore, a module that calls only one other module might just as well include the other module's function within itself. One formulation of this concept is an adaptation of the "7 $\pm$ 2 rule" (Reference 30), which states that each module should call from five to nine other modules, except in the case of terminal nodes. (Calls to system services are excluded from these counts of calls.)

For this analysis (Reference 16), the modules were grouped into three ordered classes with respect to the number of descendants: one, two to seven, and more than seven. (Terminal nodes were not included in this analysis.) The results of cross-classification indicate that modules with more descendants tend to cost more (per executable statement) to develop ($\gamma = 0.25$) and have a higher fault rate ($\gamma = 0.33$). The probability of these correlations being due to chance is less than 0.01.

Figure 3-8 illustrates the magnitude of the difference among classes for fault rate. Only 12 percent of modules with more than seven descendants were fault free, whereas 42 percent of modules with only one descendant were fault free. Apparently, the simpler the invocation structure in a module, the better. This measure of structural complexity has proven to be clearly related to fault rate (faults per executable statement), whereas more complex measures based on counts of decisions and operators have not (Reference 23). On the other hand, the total number of faults appears to increase with size, decisions, operators, etc. Earlier SEL studies dealing with descendant span (e.g., Reference 23) were handicapped because they did not distinguish between calls to system/library routines and calls to other application modules. The design analysis tool (Reference 9) developed for this study remedies that deficiency.

These results led to the development of a general design complexity model based on counts of fanout, I/O variables, and total modules (Reference 31). The model includes two components: local (intramodule) complexity and structural (intermodule) complexity. It is defined by the following equations:

$$C = S + L \qquad (3\text{-}1)$$

where C = relative complexity

S = structural complexity

L = local complexity

$$L = \frac{\Sigma \frac{V_i}{f_i + 1}}{n} \qquad (3-2)$$

where L = local (intramodule) complexity

V_i = I/O variables in module "i"

f_i = fanout of module "i"

n = number of modules in system

$$S = \frac{\Sigma f_i^2}{n} \qquad (3-3)$$

where S = structural (intermodule) complexity

f_i = fanout of module "i"

n = of modules in system

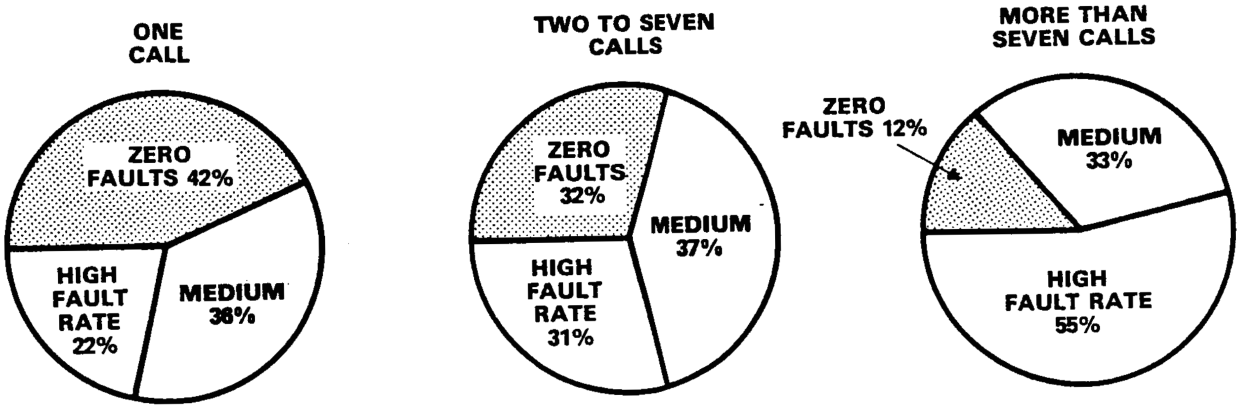


Figure 3-8. Fault Rate by Descendent Count

Reference 31 fully defines these measures and explains their derivation. Figures 3-9 and 3-10 show the correlations of relative complexity with error rate and productivity, respectively, for eight large projects. As indicated in Section 3.1, the bulk of software errors are design related, so it is reasonable to find a strong relationship between design complexity and error rate, as shown in Figure 3-9. On the other hand, many external factors have been shown to influence productivity (Reference 27), leading to the weak relationship shown in Figure 3-10. An implication of this model is that complexity is minimized when fanout is evenly distributed across all modules (except terminal nodes, of course).

3.7 DATA COUPLING

Two modules have a data coupling when data flows from one to the other or is shared in COMMON. The two issues studied by the SEL were the effects of different coupling mechanisms and the strength of coupling.

FORTRAN modules can be coupled directly in two ways: through calling sequence parameters or through COMMON block variables. (A COMMON block is a global data area.) Some authors have argued against COMMON coupling (Reference 32), even if it results in long and unwieldy calling sequences. A design measure was devised to evaluate that argument.

For this analysis (Reference 16), the modules were grouped into three ordered classes with respect to the percentage of referenced I/O variables in COMMON: zero, ≤15 percent, and >15 percent. (All calling sequence arguments count as referenced I/O variables.) No relationship was observed between fault rate and coupling. Figure 3-11 illustrates this; the percentage of zero-fault modules is about the same for both parameter and COMMON coupled modules. Earlier recommendations that COMMON coupling was best avoided may have been based on experience before the general availability of "INCLUDE" processors. In an environment where only a single

