

N82-24010

SEL-81-013

PROCEEDINGS OF THE SIXTH ANNUAL SOFTWARE ENGINEERING WORKSHOP

ORGANIZED BY:
SOFTWARE ENGINEERING LABORATORY
GSFC

DECEMBER 2, 1981



GODDARD SPACE FLIGHT CENTER
GREENBELT, MARYLAND



National Aeronautics and
Space Administration

Goddard Space Flight Center
Greenbelt, Maryland 20771

PROCEEDINGS
OF
SIXTH ANNUAL SOFTWARE ENGINEERING WORKSHOP

Organized by:
Software Engineering Laboratory
GSFC

December 2, 1981

GODDARD SPACE FLIGHT CENTER
Greenbelt, Maryland

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.

Single copies of this document can be obtained by writing to

Frank E. McGarry

Code 582.1

NASA/GSFC

Greenbelt, Maryland 20771

Page Intentionally Left Blank

SIXTH ANNUAL SOFTWARE ENGINEERING WORKSHOP

ABOUT THE WORKSHOP

The Sixth Annual Software Engineering Workshop was held on December 2, 1981, at Goddard Space Flight Center in Greenbelt, MD. Nearly 200 people, representing 6 universities, 19 agencies of the federal government, and 30 private organizations, attended the meeting.

As in the past 5 years, the major emphasis for this meeting was the reporting and discussion of experiences in the identification, utilization, and evaluation of software methodologies, models, and tools. Eleven speakers, making up four separate sessions, participated in the meeting with each session having a panel format with heavy participation from the audience.

The workshop is organized by the Software Engineering Laboratory (SEL), whose members represent the NASA/GSFC, University of Maryland, and Computer Sciences Corporation (CSC). The meeting has been an annual event for the past 6 years (1976 to 1981), and there are plans to continue those yearly meetings as long as they are productive.

The record of the meeting is generated by members of the SEL and is printed and distributed by the Goddard Space Flight Center. All persons who are registered on the mail list of the SEL receive copies of the proceedings at no charge.

Additional information about the workshop or about the SEL may be obtained by contacting:

Mr. Frank McGarry
Code 582.1
NASA/GSFC
Greenbelt, MD 20771

301-344-5048

Page Intentionally Left Blank

AGENDA

SIXTH ANNUAL SOFTWARE ENGINEERING WORKSHOP
NASA/GODDARD SPACE FLIGHT CENTER
BUILDING 3 AUDITORIUM
DECEMBER 2, 1981

8:45 a.m.	INTRODUCTORY REMARKS	F. E. McGarry/GSFC
	MORNING CHAIRMAN	F. E. McGarry
9:00 a.m.	SESSION NO. 1	"Evaluating Software Development Characteristics"
	D. Weiss (NRL)	"Analyzing Error Characteristics in Software Development"
	J. Page (CSC)	"Evaluating the Effects of an Independent Verification and Validation Team"
	V. Basili (University of MD)	"Assessment of Software Measures in the Software Engineering Laboratory"
10:30 a.m.	BREAK	
10:45 a.m.	SESSION NO. 2	"Software Metrics"
	J. Gaffney/R. Judge (IBM)	"The Quantitative Impact of Four Factors on Work Rates Experienced During Software Development"
	J. Post (Boeing Aerospace)	"Software Quality Metrics for Distributed Systems"
	D. Card (CSC)	"Identification and Evaluation of Software Metrics"
12:45 p.m.	LUNCH	
	AFTERNOON CHAIRMAN	V. Basili
1:15 p.m.	SESSION NO. 3	"Software Models"
	B. Littlewood/A. Sofer (GW University)	"A Bayesian Approach to Parameter Estimation in the Jelinski-Moranda Software Reliability Model"

	H. Sayani/C. Svoboda (ASTEC)	“The Problem of Resonance in Technology Usage”
2:45 p.m.	BREAK	
3:00 p.m.	SESSION NO. 4	“Software Methodologies”
	H. Mills/M. Dyer (IBM)	“A Methodology for Improving Software Reliability”
	B. Jones (Hughes)	“Selecting a Software Development Methodology”
	R. Hamilton (Bell Labs)	“Development Techniques for Generic Software”
5:00 p.m.	ADJOURN	

Workshop Introduction

The software engineering workshop is one attempt to promote the interchange of ideas, experiences and approaches to the measurement and evaluation of varying techniques used in the software development process. The first meeting was held in August of 1976 in partial response to NASA's concern for the apparent gap between the availability of state-of-the-art software development approaches and the actual utilization of these techniques. Also, the First International Conference on Software Engineering had been held in Washington, DC the previous year and had stimulated interest and concern within the NASA community.

The first workshop at Goddard essentially surveyed some available state-of-the-art development techniques to determine if they would be applicable in the NASA environment. The meeting was attended by approximately 25 people. As a result of this first workshop, NASA/GSFC initiated efforts to investigate the effectiveness of the numerous available approaches to developing software.

Within a few months after the first workshop, an organization was created (called the Software Engineering Laboratory--SEL) which was chartered to measure the impact that various methodologies, tools, and models had on applications software within NASA/GSFC. The SEL was formed as a partnership between NASA/GSFC, the University of Maryland, and Computer Sciences Corporation (CSC). During the first year of operation, the SEL concerned itself with the approaches to conducting software development experiments and to collecting development data for study. The SEL became very interested in finding others who were attempting to do similar things.

The Second Software Engineering Workshop was held in September 1977 at NASA/GSFC with the central theme being 'Who else is performing software experiments and collecting software data'. Approximately 55 persons attended this meeting and many approaches and experiences relating to software experiments and data collection were discussed--both during presentations and during informal discussions.

The third meeting was held in September of 1978 at NASA/GSFC. Continued emphasis was placed on the data collection and software experiments. Many of the discussions focused on the question of 'how' do you collect software data and how do you successfully conduct software experiments. This meeting was attended by approximately 70 people.

The fourth and fifth meetings again were held at NASA/GSFC in November of 1979 and November of 1980 respectively. During these sessions, the emphasis was once again placed on data collection and the actual experiences with software methodologies, models, tools, and measures.

The sixth meeting is another attempt to listen to experiences that people have had in attempting to apply various modern programming practices. Although the workshops occasionally seem to stray away from the central theme of data collections and software experiments, the major objectives are still essentially being met. As an example, these workshops have been instrumental in providing suggestions and guidance to the efforts within the SEL at Goddard. The SEL has now been in existence for about 6 years and has closely monitored 34 applications projects with NASA/GSFC, collecting approximately 15 m bytes of development data.

This data has continually been studied and evaluated and has led to numerous measurements and evaluations of software methodology models and tools.

Many effective relationships were initiated through the workshops and a great number of experiences, experimental results and data itself has been exchanged between organizations. The Sixth Workshop will attempt to stimulate further exchanges.

SIXTH ANNUAL

SOFTWARE ENGINEERING WORKSHOP

DEC. 2, 1981

NASA/GSFC

Page Intentionally Left Blank

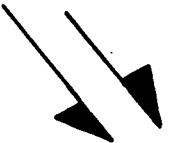
WORKSHOP BACKGROUND

1ST (AUGUST 1976) STIMULATED BY

- 1ST INTERNATIONAL CONFERENCE ON S.E. (1975)
- NASA CONCERN FOR SOFTWARE TECHNOLOGY
- APPARENT LACK OF TECHNOLOGY UTILIZATION



CREATION OF SOFTWARE ENGINEERING LABORATORY (SEL)

- 
- 2ND (SEPT. 1977) • WHO IS COLLECTING SOFTWARE DATA
• WHO IS EXPERIMENTING WITH TECHNOLOGY
- 3RD (SEPT. 1978) • HOW DO YOU VALIDATE DEVELOPMENT DATA
• HOW DO YOU INTERPRET THE DATA
- 4TH (NOV. 1979) • SOME RESULTS OF EXPERIMENTS
SOFTWARE MODELS
SOFTWARE METRICS
- 5TH (NOV. 1980) • FURTHER EXPERIMENTS, DATA COLLECTION
& PROPOSED EXPERIMENTS

Page intentionally left blank

SOFTWARE ENGINEERING LABORATORY

- CREATED FALL 1976 (NASA/GSFC-UNIV. MD.)
- WHY
 - PROFILE OF CURRENT DEVELOPMENT TECHNIQUES
 - EVALUATE EFFECTIVENESS OF MPP
 - APPLY IMPROVED METHODS TO SOFTWARE AT GSFC
- HOW TO PROCEED
 - EXTRACT DETAILED DATA FROM ACTIVE TASKS.....(FORMS/DATA
COLLECTION/VALIDITY)
 - GENERATE CONTROL EXPERIMENTS.....(EXPERIMENTAL DESIGN/
STATISTICAL ANALYSIS)
 - QUALIFY THE 'GOOD' SOFTWARE AND 'BAD' (MODELS/MEASURES/
METRICS)

Page Intentionally Left Blank

MEASURING SOFTWARE IN THE SEL BASIS FOR ANALYSIS

- LABORATORY EXPERIMENTS34 PROJECTS
- INFORMATION MONITORED 1.6 million L.O.C.
- PROGRAMMERS/MANAGERS REPRESENTED.....115 PEOPLE
- DATA EXTRACTED.....40 m BYTES ON DATA BASE
(15,000 FORMS)
FORMS
TOOLS
SUBJECTIVE
- METHODOLOGIES APPLIED 200 QUALIFYING PARAMETERS
VARIOUS MODELS,
TOOLS

NASA/SEL

Page Intentionally Left Blank

SOFTWARE ENGINEERING LABORATORY

CURRENT ACTIVITIES (FY 81)

PROJECTS BEING MONITORED

APPROACHES UNDER STUDY

<u>NAME</u>	<u>END OF DATE</u>	<u>APPROXIMATE SIZE (L.O.C.)</u>
DE-A (ADS)	6/81	68,000
DE-B	6/81	65,000
DADS	5/81	16,000
AODS	10/81	18,000
RADMAS	6/82	50,000
AADS	9/82	15,000
DECAP	6/81	12,000
GEDAP	7/81	4,000

- INDEPENDENT VERIFICATION & INTEGRATION
- CONFIGURATION MANAGEMENT TOOL
- REQUIREMENTS LANGUAGE (MEDL—R)
- INFORMATION HIDING
- DATA ABSTRACTION
- STRUCTURED ANALYSIS (YOURDON & DEMARCO)
- N² CHARTS FOR DESIGN

ANALYSIS ACTIVITIES:

- RELIABILITY MODEL EVALUATION (MUSA, GOEL, . . .)
- APPROACHES TO SOFTWARE TESTING
- MEASURES — METRICS FOR SOFTWARE (MCCA BE, HALSTEAD, MCCALL, . . .)
- TOOLS EVALUATION (PWB, MEDL-R, CAT, SAP, . . .)
- METHODOLOGY EVALUATION

Page Intentionally Left Blank

TOPICS FOR 6TH WORKSHOP

- ANALYZING ERROR CHARACTERISTICS
- MEASURES FOR SOFTWARE
- MODELS FOR RELIABILITY & DEVELOPMENT
- METHODOLOGIES FOR DEVELOPMENT

SUMMARY OF THE SESSIONS:
SIXTH ANNUAL SOFTWARE ENGINEERING WORKSHOP

Suellen Eslinger
COMPUTER SCIENCES CORPORATION .

and

THE GODDARD SPACE FLIGHT CENTER
SOFTWARE ENGINEERING LABORATORY

Prepared for the
NASA/GSFC

Sixth Annual Software Engineering Workshop

SESSION 1 - EVALUATING SOFTWARE DEVELOPMENT CHARACTERISTICS

Dave Weiss - "Analyzing Error Characteristics in Software Development"

The first speaker of the first session was Dave Weiss from the Naval Research Laboratory (NRL). The purpose of his presentation was to characterize software changes in two different software development environments. Changes required to correct errors formed one subcategory of the software changes studied. Data was used from several projects at GSFC and at NRL; data for the GSFC projects was collected by the Software Engineering Laboratory (SEL).

Although the two environments were quite different, the characteristics of the software changes were found to be very similar. For example, in both environments relatively few errors (approximately 5 percent) took more than 1 day to correct, and relatively few errors (approximately 2 to 5 percent) were caused by requirements problems. Although the error characteristics detected may not be applicable to other environments, the same type of study could be performed by another group of software developers to characterize errors in their environment. The results of this type of study can help determine where effort should be focused to reduce errors and thus improve reliability in software being developed in a given environment.

In response to questions from the audience, Weiss clarified several points:

- Interface errors were only a small part of the errors counted that affected more than one module. Unlike similar studies in the literature, relatively few errors in the two environments were found to be interface errors.

- All projects studied were completed, but no data was used from the maintenance phase of the projects.
- Changes were tracked from the time that a module was entered into the library. In both environments this process took place after the programmer had coded, compiled, and tested the module, i.e., at the completion of unit testing.
- Neither environment had a formal configuration control board. The programmer was responsible for determining the correctness of the change, and the effort to fix an error was accepted to be the amount of time the programmer said it took to make and test the change.
- The NRL environment had even less configuration control than the GSFC environment. Configuration control in the NRL project consisted of project leaders alone performing library updates.

Jerry Page - "Evaluating the Effects of an Independent Verification and Validation Team"

The next speaker of the session was Jerry Page from Computer Sciences Corporation (CSC). The purpose of his presentation was to evaluate the effectiveness of a particular methodology when utilized in the development of application software. Experiments in applying independent verification and integration (V&I) were conducted at GSFC during the development of two ground-based software projects. CSC was responsible for the V&I effort under contract to GSFC. Detailed data for the projects was collected by the SEL.

The two V&I projects were compared to two similar earlier projects monitored by the SEL for which V&I had not been used. Seven specific measures were used to weigh the effects of applying the methodology. The only clearly

favorable effect found was a reduction in the number of requirements errors. Furthermore, the V&I experimental projects were costly, and the resulting software seemed to be as error prone as the software produced by the projects for which V&I was not used. However, the speaker noted that as more experience is gained with a particular methodology, better results are usually achieved. Thus, Page indicated, more experimentation with V&I is warranted, especially with projects of a larger size (10 to 12 staff-years) and/or with high reliability requirements.

This presentation generated a large response from workshop participants. The following points were clarified by Page in answer to questions from the audience:

- The V&I teams represented approximately 15 to 18 percent of the development effort in size and were similar to the development teams in experience.

- In general, the V&I teams worked behind the development teams, verifying the completed code while new code was being developed.

- The activity of code reading was performed by the development teams as a standard practice. Since the V&I teams were relatively small compared to the amount of code produced, the V&I teams emphasized testing of the software and not code reading. In fact, testing was found to be the most cost-effective part of the V&I effort.

- No investigation was made of the effect of the V&I teams on the readability or the maintainability of the code. Since the V&I teams were not directly involved in the code reading activity, their presence was not expected to affect the quality of the code in readability or maintainability.

- In the four projects studied, similar methodologies were used, except for the presence of the V&I teams.

- In all four projects, acceptance testing was performed by an independent team, whose effort did not overlap the effort of either the development teams or the V&I teams. In particular, the V&I teams did not verify the acceptance tests. Thus, the quality of acceptance tests was not perceived to differ significantly for the four projects.

- Most errors found during acceptance testing were not due, in general, to testing with real data. Since real data is not usually obtained until very late in acceptance testing, most testing is performed with simulated data.

- A member of the audience suggested that the value of the V&I efforts may appear after acceptance testing. Page responded that in this environment, on the average, only 15 percent of the total cost is incurred during the maintenance phase. Thus, a significant savings in cost is not expected for the V&I projects during this phase. However, all of the projects studied are still being monitored, and the data will continue to be analyzed.

- There were some instances in which the development teams relied upon the V&I teams to find their errors.

- There was also an overlap in errors found by the development teams and the V&I teams although the percentages have not been computed.

- CSC's Milt Phenneger, who participated in the V&I effort, suggested that the V&I process could be improved by tailoring the design and scheduling of the software releases to an independent testing effort. However, the speaker noted that the purpose of the experiment was to assess the effect of independent V&I without perturbing the existing software development process.

Vic Basili - "Assessment of Software Measures in the Software Engineering Laboratory"

The last speaker of the session was Vic Basili from the University of Maryland. This presentation concentrated on software measures as studied in the SEL. He outlined the characteristics of measures examined by the SEL during the past 4 years. His discussion focused on various classes of measures, such as subjective and objective measures of the software process and product, cost, and quality. He discussed the use of metrics for categorization, evaluation, and prediction. One result obtained from the analysis of SEL data is that many of the complexity measures, including the Halstead measures, are highly correlated with each other and with the number of lines of code. This is a disappointing result because it indicates that in this environment none of the more sophisticated complexity measures is a better predictor than the simple measure of lines of code.

A cost model has been developed using subjective metrics to modify the basic size/effort equation. Other results indicate that in this environment productivity correlates positively with methodology but with few other factors, including size. Also, subjective measures of quality correlate positively with methodology and inversely with complexity.