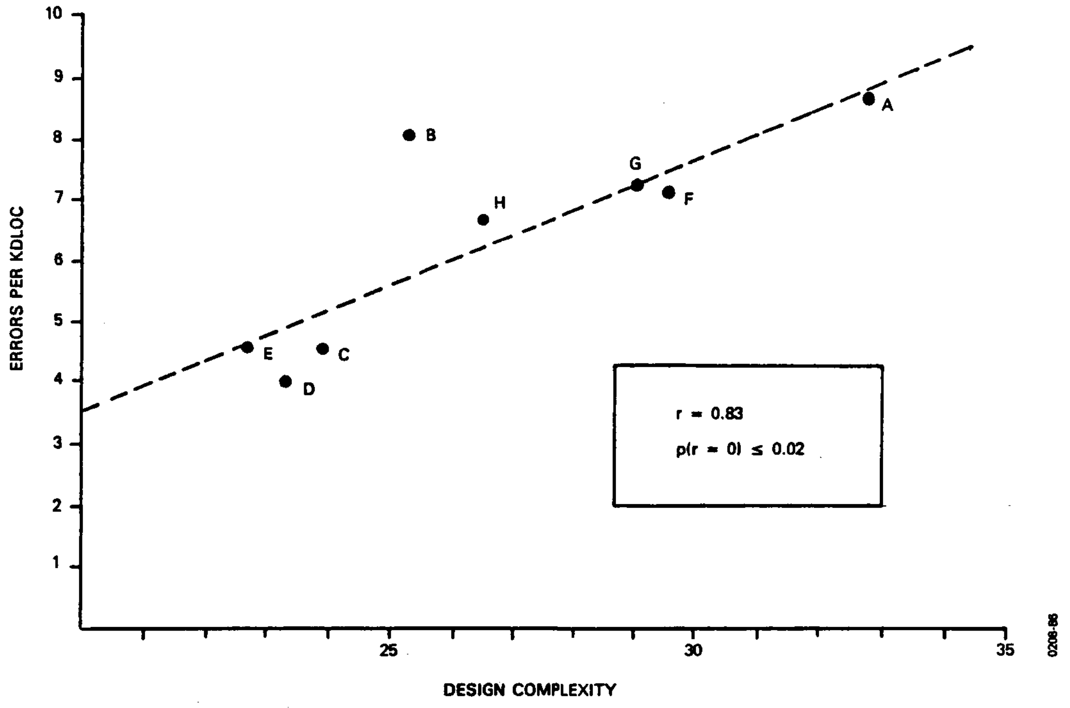


Figure 3-9. Relationship to Error Rate

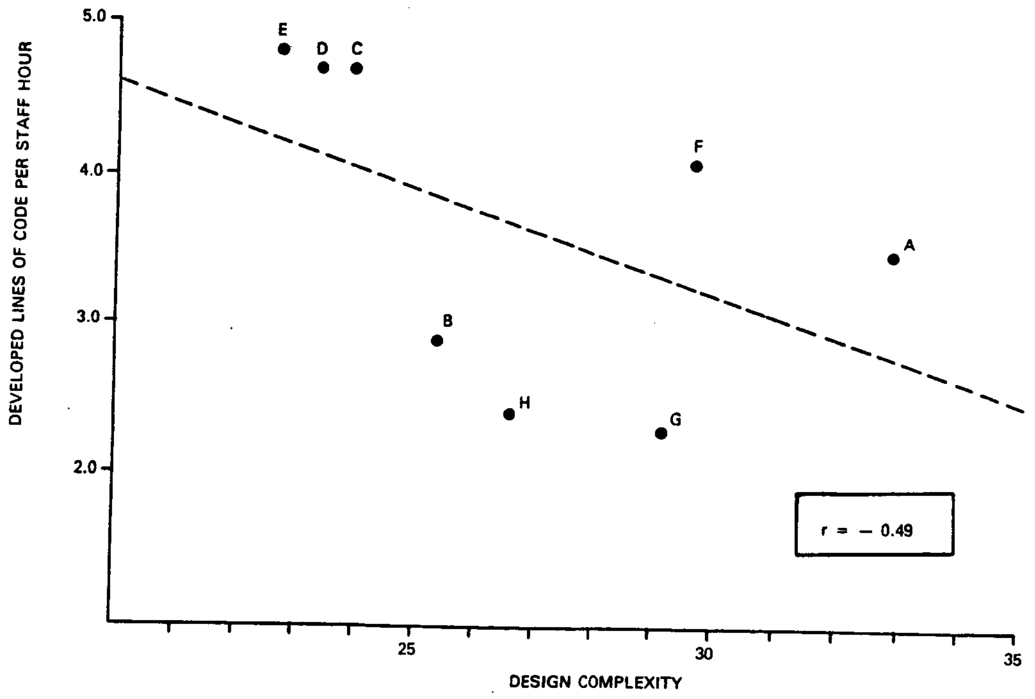


Figure 3-10. Relationship to Productivity

version of a COMMON block definition needs to be maintained, COMMON coupling is an acceptable, and sometimes preferable, alternative to parameter coupling. Furthermore, another study (Reference 33) failed to shown any significant difference between global and parameter coupling with respect to modifiability.

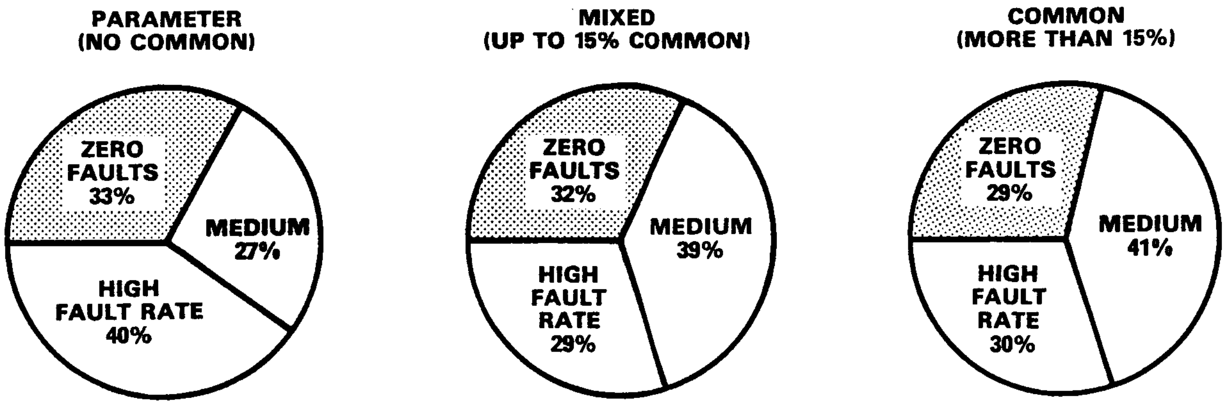
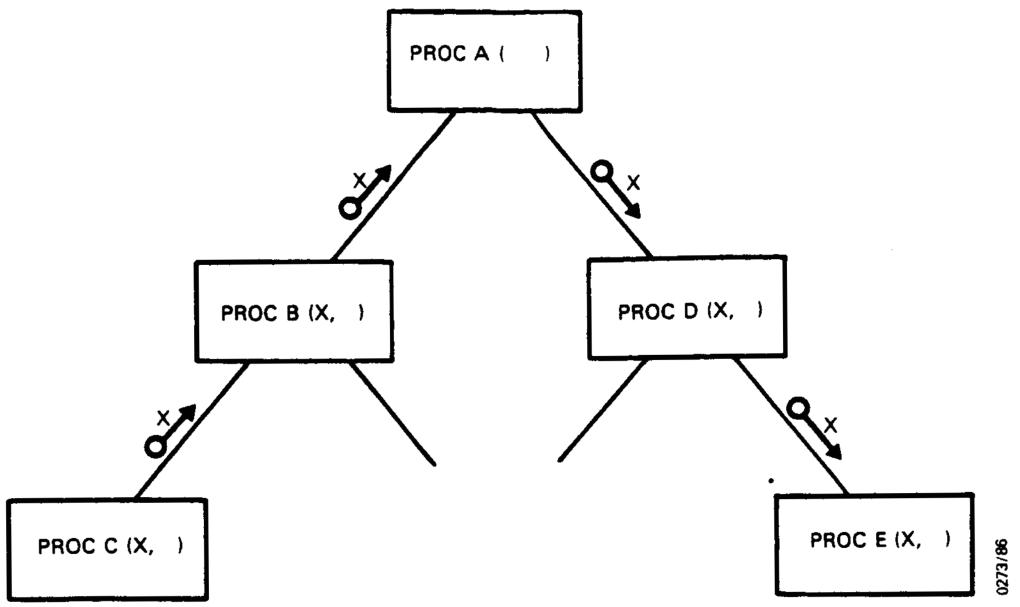