In response to questions from the audience, Basili clarified the following points:

- Examples were given of the subjective measures of quality, of the methodology measures, and of the complexity measures for which data is being collected by the SEL.
- On a typical project studied by the SEL, methodologies either tend to be used as a total group or completely avoided. As methodology is used to a larger extent, the quality and productivity tend to increase. However, the

measures dealing with the degrees of use of a particular methodology do not function individually as predictors. Rather, the overall set of methodology measures should be used.

SESSION 2 - SOFTWARE METRICS

Bob Judge - "The Quantitative Impact of Four Factors on Work Rates Experienced During Software Development"

The first speaker of the second session was Bob Judge from the International Business Machines Corporation (IBM), who presented the results of a study done jointly with John Gaffney. The purpose of the study was to attempt to use parameters (or factors) to explain the effort required for developing software with the end goal of building a cost estimation model.

The effects of four factors on work rate were measured for nine components of the software development life cycle. The four general factors studied were the personnel type (programmers versus systems engineers), the product (type of software application), the computer (one of three host computers), and the code type (new versus modified software). Data was used from projects developed within IBM. The estimation process was more effective for some components of the life cycle than for others. The four factors provided the best estimates of work rate for the components dealing with implementation and the worst estimates of work rate for the requirements analysis phase. Overall, 39 percent of the variation in work rate for the projects studied was explained.

In response to questions from the audience, Judge clarified the following points:

- The study was based on historical data for completed projects.
- The number of samples used for the analysis was the number of projects studied. However, not every project necessarily covered all nine components of the software life cycle.

- The cost data used came directly from customer charges and was, therefore, considered highly accurate. Inaccuracies, however, could be present in the distribution of costs among the nine life cycle components. Dimensions of cost were expressed in staff-months instead of dollars to eliminate the effects of inflation. The size data used could contain some inaccuracies but, on the whole, it was felt to be fairly accurate.

- The purpose of the study was to obtain a predictive model for cost estimation.

Jonathan Post - "Software Quality Metrics for Distributed Systems"

The second speaker for the session was Jonathan Post from Boeing Aerospace Corporation, who discussed measures for distributed processing systems. As part of a project to define and evaluate measures for distributed systems, personnel investigated the similarities and differences between measures applicable to distributed systems and those applicable to single-processor systems.

The starting point for the study was the set of factors or qualities desirable in a software system and the criteria for evaluating those factors as defined by J. McCall from the General Electric Company. Post added criteria applicable to distributed systems to some of McCall's factors, and he defined additional factors and associated criteria for distributed systems. The rationale for these additions was presented in some detail. Post indicated that during the next year data will be collected for distributed systems developed by Boeing Aerospace; it will then be analyzed in an attempt to evaluate the quality measures that have been defined.

In response to questions from the audience, Post clarified the following points:

- A definition of a distributed system is critical to the project to select projects for which data will be collected. Since no consensus currently exists in the community for the exact definition of a distributed system, significant effort was expended on establishing what this project considered to be a distributed system.

- The data will be collected using McCall's approach of a standard worksheet filled out by project personnel. Information will be extracted from these forms by a single person in an effort to eliminate the potential for bias in the responses. Interviews will also be held with project personnel to establish the validity of the data. Since Post is familiar with practices used in the projects being studied, he expected that his role in the company as a quality assurance monitor would help him obtain valid data.

- The set of quality metrics established includes some system metrics and some software metrics. Some of the distributed system factors are the same as those established by McCall. Other factors have been modified (i.e., new criteria added to those given by McCall), while still others are entirely new.

Dave Card - "Identification and Evaluation of Software Metrics"

The last speaker of the session was Dave Card of CSC. The purpose of his presentation was to describe a procedure for identifying the underlying qualities measured by a set of software measures. For a number of actual software projects, values have been determined by the SEL for 200 measures that cover the range of GSFC software development activities.

For this study, data was used from 22 projects for 60 measures describing the software development process and product. The product measures studied included size and resource measures, and the process measures were ratings of the degree of use of various methodologies, tools, and documentation procedures. Six of these measures, for which there were insufficient examples of use in the data, were rejected by a test of normality. A factor analysis was performed on the remaining 54 measures that extracted 5 factors accounting for 77 percent of the variance of the original data. The factors can be thought of as the underlying independent qualities being measured by the 54 measures. The five factors represented methodology intensity, project size, computer usage, quality assurance, and change rate. Card emphasized that this procedure produces a descriptive model, not a predictive model, and that it is an intermediate step toward further research.

This presentation generated considerable audience interest. In response to questions, Card briefly described the factor analysis procedure and clarified the meanings of several factors. He also expanded upon the following points:

- The factors themselves are not directly measurable. The factor analysis procedure, however, computes the correlation of the original variables (i.e., measures) with each of the factors. The measures shown as contributing to each factor were those whose correlations with the factor were at the 0.01 level of significance.
- Variance can be viewed as the amount of information contained in the data. Thus, the factor model produced accounted for 77 percent of the information in the 54 measures over the 22 projects.
- The 200 measures for which data is collected by the SEL were originally selected as completely characterizing

the GSFC software development activity. The 60 measures used in this particular study consisted of all those related to the software development process or product. Of these, 54 passed the test of normality and were used in the factor analysis.

- The measures reflecting the degree of use of a particular methodology, tool, or documentation procedure are not binary variables but are ratings on a scale of 0 to 5. These ratings, reflecting the degree of use of each procedure, were assigned to each project by a single group of people.

- The factor procedure does not produce a predictive model. It provides information different from the correlations among variables. For instance, although the productivity measure was not significantly correlated with the methodology intensity factor, it can not be implied or inferred that productivity is independent of any specific methodology. In fact, the productivity measure may be highly correlated with the degree of use of an individual methodology.

- The approach followed in this study is different from that generally followed. Usually, studies select desirable qualities and then seek measures of these qualities. Here, data from a number of measures is collected, and the qualities being measured by this data are then identified.

- Several people besides the speaker pointed out that these results reflect the environment being studied by the SEL and that they may not be applicable to other environments.

SESSION 3 - SOFTWARE MODELS

Ariela Sofer - "A Bayesian Approach to Parameter Estimation in the Jelinski-Moranda Software Reliability Model"

The first speaker of the third session was Ariela Sofer from the George Washington University, who presented the results of work done jointly with Bev Littlewood. The purpose of the presentation was to evaluate the effectiveness of the Jelinski-Moranda software reliability model.

Error data provided by John Musa from Bell Laboratories was used to perform the evaluation. Estimates produced by the Littlewood model from this data were shown to be better than similar estimates obtained from the Jelinski-Moranda model. Several shortcomings in the Jelinski-Moranda model were enumerated. In particular, the estimates obtained from this model were consistently too optimistic. A Bayesian reparameterization of the Jelinski-Moranda model was presented; and estimates produced by the standard and reparameterized versions of the Jelinski-Moranda models for the error data were compared. This comparison showed that the reparameterized Jelinski-Moranda model produced better results than the standard version.

In response to questions from the audience, Sofer clarified the following points:

- In the error data used, the times between failure were calculated as the execution times between program failure. John Musa, who collected the data, further explained that a program failure was considered to be any occasion on which the program did not perform according to its requirements.

- The models being evaluated assume that the times between failures are independent. This may not be the case with actual data.

- The models assume that when a program failure occurs, the error is corrected before execution of the program continues.

Disagreement on the approach presented in Sofer's talk was evidenced by comments from John Musa and Nozer Singpurwalla. Musa stated that it was unfortunate that Littlewood was not present at the workshop to participate. Certain other points were made as follows:

- Musa stated that he had published a comparable re-parameterization of the Jelinski-Moranda model in 1975.

- Both Musa and Singpurwalla pointed out that there are problems with using quantile-quantile (Q-Q) plots to evaluate the models. Q-Q plots are based on an assumed distribution of the random variable being studied. Thus, they are sensitive to the choice of this distribution for which no clear criteria are available.

- Furthermore, Singpurwalla noted that if a uniform prior distribution were assumed, the Bayesian model should have given the same result as the original Jelinski-Moranda model. The fact that it did not suggests an error in the calculations.

- Musa said that the flaws in this approach to comparing reliability models were pointed out to him by Amrit Goel. Musa relayed this information to Littlewood but has not yet received a response from him.

Hasan Sayani - "The Problem of Resonance in Technology Usage"

The second speaker of this session was Hasan Sayani from ASTEC Corporation, who presented the results of work done

jointly with Cyril Svoboda. His presentation focused on the management considerations of introducing tools into any software development environment.