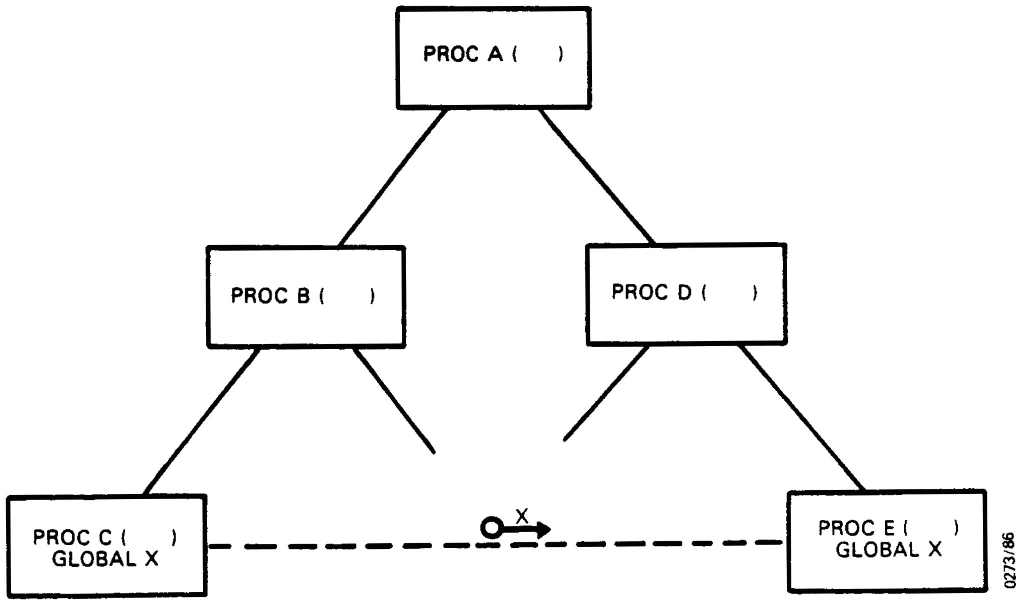
Figure 3-11. Fault Rate by Coupling Type

The SEL complexity model (Reference 31) described in Section 3.6 implies that the strength of data coupling, in terms of the number of variables shared, should be minimized. Rigorous application of the principle of information hiding (Reference 34) should reduce variable repetition and, hence, local complexity.

Figure 3-12 shows two design parts of equal structural complexity (according to the SEL complexity model): the number and distribution of fanouts are identical. Figure 3-12a traces a variable through a design following strict top-down decomposition rules. The variable appears in the higher level modules (A, B, D) as well as the lower level modules (C, E) that actually use it. Figure 3-12b shows an alternative design with a horizontal transfer of data that bypasses



(a) STRICT TOP-DOWN STRUCTURED DESIGN



(b) LOWER COMPLEXITY WITH LATERAL TRANSFER

Figure 3-12. Reducing Variable Repetition To Minimize Complexity

the higher level modules. The local complexity of the intermediate modules (B, D) in the strict top-down configuration (Figure 3-12a) exceeds their counterpart in the alternative design (Figure 3-12b) because their counts of I/O variables are larger. This complexity model emphasizes the number of data couples rather than the nature of the coupling mechanism.

Rotenstreich and Howden (Reference 35) argue that both horizontal and vertical data flow are essential to good design. The appropriate use of horizontal transfers prevents data flows from violating levels of abstraction. COMMON blocks provide the only mechanism for horizontal data transfers in FORTRAN. Figure 3-12 shows that horizontal data flows can reduce the magnitude of the local complexity measure in some situations.

Of course, a less complex design might also be produced by partitioning the work differently and restructuring this design. For example, PROC C could be invoked directly by PROC E (if the nature of the problem permitted). However, this simpler structure would still be reflected in lower values of the local complexity measures defined by this model.

3.8 SYSTEM STRUCTURE

Another study (Reference 36) looked at how a system can be organized into subsystems and how subsystems can communicate with each other and the external environment. Six hypotheses were studied, as follows:

- Complex Utilities--Complex utilities (modules with fanin greater than one and fanout greater than zero) adversely affect productivity and error rate.
- File Access--Confining I/O operations to one module per file reduces the potential for error.

- Programs--Allocating distinct functions to separate executable images improves productivity and reduces the potential for error.
- COMMON Usage--Limiting COMMON block usage to one block per subsystem and file reduces the potential for error.
- Hierarchical Level--Design structures with deep calling sequence trees are error prone.
- Nonexecutable Components--Systems with a high percentage of nonexecutable components are error prone.

The analyses included data on eight flight dynamics projects from which design data had been obtained via the analyzer program (Reference 9) plus one other project for which only incomplete data were available. Regardless of the initial hypothesis statement, all design characteristics were evaluated with respect to their effects on both productivity and reliability.

Table 3-13 shows the correlations of the structure measures (derived from the hypotheses) with the quality measures. None of the design characteristics studied demonstrated a statistically significant relationship to productivity or error rate. This does not mean that these characteristics are not important; rather, it implies that their effects are not general. For example, the use of a complex utility may be appropriate in one design context and inappropriate in another.

It was not possible, however, to fully study all the design hypotheses of interest with data extracted from the "design as implemented in code." Much important design information is not expressed in the code.

Table 3-13. System Structure and Quality

STRUCTURE MEASURE	AVERAGE VALUE	SAMPLE SIZE	CORRELATIONS	
			PRODUCTIVITY	ERROR RATE
PERCENT COMPLEX UTILITIES	10	8	–0.61	0.46
FILES PER SUBSYSTEM	1.5	9	0.19	0.31
SUBSYSTEMS PER PROGRAM	5.1	9	–0.01	–0.54
PERCENT COMMONs CROSSING SUBSYSTEMS	48	8	0.64	–0.20
MEAN MODULE LEVEL	6	8	–0.12	–0.21
PERCENT NONEXECUTABLE COMPONENTS	18	9	0.07	0.36

0273 S7/3/16

SECTION 4 - DESIGN METHOD STUDIES

In addition to the measurement activities associated with the existing flight dynamics design process, the SEL has undertaken a number of studies of alternative design methods. These studies have produced some additional measurement data, as described in this section.

4.1 OBJECT-ORIENTED DESIGN

A software object is an abstract model of a problem domain entity. Objects are packages of both data and operations on that data (References 37 and 38). The Ada[1] package construct is representative of this general notion of an object. Object-oriented design is the technique of using objects as the basic unit of modularity in system design.

The suitability and adaptability of Ada for flight dynamics software development is currently being evaluated. The SEL is involved in a pilot project to develop a satellite dynamics simulator in Ada using object-oriented methods (Reference 39). NASA/GSFC and CSC are cosponsors of the experiment, which is supported by personnel from all three SEL participating organizations.

The objective of the overall experiment is to determine the effectiveness of Ada for flight dynamics software development at NASA/GSFC. The experiment, begun in January 1985, consists of the parallel development, in FORTRAN and Ada, of the attitude dynamics simulator for the Gamma Ray Observatory (GRO) spacecraft. When completed, the system is expected to comprise approximately 40,000 source lines of code to execute on a DEC VAX-11/780 computer. Additional information about the experiment is presented in Reference 40.

---

[1]Ada is a registered trademark of the U.S. Government (Ada Joint Program Office).

This project has resulted in the development of a new design notation and formal procedures for integrating object-oriented design into the software development life cycle (Reference 41).

The principal design abstractions incorporated in this procedure are the state machine abstraction (Reference 42) and the representation of the system according to the orthogonal views of a seniority hierarchy and a parent-child hierarchy (Reference 43). The design is expressed as a set of object diagrams. The state machines are conveniently implemented as Ada packages consisting of internal state data and a group of related procedures that operate on that state data.

An instance of an object diagram is shown in Figure 4-1. The figure indicates that the OBC package is <u>senior</u> to the Truth Model package. The arc between the two packages shows that OBC uses operations (subprograms) of the Truth Model. Arcs do not go from a package to one that is above it. In this way, each diagram expresses the relative seniority of the packages. The orthogonal parent-child (or inclusion) hierarchy provides for a package (like one of those in Figure 4-1) to be represented on a separate diagram in terms of its constituent elements; for example, subprograms, other packages, and state data.

As shown in the example, these object diagrams integrate concepts from several other methodologies (References 38, 43, 44, and 45). Their use provides the following:

- A design notation that maps into Ada, thus providing a composite mapping from a specification to Ada software
- A design notation flexible enough to represent both traditional structured designs and nonhierarchical designs

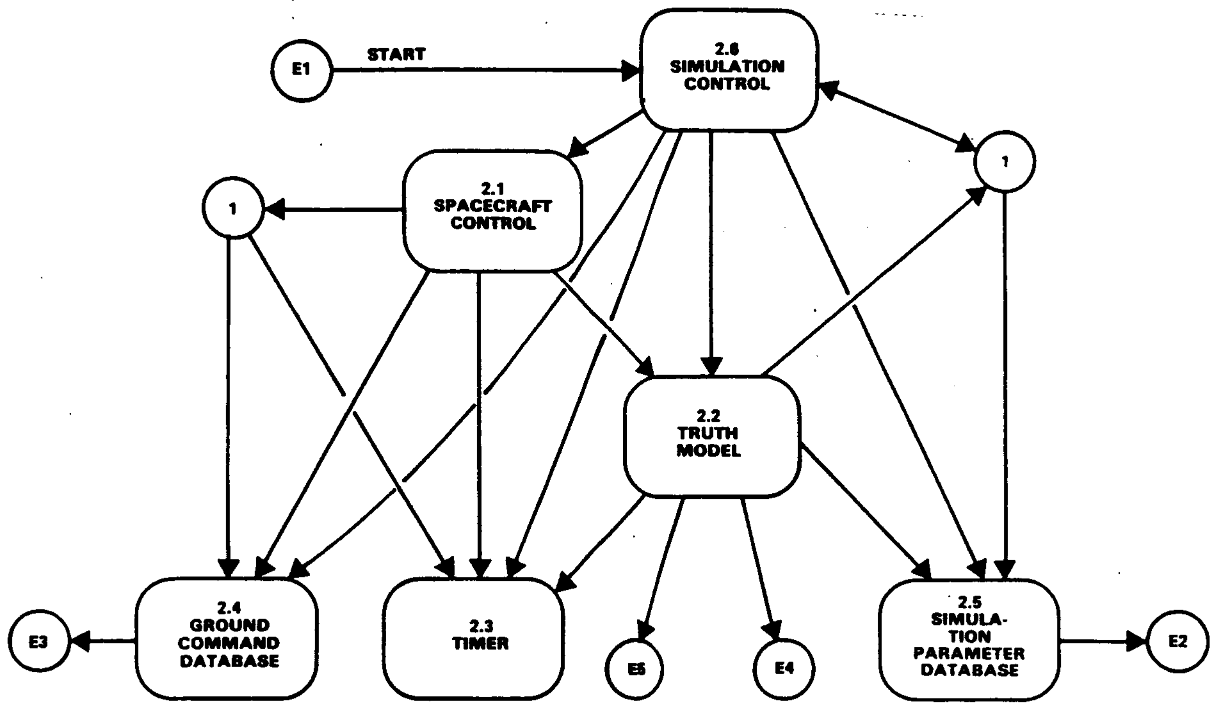


Figure 4-1. Ada Design: Seniority Hierarchy of Packages

- Criteria for partitioning a software system into modules and for choosing direction of control
- Support for walk-throughs and iterative refinement of a design through the use of graphical notation for both the specification and the design

These concepts form an integral part of an object-oriented software development life cycle. Studies are now under way to determine how object-oriented concepts can be used in other phases of the life cycle, such as specification and testing. When complete, this synthesis should produce a general object-oriented development methodology.

4.2 FORTRAN/ADA-ORIENTED DESIGN COMPARISON

Some early experiences using Ada for scientific applications (e.g., Reference 46) showed that the design of the Ada system "looked like a FORTRAN design." As part of an experiment on the effectiveness of Ada (Reference 40), the experiment planners identified the following factors that were believed to be prerequisites for obtaining a new design, one that would take full advantage of Ada features:

- The opportunity to set aside previous designs for the system and work directly from system requirements
- Training in design methods that exploit Ada's capabilities
- Encouragement to explore these new design methods

The purpose of this analysis was to address the following question:

> When these prerequisites were satisfied, was a different design produced?

In this analysis, a FORTRAN design was compared with an Ada-oriented design for the same system. In this high-level look at each design, the overall system structure, the external and internal data flows, and the control flow are discussed. Some simple quantitative measures are extracted from each design.

4.2.1 SYSTEM STRUCTURE

A top-level system diagram for each design is shown in Figures 4-2 and 4-3. To facilitate comparison, the identical system input and output objects are placed at the top and bottom, respectively, of each figure. The FORTRAN system consists of the five subsystems in the middle of Figure 4-2. The Ada system is the product of a design method (Section 4.1) that differs from the FORTRAN team method. So, although "subsystem" will be used to refer to the major Ada units, they are, in fact, Ada packages. Furthermore, the Simulation Support Subsystem in Figure 4-3 is really a collection of three Ada packages for the simulation timer,