The discussion was based on observations made while consulting in this field with a number of companies. The importance of having an appropriate tool environment in developing software was brought out; and the problems involved in the implementation of such an environment were discussed from both the user and managerial point of view. In particular, Sayani identified specific recommendations (both dos and don'ts) to guide the process of adopting tools. The central theme of his presentation was the need for a systems approach to the management of software technology.

This presentation generated considerable audience interest. The chairman of the afternoon sessions, Vic Basili, remarked that Sayani had presented a comprehensive list with which he agreed. The speaker clarified the following points in the ensuing discussion:

- The tools whose implementations were studied included PSL/PSA, data base design tools, process design tools, and librarian systems.
- Members of the audience remarked that the study appeared to be applicable to the implementation of other technologies in addition to tools. Sayani agreed and stated that the approach might also be applied to introducing technology to developing nations.
- Users generally agree that tools are oversold. This situation creates management problems.
- Methodologies and tools tend to be sold to people with weak systems backgrounds who do not understand how the new technologies interact with the total software development life cycle.

- The training and maintenance of a toolsmith group is an important part of the tool implementation process to avoid the problem of tools falling into disuse when key people leave the environment.

- Companies should also standardize and institutionalize these tools to enforce their use.

- A member of the audience remarked that Japanese management techniques might be applicable to this topic. Sayani responded that certain of their techniques would be pertinent but others would not because of cultural differences. However, the Japanese have adopted the use of certain technologies that were developed here but are not as widely used in this country. For example, there are a large number of PSL/PSA users in Japan.

SESSION 4 - SOFTWARE METHODOLOGIES

Mike Dyer - "The Clean Room Software Development Process"

The first speaker of the fourth session was Mike Dyer from IBM, who presented the results of work done jointly with Harlan Mills. The purpose of the presentation was to describe the mechanics of the "clean room" software development process. Pilot projects for this approach are still being set up.

After the preparation of a structured specification, the software development process is divided between two groups of people: design engineers and product engineers. The design engineers will design and code the software product with the goal of producing first-time correct code. No use of the computer will be made by the design engineers in accomplishing this goal; instead, extensive inspections and reviews will be conducted. The product engineers will perform operational testing on the code produced by the design engineers with the goal of testing for the customer environment. Tests will be selected randomly from a set of tests developed by the product engineers from the structured specification, and errors identified by the product engineers will be returned to the design engineers for correction. This software development process purposely omits the usual step of unit testing.

Dyer stated that, based upon small experiments already conducted, there is evidence that this process works. More extensive experiments are now being planned in which data will be collected to evaluate the effect of this approach on the reliability of the software produced.

The audience reaction generated by this presentation was the largest of the entire workshop. Harlan Mills joined Mike Dyer in responding to the questions from the audience.

The following points were brought out in the ensuing discussion:

- Design engineers will be experienced in software design and coding; product engineers will be experienced in system integration and testing. Dyer and Mills indicated that IBM currently has on its staff skilled people who can perform, or can be trained to perform, in this new environment.
- The product engineers are not considered quality assurance personnel. They must perform the analysis necessary to produce the data base of test cases from the structured specification. They must also run the tests and analyze the results. To function properly the product engineers must have a thorough knowledge of the customer's operational environment.
- The product engineers will participate in drawing up the structured specification. They will reenter the software life cycle after the code is developed. They will not be allowed access to design materials during the testing phase.
- Good specifications are necessary for this approach to be successful. The entire process is based on the use of a structured specification methodology.
- This approach to software development is not primarily aimed at cost savings. The question of whether or not the "clean room" process will yield productivity gains has not been addressed. The expected benefit is in the increased reliability of the software produced. However, the testing phase in the "clean room" process is not expected to cost any more than is currently spent in the usual unit, functional, and acceptance testing phases.

- This process is also not expected to help in sizing software systems.

- Mills and Dyer clarified an earlier point by saying that test data will not be chosen at random. Rather, random tests will be selected from a data base of test cases that are designed to test all capabilities set forth by the structured specification. There will be errors that are not found by the random selection of tests, but evidence is available that random testing is as good as any other form of testing. In fact, since in sampling theory the sample size, and not the population size, is critical, Mills believes that a random sample of tests can provide better testing coverage than conventional testing.

- Evidence also exists that successful system testing can be performed without unit testing.

- Mills indicated that they do not expect to attain perfection but that they do expect to achieve an increase in reliability.

- No plans have been made to seed code with errors to assess the efficiency and effectiveness of the product engineers.

- A member of the audience observed that this process appears to push error detection farther into the software life cycle. Dyer responded that this is not the case. More errors are expected to be found by the design engineers through the review process. Moreover, since the product engineers will be performing operational testing, they are expected to find errors that normally would not be uncovered until the software was operational.

- To evaluate this process, a complete history of errors must be maintained.

- Several members of the audience questioned the use of mean time between failures (MTBF) as a measure of software reliability. Mills and Dyer indicated that they believed MTBF to be a reasonable measure and one that was familiar to management and demanded by customers. Vic Basili indicated that MTBF is a measure that is associated with other measures of software quality. Another member of the audience suggested the use of mean time to repair (MTTR).

- Mills emphasized that the "clean room" software development process would require some modification in programmer behavior. Since it is known that programmers can write thousands of lines of correct code, the goal of producing first-time correct code is not unreasonable. Programmers must be made to believe that they can do this without the use of the computer. Mills and Dyer hope to achieve this behavior modification by not allowing the programmers to have access to the compilers.

- Mills also stated that product engineering was devised because they felt that testing is a critical part of the development process. This process does not remove the ability to test the software; rather, design engineers are asked to test by thinking instead of making computer runs.

- No projects using this approach are yet complete. The pilot projects are still in the process of being set up.

- The approach is expected to work for any type of software application.

- Vic Basili indicated that in recent testing experiments he has run, the functional tests uncovered most of the errors. However, the testers did not always recognize that the test results had indicated errors.

Bob Jones - "Selecting a Software Development Methodology"

The second speaker of the session was Bob Jones from Hughes Aircraft, who discussed an approach for selecting a software methodology. The presentation centered on a Hughes contract with the U.S. Air Force to define a set of tools and methodologies to be used for integrated digital flight control software development. In response to this specific need of the Air Force, Hughes surveyed the environment and attempted to take a logical approach to the selection of tools and methodologies for that environment. The results of the study have been presented in a guidebook, a document of considerable size. Jones indicated that Hughes has started to collect data to evaluate the cost benefits of using the techniques specified by the guidebook.

In response to questions from the audience, Jones clarified several points:

- The tools and methodologies recommended included the use of CADSAT, structured design, high-order languages, and modern programming languages.
- The software produced will not be verified in flight. There is a standard procedure for verifying flight control software that uses simulated data. It is not planned to use the software produced by this experiment in flight but only to verify that it performs according to specification.
- Hughes will be collecting only cost data for this experiment. In evaluating cost-benefit tradeoffs, the benefits obtained by following the guidebook will be determined by the customer.
- A member of the audience pointed out that if the guidebook covered all the tools and methodologies mentioned, it would constitute a 4-year curriculum. Jones agreed but

stated that the guidebook did not present detailed instructions in the technologies.

Richard Hamilton - "Development Techniques for Generic Software"

The last speaker of the session was Richard Hamilton from Bell Laboratories, who spoke about a methodology for developing generic software. His discussion centered on one class of application: networking with a specific protocol. The use of a layered approach and a finite state machine in implementing the X.25 protocol was presented. The complexity, size, and speed of the newly developed generic program were compared to an older, machine-specific X.25 protocol program. Hamilton indicated that the complexity of the two programs was about the same. However, the size of the generic program was larger and its speed was faster.

In response to questions from the audience, Hamilton clarified the following points:

- The complexity measure used was the McCabe measure that provides a measure of the number of branches in the program.
- Hamilton indicated that the finite state machine used in the generic program was modeled as closely as possible to the specification.
- A member of the audience commented that there might be a size and/or speed tradeoff effect operating in this instance. That is, the increased size in terms of more modularity might contribute to its increased speed.
- The layered approach often requires extra overhead in additional procedure calls. Hamilton noted that several hundred extra bytes were attributable to this overhead.
- No attempt was made to use macros to decrease the overhead.

PANEL #1

EVALUATING SOFTWARE DEVELOPMENT CHARACTERISTICS

D. Weiss, Naval Research Laboratory
J. Page, Computer Sciences Corporation
V. Basili, University of Maryland

EVALUATING SOFTWARE DEVELOPMENT CHARACTERISTICS:
A Comparison Of Software Errors In Different Environments

David M. Weiss
Naval Research Laboratory

Introduction

According to the mythology of computer science, the first computer program ever written contained an error. Error detection and error correction are now considered to be the major cost factors in software development [Boe72, Boe73, Wol74]. Much current and recent research is devoted to finding ways to prevent software errors. One result is that techniques claimed to be effective for preventing errors are in abundance. Unfortunately, there have been few empirical attempts to verify that proposed techniques work well in production environments. Indeed, there have been few attempts even to collect data that could yield insight into the issues involved. The purpose of this paper is to compare error data obtained from two different software development environments.

To obtain data that was complete, accurate, and meaningful, a goal-directed data collection methodology was used. The approach was to monitor changes made to software concurrently with its development. The results reported here were obtained by applying the methodology to three projects at NASA/GSFC, and one project at the Naval Research Laboratory (NRL). Although all changes were monitored for most projects, we are concerned here only with results obtained from the error data, and only with data that may be used to compare the two environments. Readers interested in a more detailed description of the research methodology or other analyses using other data from the same sources are referred to [Bas81, Wei79, Wei81].

Research Methodology

The methodology is goal oriented. It starts with a set of questions to be answered, and proceeds step-by-step through the design and implementation of a data collection and validation mechanism. Analysis of the data yields answers to the questions of interest, and may also yield a new set of questions. The procedure relies heavily on an interactive data validation process; those supplying the data are interviewed for validation purposes concurrently with the software development process. The methodology has six basic steps, as described in the following.

1. Establish the goals of the data collection.
Many (but not all) of our goals are related to claims made for the software development methodology being used. As an example, a goal of a particular methodology might be to develop software that is easy to change. The corresponding data collection goal is to evaluate the success of the developers in meeting this goal, i.e. evaluate the ease with which the software can be changed.

2. Develop a list of questions of interest
Once the goals of the study are established, they are used to develop a list of questions to be answered by the study. In general, each goal will result in the generation of several different questions of interest. For example, if the goal is to evaluate the ease with which software can be changed, we may identify questions of interest such as: "Is it clear where a change has to be made?", "Are changes confined to a single modules?", "What was the average effort involved in making a change?"
3. Establish data categories
Once the questions of interest have been established, categorization schemes for the changes and errors to be examined may be constructed. Each question generally induces a categorization scheme. If one question is, "How many errors result from requirements changes?", one will want to classify errors according to whether or not they are the result of a change in requirements.
4. Design and test data collection forms
To provide a permanent copy of the data and to reinforce the programmers' memories, a data collection form is used. Forms design was one of the trickiest parts of the studies conducted, and will not be discussed here.
5. Collect and validate data
Data are collected by requiring those people who are making software changes to complete a change report form for each change made, as soon as the change is completed. Validation consists of checking the forms for correctness, consistency, and completeness, and interviewing those filling out the forms in cases where such checks reveal problems. Both collection and validation are concurrent with software development.
6. Analyze the data
Data are analyzed by calculating the parameters and distributions needed to answer the questions of interest.

To apply the methodology to the collection of change data, the following definitions were used.

A change is an alteration to baselined design, code or documentation.

An error is a discrepancy between a specification and its implementation.

A modification is a change made for any reason other than to correct an error.

The Projects Studied

The studies reported here contain complete results from four different projects. Two different environments and several different methodologies were used. One environment was a research group at the Naval Research Laboratory (NRL), and the other was a NASA software production environment at Goddard Space Flight Center. Table 1 is an overview of the data collected for each project. For the ARF project, only error data were collected. Table 2 gives the values of parameters often used to characterize software development projects.

The Architecture Research Facility

The purpose of the Architecture Research Facility (ARF) project, developed at NRL, was to develop a facility for simulating different computer architectures. The simulation is based on a description of the target architecture written in the Instruction Set Processor language [Bel71]. A complete description of the ARF simulator is available elsewhere [Elo79]. Briefly, to simulate a machine, the ARF uses a set of tables that describe the machine being simulated and its state, a module to perform instruction simulation, and a module to handle the interface to the user. The machine description contained in the tables is produced by an ISP compiler (an existing compiler was used)

The ARF was developed by a team of nine people, not all full time. Development took about ten months and 192 people-weeks, exclusive of consulting and secretarial support, to develop. The delivered system contained about 20,000 lines of FORTRAN code.

The primary goal of the ARF designers was to produce a working simulator that would permit the simulation of small target-machine programs. The designers also viewed the ARF development as an experiment in the application of software engineering technology [Elo79]. The key parts of the technology used are the following.

- * Rather than developing the whole system at one time, the ARF was to be done using the family approach to software development [Par76]. The system was to be built in three main stages. Each stage would produce a member of the ARF "family" of programs, providing different facilities.
- * The information-hiding principle [Par72a] was to be applied to conceal design decisions that were expected to change during the lifetime of the ARF.
- * Informal design specifications, followed by standardized interface specifications, followed by high-level language coding specifications were written for each major module of the ARF before any code was written. Each specification was reviewed before its successor was produced.
- * FORTRAN code was written from the coding specifications, compiled, and then reviewed by someone other than the coder prior to debugging. The coder debugged the code and delivered it for testing. A tester (usually) other than the coder or designer, was selected to test the debugged code.

- * At the possible expense of some run time performance, several debugging aids were designed into the system to make development easier. These included
- a. A method for detecting errors involving improper access to table entries, known as the binding mechanism,
 - b. A consistent execution-time error reporting scheme for table interface functions, and
 - c. A mechanism for inserting, and turning on and off, debugging code through the use of a compile-time preprocessor.

The Software Engineering Laboratory

The Software Engineering Laboratory (SEL) is a NASA sponsored project to investigate the software development process, based at Goddard Space Flight Center (GSFC). A number of different software development projects are being studied as part of the SEL investigations [Bai81, Bas77]. Studies of changes made to the software as it is being developed constitute one part of those investigations.

Typical projects studied by the SEL are medium size FORTRAN programs that compute the position (known as attitude) of unmanned spacecraft, based on data obtained from sensors on board the spacecraft. Attitude solutions are displayed to the user of the program interactively on CRT terminals. Because the basic functions of these attitude determination programs tend to change slowly with time, large amounts of design and sometimes code are often re-used from one program to the next. The programs range in size from about 20,000 to about 120,000 lines of source code. They include subsystems to perform such functions as reading and decoding spacecraft telemetry data, filtering sensor data, computing attitude solutions based on the sensor data, and providing an (interactive) interface to the user.

Development is done by contract in a production environment, and is often separated into two distinct stages. The first stage is a high-level design stage. The system to be developed is organized into subsystems, and then further subdivided. For the purposes of the SEL, each named entity in the system is called a component. The result of the first stage is a tree chart showing the functional structure of the subsystem, in some cases down to the subroutine level, a system functional specification describing, in English, the functional structure of the system, and decisions as to what software may be reused from other systems.

The second stage consists of completing the development of the system. Different components are assigned to (teams of) programmers, who write, debug, test, and integrate the software. Before delivery, the software must pass a formal acceptance test. On some projects, programmers produce no intermediate specifications between the functional specifications produced as part of the first stage and the code. Some projects produce pseudo-code specifications for individual subroutines before coding them in FORTRAN. During the period of time that the SEL has been in existence, a structured FORTRAN preprocessor has come into general use.

In distinction to the ARF developers, NASA is not concerned with experimenting with new software engineering techniques. It is concerned with introducing improved techniques into its software development process.

Nonetheless, the principal design goal of the major SEL projects is to produce a working system in time for a spacecraft launch. Results from SEL studies of three different NASA projects, denoted SEL1, SEL2, and SEL3, are included here.

Project	Number of Changes	Number of Modifications	Number of Errors
SEL1	281	101	180
SEL2	229	110	119
SEL3	760	453	307
ARF			143

Table 1 Overview of Data Collected

Project	Effort (Months)	Number of Developers	Lines of Code (K)	Dev. Lines of Code (K)	Number of Components
SEL1	79.0	5	50.9	46.5	502
SEL2	39.6	4	75.4	31.1	490
SEL3	98.7	7	85.4	78.6	639
ARF	44.3	9	21.8	21.8	253

Table 2 Summary of Project Information

Project	Errors Per K Lines Of Developed Code	Errors Resulting From Change (As Percentage Of NonClericals)	Repeated Error Ratio (Average Number Of Corrections Per Error)
SEL1	3.9	5	1.02
SEL2	3.8	14	1.08*
SEL3	3.9	12	1.05
ARF	6.6	13	1.007

* Upper bound. Exact number of repeated errors for SEL2 is unknown. By conservative means, the ratio could be estimated as 1.04.