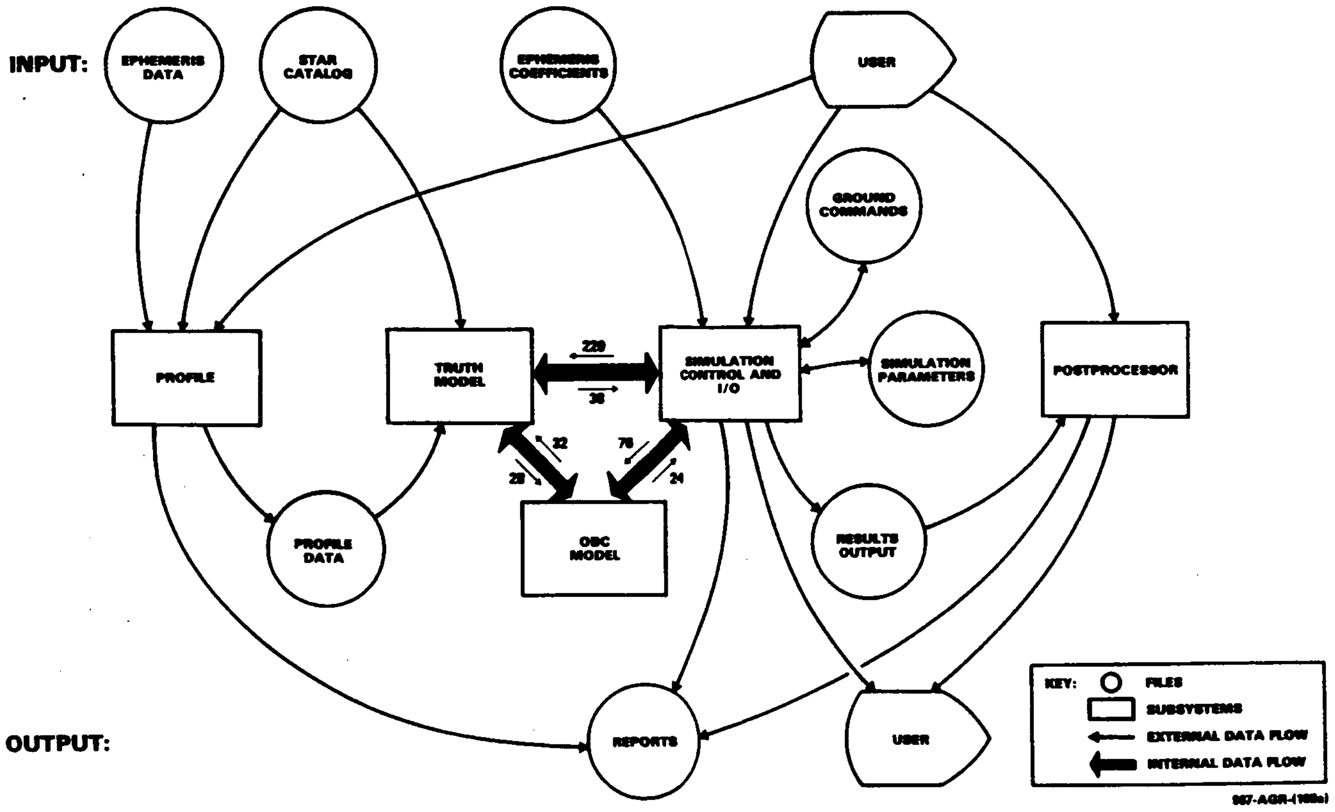


Figure 4-2. FORTRAN System Diagram

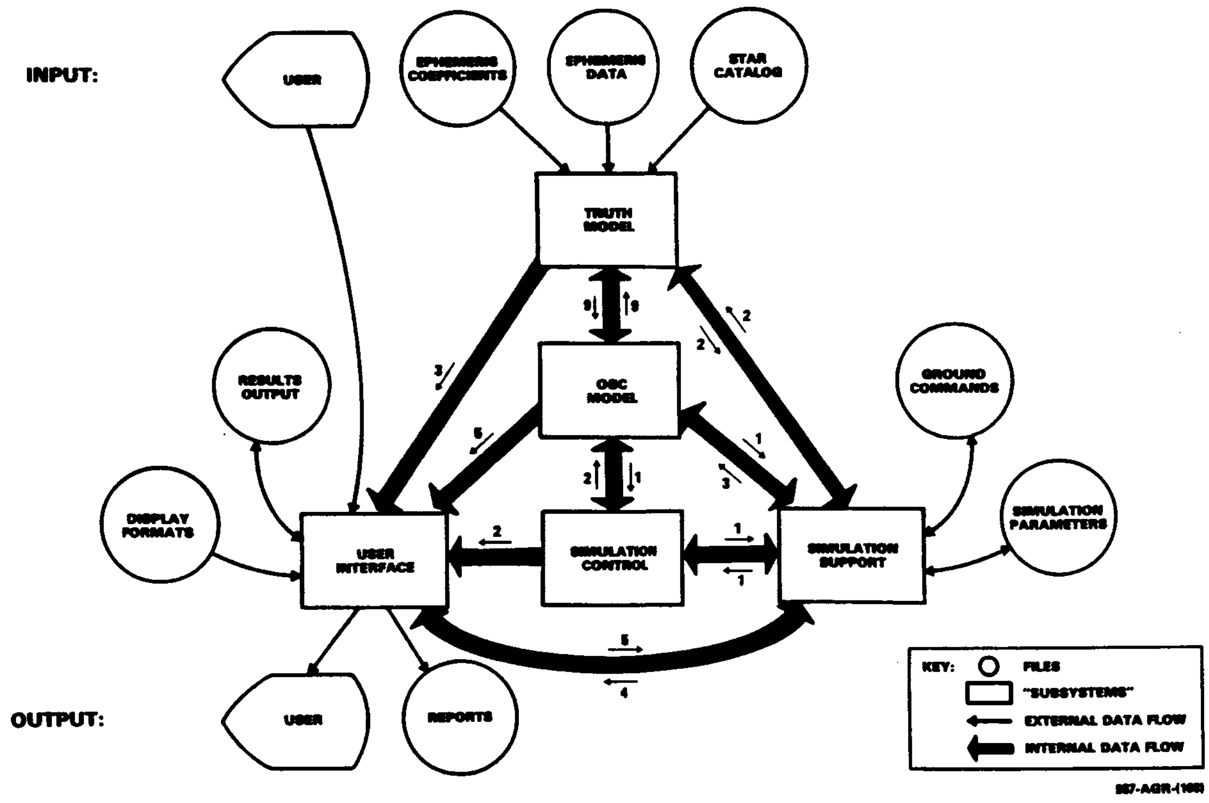


Figure 4-3. Ada System Diagram

parameters, and ground commands. The Ada system appears in Figure 4-3 as five subsystems only to invite comparison with FORTRAN regarding the high-level data flow.

The FORTRAN system is composed of three distinct programs: Profile, Postprocessor, and Simulator (Truth Model, OBC Model, and Simulation Control-I/O). As separate programs, each interacts with the user, as shown by the external data flows in Figure 4-2. The assignment of processing functions to each subsystem is shown in Table 4-1 for both the FORTRAN and Ada systems.

Table 4-1. Allocation of Functions Among Subsystems

NUMBER OF SUBROUTINES	FORTRAN SYSTEM	FUNCTION	ADA SYSTEM	NUMBER OF SUBPROGRAMS
13	POSTPROCESSOR	ANALYZE RESULTS	USER INTERFACE	47
86	SIMULATION CONTROL AND I/O	INTERACT WITH USER		
		UPDATE PARAMETERS	SIMULATION SUPPORT	13
		SCHEDULE GROUND COMMANDS		
		MAINTAIN SIMULATED TIME		
		CONTROL SIMULATION	SIMULATION CONTROL	3
58	PROFILE	CALCULATE ATTITUDE - INDEPENDENT PROFILE (ENVIRONMENTAL TORQUES, EPHEMERIDES, ETC.)	TRUTH MODEL	102
46	TRUTH MODEL	INTEGRATE EQUATIONS OF MOTION		
		MODEL SENSORS AND ACTUATORS		
59	OBC MODEL	SIMULATE GRO ATTITUDE DETERMINATION	OBC MODEL	87
		MODEL CONTROL LAWS OF ONBOARD COMPUTER		
		MODEL BACKUP CONTROL PROCESSING ELECTRONICS		
262 TOTAL SUBROUTINES				252 TOTAL SUBPROGRAMS

0273 S7/3/17

The Ada system is designed as a single program, with each subsystem performing the functions listed in Table 4-1. The OBC Model is functionally similar to its FORTRAN counterpart. The Ada Truth Model incorporates the processing performed in the FORTRAN Profile in addition to the FORTRAN Truth Model. (The FORTRAN user has the option of choosing not to use Profile and having those calculations performed in the Truth

Model, thereby mirroring the Ada design.) The Ada design pulls apart the simulation control functions from the User Interface; these processing elements are combined in the FORTRAN design. However, the User Interface in Ada includes the results processing that, in FORTRAN, is delegated to a separate program, the Postprocessor. Both designs have major units named Truth Model and OBC Model to reflect the underlying control problem.