Table 3 Measures of Erroneous Change

Results

The results presented here are derived from analyses of several different data parameters and distributions. Table 3 shows error density, errors resulting from change, and repeated error ratio for each project. These parameters indicate that for all projects most changes were made correctly on the first attempt.

Figures 1 and 2 are an overview of the change distributions for the SEL projects (recall that data on modifications is not available for the ARF project). Figure 3 shows sources of modifications, i.e. reasons for modifying the software, and figure 4 shows sources of nonclerical errors. Although there were a significant number of requirements changes for two of the SEL projects, none of the projects show a significant number of errors resulting from incorrect or misunderstood requirements.

For all projects, the major source of errors was the design and implementation of single components. (For these projects, a single component is nearly always a FORTRAN subroutine or block data.) Relatively few errors were the result of misunderstandings of requirements, specifications, programming language or compiler, or software or hardware environment. Aspects of the design involving more than one component was also not a major source of errors. Figure 5 shows a continuation of the same pattern. For most projects, interfaces were not a significant source of errors.

A further categorization of design and implementations errors, including both single and multi-component design errors is shown in figure 6. The pattern for the SEL and ARF projects is quite different here; relatively few ARF errors involved the use (including definition, representation, and access) of data. For the SEL projects, data errors were a significant fraction of design and implementation errors.

A direct measure of ease of error correction is shown in figure 7. For all projects, the overwhelming majority of errors took less than a day of effort to correct. Indeed, most error corrections took an hour or less of effort.

Figure 8 is a measure of locality of errors with respect to project components. Only components that required at least one error correction (one fix) are represented. The majority of such components required no more than one correction. For all projects, 80% or more of such components were corrected at most three times.

Locality of errors with respect to project subsystem (project module for the ARF), is shown in figure 9. The distributions here show the reverse pattern of those in figure 8, i.e. most corrections are clustered in a few subsystems (modules).

Conclusions

The ARF and SEL projects involved different applications and were developed in different environments, using different methodologies, people with different backgrounds, and different computer systems. Despite these differences there are a number of similarities between the two, as listed in the following.

1. There is a common pattern to the sources of error distributions. The principle error source is in the design and implementation of single routines. Requirements, specifications and interface misunderstandings are all minor sources of errors.
2. Few errors are the result of changes, few errors require more than one attempt at correction, and few error corrections result in other errors.
3. Relatively few errors take more than a day to correct.

These similarities may be explained by different factors in the different environments. The SEL projects may be viewed as redevelopments. Much of the same design and some of the same code is reused from one project to the next. As a result of experience with the application, the changes most likely to occur from one project to the next have been identified by the designers. The systems are now designed so that these changes are easy to make. Confirmation of this explanation was provided by one of the primary system designers in discussions held after the data were analyzed.

In the ARF environment, the explicit use of techniques to identify and design for potential changes is a likely contributing factor to the similarities in the distributions.

Common factors to both the SEL and ARF projects were the stability of the hardware and software supporting the development and the familiarity of the programmers with the language they were using.

The most striking difference between the ARF and SEL projects is in the proportion of intended use to data errors. The ARF project has a considerably smaller proportion of data errors than the SEL projects. One reason for this may be the conscious attempt of the ARF developers to apply abstract data typing and strong typing in their design.

Acknowledgements

Support for a research project involving data collection in a production environment must come from many sources. These sources include project management, the programmers supplying the data, those maintaining the data base (in both paper and computerized form), those assisting in data analysis, and those providing technical review and guidance. A few of the people providing such support were Frank McGarry, Drs. Victor Basili, David Parnas, John Shore, and Gerald Page, Honey Elovitz, Alan Parker, Jean Grondalski, Sam DePriest, Joanne, Shana, and Joshua Weiss, and Kathryn Kragh.

References

- [Bai81] J. Bailey and V. Basili, "A Meta-Model For Software Development Resource Expenditures," Proc. Fifth Int. Conf. Software Eng., pp. 107-116, 1981
- [Bas77] V. Basili, M. Zelkowitz, F. McGarry, et al., The Software Engineering Laboratory, University of Maryland Technical Report TR-535, May 1977
- [Bas81] V. Basili and D. Weiss, "Evaluation of a Software Requirements Document By Analysis of Change Data," Proc. Fifth Int. Conf. Software Eng., pp. 314-323, 1981
- [Bel71] C. Bell and A. Newell, Computer Structures: Readings and Examples, McGraw-Hill, New York, 1971
- [Elo79] H. Elovitz, "An Experiment In Software Engineering: The Architecture Research Facility As A Case Study, Proc. Fourth Int. Conf. Software Eng., pp. 145-152, 1979
- [Par72a] D. L. Parnas, "A Technique For Software Module Specification With Examples," Comm. ACM, vol. 15 no. 5, May, 1972, pp. 330-336
- [Par76] D. L. Parnas, "On the Design and Development of Program Families," IEEE Trans. Software Eng., vol. SE-2 no. 1, pp. 1-9, 1976
- [Wei79] D. Weiss, "Evaluating Software Development by Error Analysis: The Data from the Architecture Research Facility," J. Systems and Software, vol. 1, pp. 57-70, 1979
- [Wei81] D. Weiss, "Evaluating Software Development By Analysis Of Change Data," Ph.D. Thesis, University of Maryland, 1981

**THE VIEWGRAPH MATERIALS
for the
D. WEISS PRESENTATION FOLLOW**

PURPOSE OF RESEARCH

- * FIND A WAY OF EVALUATING SOFTWARE DEVELOPMENT METHODOLOGIES
- * LEARN ABOUT THE SOFTWARE DEVELOPMENT PROCESS
- * LEARN ABOUT MEASURING THE SOFTWARE DEVELOPMENT PROCESS

APPROACH

- * STUDY CHANGES USING GOAL-DIRECTED DATA COLLECTION

RESEARCH METHODOLOGY DEVELOPED

- * ESTABLISH GOALS

EXAMPLE: EVALUATE THE DIFFICULTY OF CHANGING SOFTWARE

- * DEFINE QUESTIONS OF INTEREST

EXAMPLES: IS IT CLEAR WHERE A CHANGE HAS TO BE MADE?

ARE CHANGES CONFINED TO SINGLE MODULES?

WHAT WAS THE AVERAGE EFFORT INVOLVED IN MAKING A
CHANGE?

- * DESIGN DATA COLLECTION FORM

- * COLLECT AND VALIDATE DATA CONCURRENTLY WITH DEVELOPMENT

- * ANALYZE DATA

TYPES OF CHANGES

- * DEF: A CHANGE IS AN ALTERATION TO (BASELINED) DESIGN, CODE, OR DOCUMENTATION.
- * DEF: AN ERROR IS A DISCREPANCY BETWEEN A SPECIFICATION AND ITS IMPLEMENTATION.
- * DEF: A MODIFICATION IS A CHANGE MADE FOR ANY REASON OTHER THAN TO CORRECT AN ERROR.
- * $\text{CHANGES} = \text{MODIFICATIONS} + \text{ERROR CORRECTIONS}$

SUBCATEGORIES OF CHANGES

* MODIFICATIONS

IMPLEMENTATION OF REQUIREMENTS CHANGE

OPTIMIZATIONS

IMPROVEMENTS OF USER SERVICES

IMPROVEMENT OF CLARITY, MAINTAINABILITY, OR DOCUMENTATION

ADAPTATION TO ENVIRONMENT CHANGE

* ERROR CORRECTIONS

CLERICAL ERRORS

NON-CLERICAL ERRORS

REQUIREMENTS INCORRECT OR MISINTERPRETED

SPECIFICATIONS INCORRECT OR MISINTERPRETED

DESIGN ERROR INVOLVING SEVERAL COMPONENTS

ERROR IN DESIGN/IMPLEMENTATION OF A SINGLE COMPONENT

ERROR IN USE OF PROGRAMMING LANG OR COMPILER

MISUNDERSTANDING OF ENVIRONMENT

Project	Number of Changes	Number of Modifications	Number of Errors
SEL1	281	101	180
SEL2	229	110	119
SEL3	760	453	307
ARF			143
A-7	88	9	79

Table 5.4a Overview of Data Collected

Project	Effort	Number of Developers	Lines of Code (K)	Dev. Lines of Code (K)	Number of Components
SEL1	79.0	5	50.9	46.5	502
SEL2	39.6	4	75.4	31.1	490
SEL3	98.7	7	85.4	78.6	639
ARF	44.3	9	21.8	21.8	253
A-7					

Table 5.4b Summary of Project Information

Project	Changes Per K Lines Of Developed Code	Errors Per K Lines Of Developed Code	Error To Mod Ratio (NonClericals Only)
SEL1	6.0	3.9	1.3
SEL2	7.4	3.8	.92
SEL3	9.7	3.9	.54
ARF		6.6	

Table 5.5 Change and Error Densities

Project	Erroneous Change Rate (Ratio Of Changes Resulting In Errors To All Changes)	Errors Resulting From Change (As Percentage Of NonClericals)	Repeated Error Ratio (Average Number Of Corrections Per Error)
SEL1	.025	5	1.02
SEL2	.061	14	1.08*
SEL3	.041	12	1.05
ARF		13	1.007

* Upper bound. Exact number of repeated errors for SEL2 is unknown. By conservative means, the ratio could be estimated as 1.04.

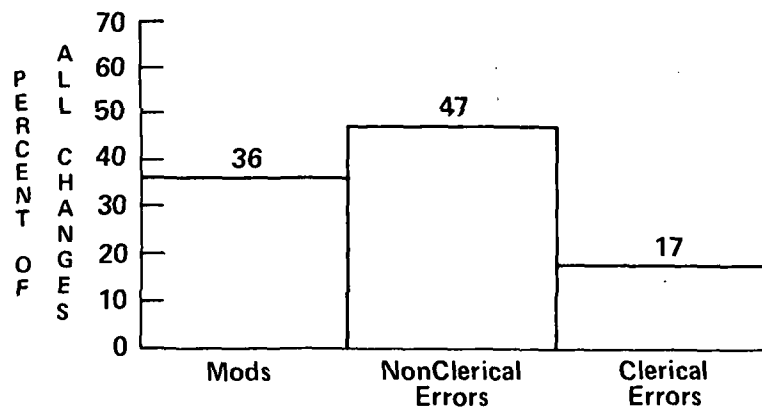
Table 5.6 Measures of Erroneous Change

Project	Number Of People	Errors Per Person
SEL2	4	25
SEL1	5	26
SEL3	7	44
ARF	9	10

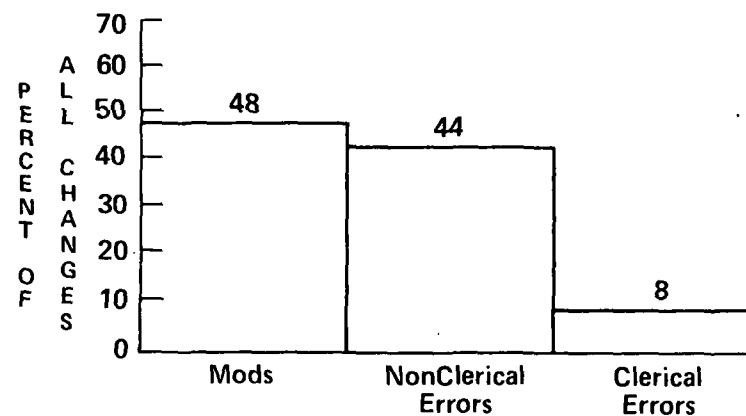
Table 5.7 Errors Per Person By Number Of People

Project	Effort (People-Months)	Errors Per Person-Month	Changes Per Person-Month
SEL2	39.6	2.4	5.8
ARF	44.3	2.1	
SEL1	79.0	1.7	3.6
SEL3	98.7	3.1	7.7

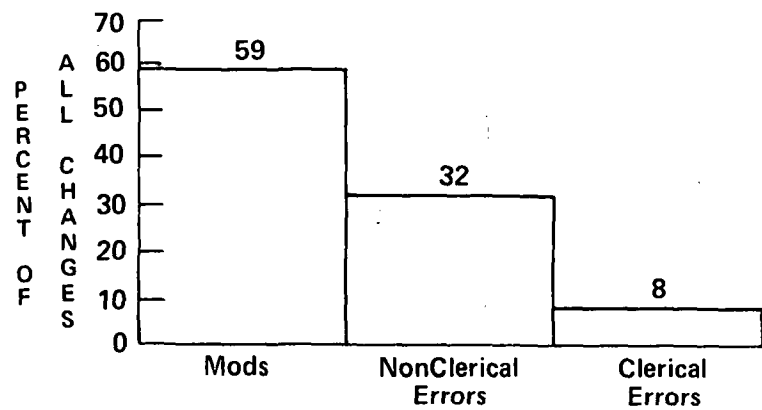
Table 5.8 Errors Per Effort By Effort



Change Type
SEL1



Change Type
SEL2



Change Type
SEL3

Figure 5.1 Changes.

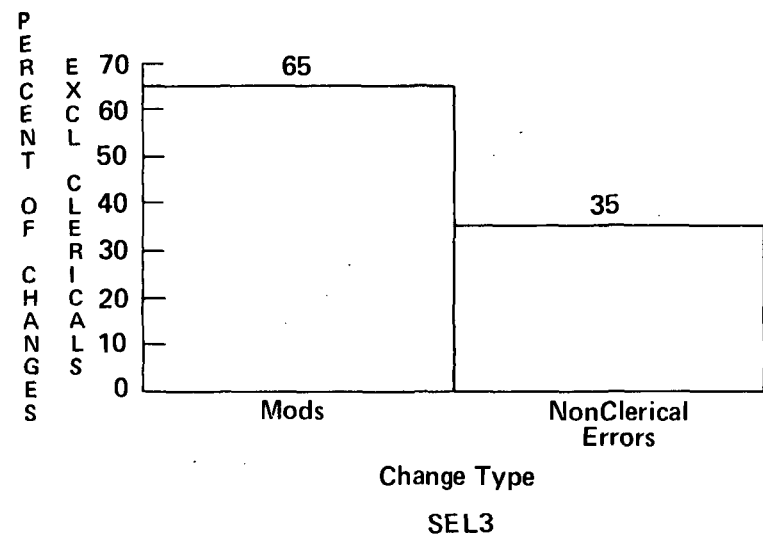
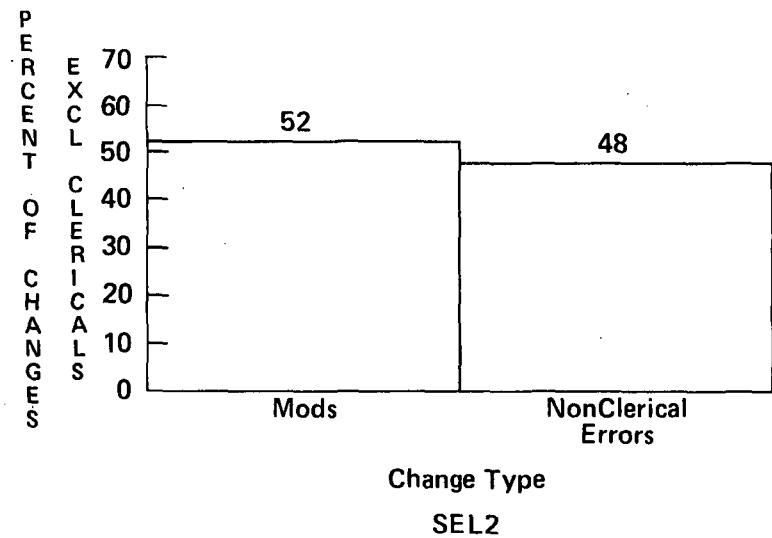
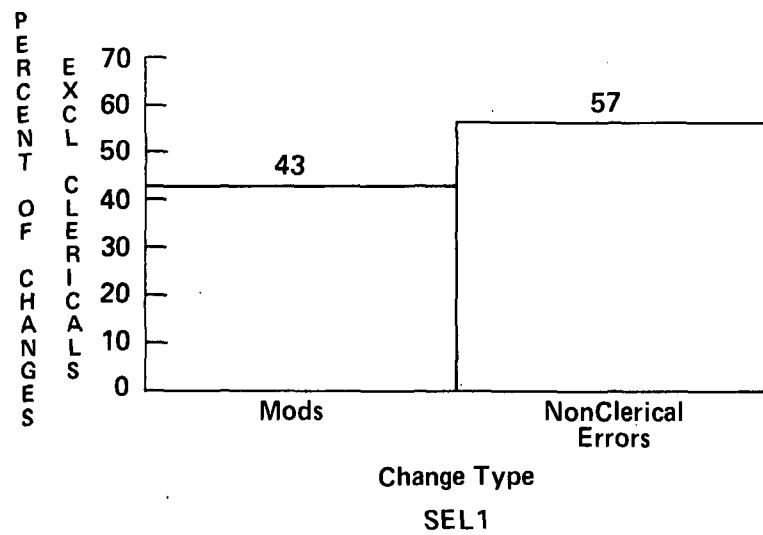
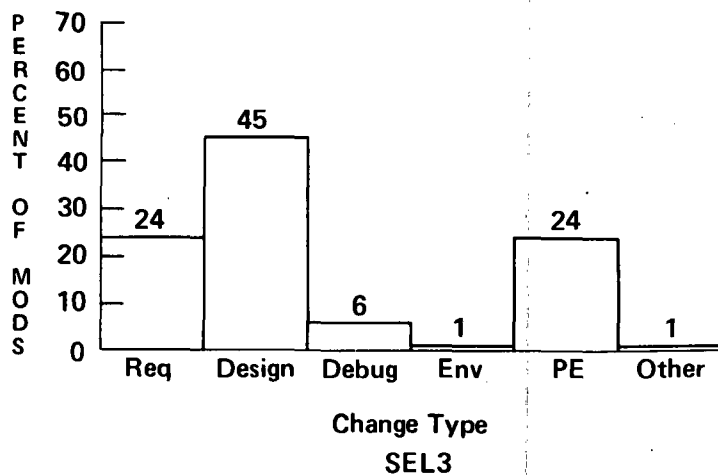
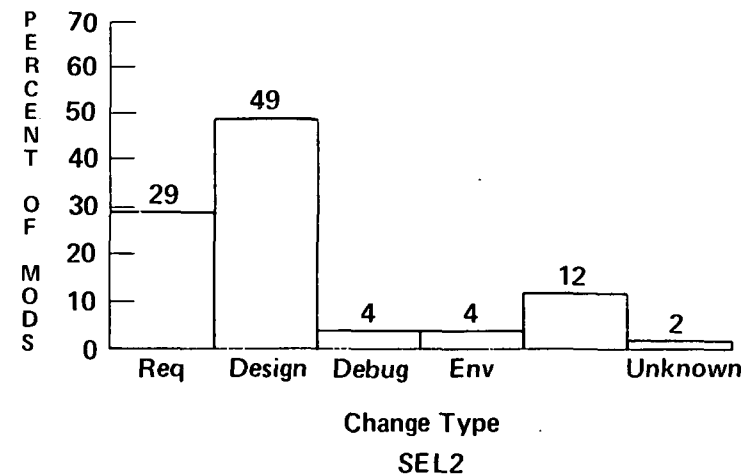
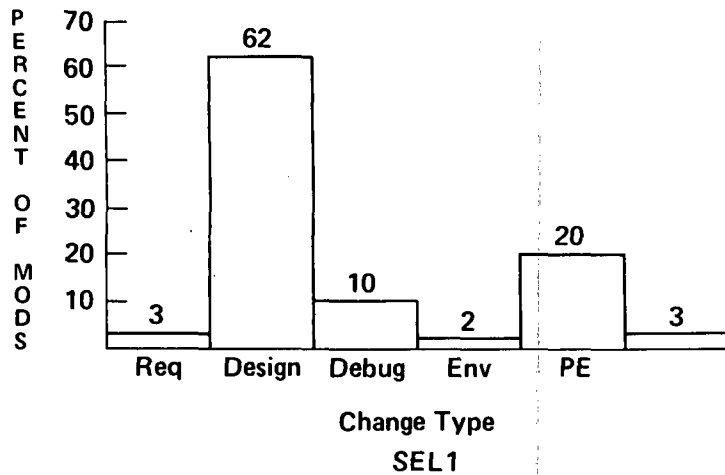