4.2.2 EXTERNAL DATA FLOW

Both designs in Figures 4-2 and 4-3 show communication with nine external objects (files or devices). Eight of the nine are identical, the difference being the profile data file in FORTRAN and the display format file in Ada. The FORTRAN design requires the profile data file to decouple the Profile and Truth Model processing. The use of a display format file in the Ada design is motivated by reusability considerations. By keeping the detailed formats of menus and displays on an external file, the user interface is easier to reuse on a future simulator. The number of external data flows is greater in the FORTRAN design, as shown in Table 4-2. Most of the additional data flows arise from the separation of the FORTRAN design into three programs, requiring more data flows to and from the user and distinct data flows to the profile data and results output files that decouple the programs. Also, as shown in Figure 4-2, the star catalog external file is required in both Profile and the Truth Model.

The Ada design (Figure 4-3) involves the minimum number of external data flows. The details of accessing each file are confined to a single subsystem.

4.2.3 INTERNAL DATA FLOW

Table 4-2 shows that the Ada design has nine internal data flows, versus three for the FORTRAN design. Of course, no more internal data flows are possible in the FORTRAN case because Profile and the Postprocessor are separate programs. The three remaining subsystems in the FORTRAN design exchange data with one another via COMMON blocks. (Although the use of COMMON has been criticized, results from the flight dynamics environment described in Section 3.7 have shown it to be effective.)

Table 4-2. Basic Quantitative Design Characteristics

CHARACTERISTIC	FORTRAN DESIGN	ADA DESIGN
SEPARATE PROGRAMS	3	1
TASKS	5 (IN SIMULATOR PROGRAM)	5
EXTERNAL ENTITIES	9	9
EXTERNAL DATA FLOWS	18	10
INTERNAL DATA FLOWS	3	9
SUBROUTINES/SUBPROGRAMS	262	252
PACKAGES	N/A	104

0273 S7/3/11

The number of distinct data flows (connections) between subsystems is greater in Ada; however, fewer data items pass over these connections than in FORTRAN. Figures 4-2 and 4-3 are annotated with the count of data items associated with each data flow. An example will show how various Ada language features help to reduce the proliferation of data item names.

Both designs provide for the recording of simulation analysis results. In FORTRAN (Figure 4-2), these results pass from the Truth Model and OBC Model via COMMON to the Simulation Control-I/O Subsystem, which writes them to the external results output file. In Ada (Figure 4-3), the internal data flows from the Truth Model, OBC Model, and Simulation

Control carry results data to the User Interface, which writes them to the results output file.

In the FORTRAN design, the results data record comprises 43 distinct variable names. In Ada, the results are passed under a single identifier, Results_Data, when a procedure, Put_Results_Data, in the User Interface is called by the Truth Model, OBC Model, or Simulation Control. This reduction in the number of identifiers is possible because of the use of Ada's variant record feature. In the example, Results_Data can be either an executed ground command, parameter update, error message, or analysis result. In Ada, the user can declare Results_Data as type RESULT, defined as a record type with a variant part as follows:

```
type RESULT_KIND is (Error_Msg, Log_Command, Results,
                     Parameters;
type RESULT (Kind:  RESULT_KIND:=Results) is
  record
    case Kind is
      when Error_Msg | Log_Command =>
        Results_Line:  STRING (1...80);
      when Results | Parameters =>
        Result_Rec:  PARAM_RESULT;
    end case;
  end record;
```

Because of such features, the count of data items is consistently lower over the Ada data flows than over the FORTRAN data flows.

A further analysis of the internal data flow compared the average number of data sources used by modules in each design. A data source could be a parameter list, a COMMON block in FORTRAN, or a collection of state data in Ada. The OBC subsystems were examined because they are the most closely matched by functionality (Table 4-1). The results showed that a module in FORTRAN uses an average of 4.03 data sources compared to 1.95 data sources for a typical Ada subprogram. The higher FORTRAN value reflects the 37 COMMON blocks used to store data. The Ada design provides a single Attitude Control and Determination Subsystem (ACADS) data base of 222 OBC parameters. The typical Ada subprogram in the OBC Subsystem uses data from this ACADS data base along with a smaller source of state data.

4.2.4 CONTROL FLOW

A study of modules in the OBC Subsystems of each design compared the distribution of fanout (number of modules called by a given module). Ada modules showed a higher average fanout, 1.68, versus 1.34 for FORTRAN. In a comparison of average fanout squared, Ada modules were higher, 12.68, versus 6.24 for FORTRAN modules. Higher fanout in the Ada design reflects a basic pattern of obtaining data values by subprogram invocation. The FORTRAN practice is to obtain values from COMMON without a subroutine call. An example shows this difference.

The example focuses on the function of modeling the magnetometer. The modules that perform this function were identified and examined in each design. In each case, the same output quantities, magnetometer measurements, are produced. The Ada design involves invoking five subprograms in addition to the calls on the Simulation Support Subsystem to obtain magnetometer parameters. Four of the subprograms are invoked to obtain current data needed for the magnetometer modeling: spacecraft attitude, geomagnetic field, torquer dipoles, and magnetic field of the coils. The fifth subprogram call is to put the results data to the user interface.

Magnetometer modeling in the FORTRAN design requires no calls to supporting subroutines for data. Six COMMON blocks are referenced to obtain necessary data (attitude, magnetic field, etc.) and to pass output magnetometer measurements.

4.2.5 CONCLUSIONS

The comparison of FORTRAN and Ada designs revealed significant differences in both the design processes and products. In this experiment, the Ada design was shown to be different to a significant degree from the FORTRAN design. This result differs from that reported in Reference 46 for another monitored Ada development project in a different environment.

The results have implications for other organizations contemplating the use of Ada. This experiment led to a design that exploits Ada's features for expressing design abstractions. However, this result was facilitated by (1) the use of a specification method, the Composite Specification Model (Reference 5), to counteract the influence of design-laden requirements; (2) the explicit allowance for the Ada team to pursue new design methods, not requiring the team to take the less costly route of reusing the existing deign; and (3) training in alternative design methods.

4.3 DESIGN METHOD EVALUATION

Several SEL-supported design studies have been conducted at the University of Maryland using students as subjects. Two recent experiments are of particular interest because they are concerned with the differences among design methods. The full results of these experiments are not yet available (Reference 47).

In the first experiment, subjects used different design methods to develop software in Ada for either a text formatter or an electronic mail system. The design methods included Jackson System Development (JSD) (Reference 48), object-oriented design (OOD) (Reference 38), structured design (SD) (Reference 44), and several ad hoc design approaches. The objectives of the experiment were to learn whether the prescriptive design methods (JSD, OOD, SD) were correctly applied, identify structural differences among the designs, and analyze the resulting programs. As a follow-on, the programs were given to graduate students who were directed to make modifications to the programs. This follow-on study investigated a possible relationship between design method and ease of modifying the corresponding program.

The second experiment consisted of ten three-person teams: five teams using OOD and five using SD. Again the objective was to explore differences in the resulting programs that could be associated with the particular design method employed. The programs ranged in size from 2400 to 3000 source lines of Ada.

The results of these experiments are still being analyzed, but are inconclusive thus far.

SECTION 5 - SUMMARY

The preceding sections described the results to date of an extensive program of design measurement and methodology evaluation conducted by the SEL. These investigations have obtained several tangible benefits for flight dynamics (and other) software developers:

- Definition of a basic set of 19 design measures for data collection and historical reference
  - Design-unique data must be captured at design time.
  - Module-level data must be summarized to the project level for archiving.
- Evaluation of common design practices and structural characteristics, showing
  - GESS use leads to significant structural differences among projects but not to productivity or reliability changes.
  - Software reuse leads to higher productivity and reliability of the full system.
  - Module size is not an effective criterion for software design.
  - High-strength modules exhibit lower costs and fault rates.
  - Control coupling, in terms of the number of invocations from a module, should be minimized.
  - Parameter or COMMON coupling should be used as appropriate. Neither is generally better than the other.

- Development of a design complexity model that suggests the following:
  - For a given total fanout, calls should be as evenly distributed across modules as is consistent with the design problem.
  - Repetition of variables should be minimized via information hiding and horizontal data transfers.
- Adaptation of object-oriented design methods for flight dynamics use resulting in
  - A procedure for defining objects from a data flow diagram
  - A graphical notation for expressing object-oriented designs
- Assessment of alternative design methods, showing
  - Specific prerequisites must be satisfied to produce substantially different designs for the same problem.
  - No single design method is generally better than the others for all design problems.

These results will be incorporated into future versions of flight dynamics standards (e.g., References 7 and 49).

REFERENCES

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

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

3. D. N. Card, "A Software Technology Evaluation Program," Annais do XVIII Congresso Nacional de Informatica, October 1985

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

5. W. W. Agresti, "An Approach to Developing Specification Measures," Proceedings From the Ninth Annual Software Engineering Workshop, NASA/GSFC, SEL-84-004, November 1984

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

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

8. Computer Sciences Corporation, Informational Memorandum, "Design Data Processing Procedures," K. Hunter, April 1986

9. --, CSC/TM-84/6154, Definition of Design Measures for the Software Engineering Laboratory, W. J. Decker, December 1984

10. --, PCA/IM-86/017(470), "Definition of Characteristic Software Design Parameters," D. N. Card, W. W. Agresti, V. E. Church, and W. J. Decker, April 1986

11. D. M. Weiss and V. R. Basili, "Evaluating Software Development by Analysis of Changes: Some Data From the Software Engineering Laboratory," IEEE Transactions on Software Engineering, February 1985

12. V. R. Basili and R. W. Selby, "Calculation and Use of an Environment's Characteristic Software Metric Set," Proceedings of the Eighth International Conference on Software Engineering. New York: IEEE Computer Society Press, August 1985

13. M. H. Halstead, Elements of Software Science. New York: Elsevier, 1977

14. Computer Sciences Corporation, PCA/IM-85/058(455), "Classification of Software Engineering Laboratory Projects," Q. L. Jordan, D. N. Card, and W. W. Agresti, December 1985

15. --, CSC/SD-75/6057, Graphic Executive Support System User's Guide, J. E. Hoover, T. E. Board, and A. M. Montgomery, August 1975

16. --, Technical Memorandum, "Characteristics of FORTRAN Modules," D. N. Card, Q. L. Jordan, and V. E. Church, June 1984

17. D. N. Card, V. E. Church, and W. W. Agresti, "An Empirical Study of Software Design Practices," IEEE Transactions on Software Engineering, February 1986

18. J. D. Bowen, "Module Size: A Standard or Heuristic?" Journal of Systems and Software, 1984:4

19. V. R. Basili and B. T. Perricone, "Software Errors and Complexity: An Empirical Investigation," Communications of the ACM, January 1984

20. V. Y. Shen, T. Yu, S. M. Thebaut, and L. R. Paulsen, "Identifying Error-Prone Software-An Empirical Study," IEEE Transactions on Software Engineering, April 1985

21. P. C. Belford, R. C. Berg, and T. L. Hannan, "Central Flow Control Software Development: A Case Study of the Effectiveness of Software Engineering Techniques," Proceedings of the Fourth International Conference on Software Engineering. New York: IEEE Computer Society Press, September 1979

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

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

24. G. J. Myers, Composite Structured Design. New York: Van Nostrand Reinhold, 1978

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

26. T. J. Emerson, "A Discriminant Metric for Module Cohesion," Proceedings of the Seventh International Conference on Software Engineering. New York: IEEE Computer Society Press, 1984

27. D. N. Card, G. T. Page, and F. E. McGarry, "Criteria for Software Modularization," Proceedings of the Eighth International Conference on Software Engineering. New York: IEEE Computer Society Press, August 1985

28. D. N. Card, F. E. McGarry, and G. T. Page, "Evaluating Software Engineering Technologies," Proceedings From the Eighth Annual Software Engineering Workshop, NASA/GSFC, SEL-83-007, November 1983

29. E. Soloway and K. Ehrlich, "Empirical Studies of Programming Knowledge," IEEE Transactions on Software Engineering, September 1984

30. G. A. Miller, "The Magical Number Seven, Plus or Minus Two: Some Limits on our Capacity for Processing Information," Psychology Review, 1956

31. Computer Sciences Corporation, PCA/IM-86/018(470), "Measuring Software Design Complexity," D. N. Card and W. W. Agresti, June 1986

32. B. W. Kernighan and P. S. Plauger, The Elements of Programming Style. New York: McGraw Hill, 1974

33. J. B. Lohse and S. H. Zweben, "Experimental Evaluation of Software Design Principles: An Investigation Into the Effect of Module Coupling on System Modifiability," Journal of Systems and Software, 1984:4

34. D. L. Parnas, "On the Criteria To Be Used in Decomposing Systems into Modules," Communications of the ACM, December 1972

35. S. Rotenstreich and W. E. Howden, "Two-Dimensional Program Design," IEEE Transactions on Software Engineering, March 1986

36. Computer Sciences Corporation, PCA/IM-86/022(470), "Evaluation of Six Design Hypotheses", D. N. Card, June 1986

37. A. Goldberg and D. Robson, Smalltalk 80: The Language and its Implementation. New York: Addison-Wesley, 1983

38. G. Booch, Software Engineering with Ada. New York: Benjamin/Cummings, 1983

39. W. W. Agresti, V. E. Church, D. N. Card, and P. L. Lo, "Designing With Ada for Satellite Simulation: A Case Study," Proceedings of the First International Conference on Ada Applications for the Space Station, June 1986

40. W. W. Agresti, "Measuring Ada as a Software Development Technology in the Software Engineering Laboratory," Proceedings From the Tenth Annual Software Engineering Workshop, NASA/GSFC, SEL-85-006, December 1985

41. Software Engineering Laboratory, SEL-86-002, General Object-Oriented Software Development, E. Seidewitz and M. Stark, July 1986

42. E. W. Dijkstra, "The Structure of 'THE' Multiprogramming System," Communications of the ACM, December 1972

43. V. Rajlich, "Paradigms for Design and Implementation in Ada," Communications of the ACM, July 1985

44. E. Yourdon and L. L. Constantine, Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design. New York: Prentice-Hall, 1979

45. G. W. Cherry and B. S. Crawford, The Pamela Methodology, November 1985

46. V. R. Basili et al., "Characteristics of an Ada Software Development," Computer, September 1985

47. E. E. Katz, H. D. Rombach, and V. R. Basili, "Structure and Maintainability of Ada Programs: Can We Measure the Differences?" Presented at Ninth Minnowbrook Workshop on Software Performance Evaluation, August 1986

48. M. A. Jackson, System Development, Englewood Cliffs: Prentice-Hall, 1982

49. Software Engineering Laboratory, SEL-86-001, Programmer's Handbook for Flight Dynamics Software Development, R. J. Wood and E. C. Edwards, March 1986

STANDARD BIBLIOGRAPHY OF SEL LITERATURE

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

SEL-ORIGINATED DOCUMENTS

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

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

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

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

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

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

SEL-78-302, FORTRAN Static Source Code Analyzer Program (SAP) User's Guide (Revision 3), W. J. Decker and W. A. Taylor, July 1986

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

SEL-81-106, Software Engineering Laboratory (SEL) Document Library (DOCLIB) System Description and User's Guide, W. Taylor and W. J. Decker, May 1985

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

SEL-81-110, Evaluation of an Independent Verification and Validation (IV&V) Methodology for Flight Dynamics, G. Page, F. E. McGarry, and D. N. Card, June 1985

SEL-81-203, Software Engineering Laboratory (SEL) Data Base Maintenance System (DBAM) User's Guide and System Description, P. Lo, June 1984

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

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

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

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

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

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

SEL-82-102, FORTRAN Static Source Code Analyzer Program (SAP) System Description (Revision 1), W. A. Taylor and W. J. Decker, April 1985

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

SEL-82-406, Annotated Bibliography of Software Engineering Laboratory Literature, D. N. Card, Q. L. Jordan, and F. E. McGarry, November 1986

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

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

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

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

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

SEL-84-001, Manager's Handbook for Software Development, W. W. Agresti, F. E. McGarry, D. N. Card, et al., April 1984

SEL-84-002, Configuration Management and Control: Policies and Procedures, Q. L. Jordan and E. Edwards, December 1984

SEL-84-003, Investigation of Specification Measures for the Software Engineering Laboratory (SEL), W. W. Agresti, V. E. Church, and F. E. McGarry, December 1984

SEL-84-004, Proceedings From the Ninth Annual Software Engineering Workshop, November 1984

SEL-85-001, A Comparison of Software Verification Techniques, D. N. Card, R. W. Selby, Jr., F. E. McGarry, et al., April 1985

SEL-85-002, Ada Training Evaluation and Recommendations From the Gamma Ray Observatory Ada Development Team, R. Murphy and M. Stark, October 1985

SEL-85-003, Collected Software Engineering Papers: Volume III, November 1985

SEL-85-004, Evaluations of Software Technologies: Testing, CLEANROOM, and Metrics, R. W. Selby, Jr., May 1985

SEL-85-005, Software Verification and Testing, D. N. Card, C. Antle, and E. Edwards, December 1985

SEL-85-006, Proceedings From the Tenth Annual Software Engineering Workshop, December 1985

SEL-86-001, Programmer's Handbook for Flight Dynamics Software Development, R. Wood and E. Edwards, March 1986

SEL-86-002, General Object-Oriented Software Development, E. Seidewitz and M. Stark, August 1986

SEL-86-003, Flight Dynamics System Software Development Environment Tutorial, J. Buell and P. Myers, July 1986

SEL-86-004, Collected Software Engineering Papers: Volume IV, November 1986

SEL-86-005, Measuring Software Design, D. N. Card, October 1986

SEL-RELATED LITERATURE

Agresti, W. W., Definition of Specification Measures for the Software Engineering Laboratory, Computer Sciences Corporation, CSC/TM-84/6085, June 1984

[4]Agresti, W. W., V. E. Church, D. N. Card, and P. L. Lo, "Designing With Ada for Satellite Simulation: A Case Study," Proceedings of the First International Symposium on Ada for the NASA Space Station, June 1986

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

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

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

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

[3]Basili, V. R., "Quantitative Evaluation of Software Methodology," Proceedings of the First Pan-Pacific Computer Conference, September 1985

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

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

[3]Basili, V. R., and N. M. Panlilio-Yap, "Finding Relationships Between Effort and Other Variables in the SEL," Proceedings of the International Computer Software and Applications Conference, October 1985

[4]Basili, V. R., and D. Patnaik, A Study on Fault Prediction and Reliability Assessment in the SEL Environment, University of Maryland, Technical Report TR-1699, August 1986

[2]Basili, V. R., and B. T. Perricone, "Software Errors and Complexity: An Empirical Investigation," Communications of the ACM, January 1984, vol. 27, no. 1

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

[3]Basili, V. R., and C. L. Ramsey, "ARROWSMITH-P--A Prototype Expert System for Software Engineering Management," Proceedings of the IEEE/MITRE Expert Systems in Government Symposium, October 1985

Basili, V. R., and R. Reiter, "Evaluating Automatable Measures for Software Development," Proceedings of the Workshop on Quantitative Software Models for Reliability, Complexity, and Cost. New York: IEEE Computer Society Press, 1979

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

[3]Basili, V. R., and R. W. Selby, Jr., "Calculation and Use of an Environments's Characteristic Software Metric Set," Proceedings of the Eighth International Conference on Software Engineering. New York: IEEE Computer Society Press, 1985

Basili, V. R., and R. W. Selby, Jr., Comparing the Effectiveness of Software Testing Strategies, University of Maryland, Technical Report TR-1501, May 1985

[4]Basili, V. R., R. W. Selby, Jr., and D. H. Hutchens, "Experimentation in Software Engineering," IEEE Transactions on Software Engineering, July 1986

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

[3]Basili, V. R., and D. M. Weiss, "A Methodology for Collecting Valid Software Engineering Data," IEEE Transactions on Software Engineering, November 1984

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

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

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

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

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

[3]Card, D. N., "A Software Technology Evaluation Program," Annais do XVIII Congresso Nacional de Informatica, October 1985

[4]Card, D. N., V. E. Church, and W. W. Agresti, "An Empirical Study of Software Design Practices," IEEE Transactions on Software Engineering, February 1986

[3]Card, D. N., G. T. Page, and F. E. McGarry, "Criteria for Software Modularization," Proceedings of the Eighth International Conference on Software Engineering. New York: IEEE Computer Society Press, 1985

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

[4]Church, V. E., D. N. Card, W. W. Agresti, and Q. L. Jordan, "An Approach for Assessing Software Prototypes," ACM Software Engineering Notes, July 1986

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

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

[3]McGarry, F. E., J. Valett, and D. Hall, "Measuring the Impact of Computer Resource Quality on the Software Development Process and Product," Proceedings of the Hawaiian International Conference on System Sciences, January 1985

[3]Page, G., F. E. McGarry, and D. N. Card, "A Practical Experience With Independent Verification and Validation," Proceedings of the Eighth International Computer Software and Applications Conference, November 1984

[3]Ramsey, J., and V. R. Basili, "Analyzing the Test Process Using Structural Coverage," Proceedings of the Eighth International Conference on Software Engineering. New York: IEEE Computer Society Press, 1985

[4]Seidewitz, E., and M. Stark, "Towards a General Object-Oriented Software Development Methodology," Proceedings of the First International Symposium on Ada for the NASA Space Station, June 1986

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

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

[3]Weiss, D. M., and V. R. Basili, "Evaluating Software Development by Analysis of Changes: Some Data From the Software Engineering Laboratory," IEEE Transactions on Software Engineering, February 1985

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

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

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

NOTES:

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

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

[3]This article also appears in SEL-85-003, Collected Software Engineering Papers: Volume III, November 1985.

[4]This article also appears in SEL-86-004, Collected Software Engineering Papers: Volume IV, November 1986.