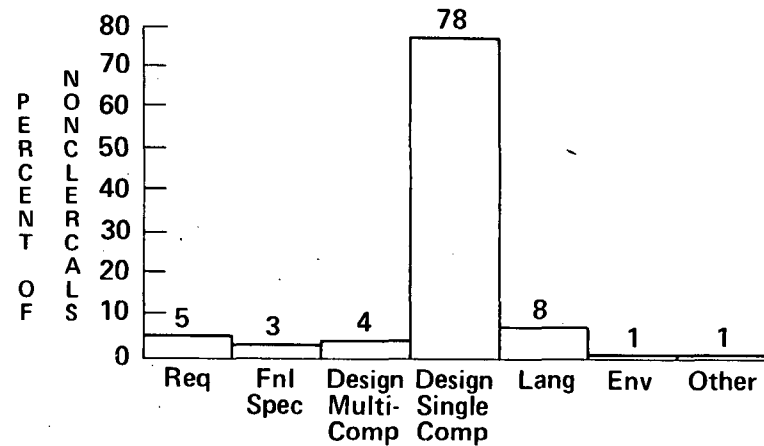
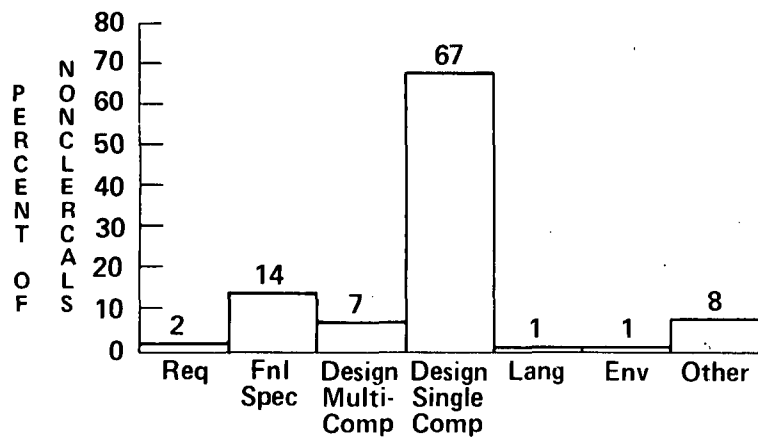
Figure 5.2 Changes (Clerical Errors Excluded).



Key to Figure 5.3

- Design Modifications caused by changes in design
- Debug Modifications to insert or delete debug code
- Env Modifications caused by changes in the hardware or software environment
- PE Planned Enhancements
- Req Modifications caused by changes in requirements or functional specifications
- Unknown Causes of these modifications are not known

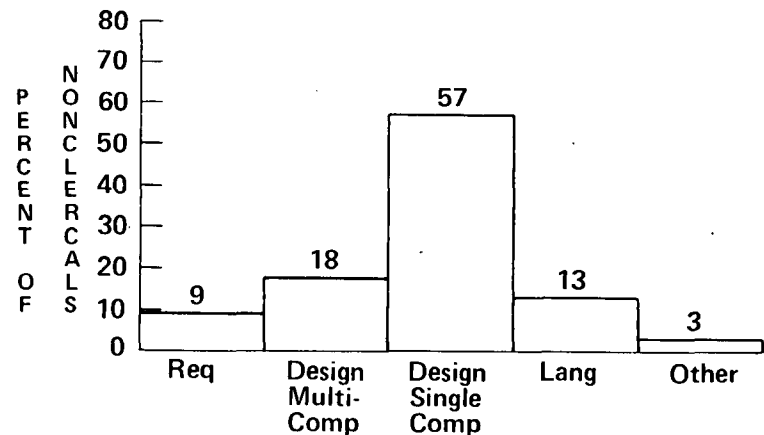
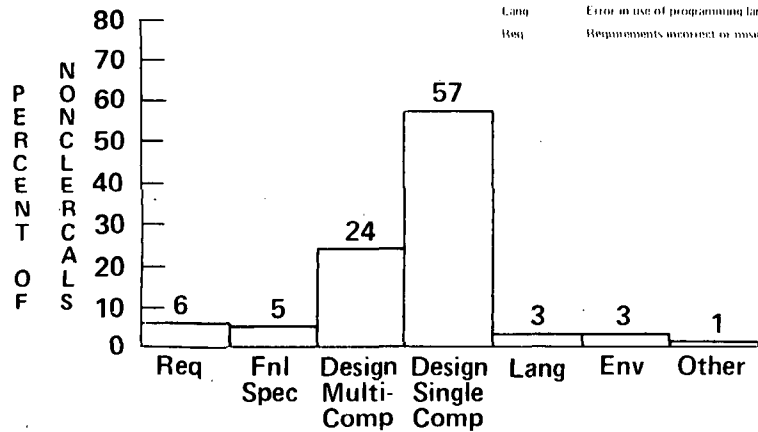
Figure 5.3. Sources of Modifications.



Type of Error

Type of Error

Key To Figure 5.5
 Design Multi-Comp Design error involving several components
 Design Single Comp Error in the design or implementation of a single component
 Env Misunderstanding of external environment, except language
 Fnl Spec Functional specifications incorrect or misinterpreted
 Lang Error in use of programming language/compiler
 Req Requirements incorrect or misinterpreted



Type of Error

Type of Error

Figure 5.5. Sources of Nonclerical Errors.

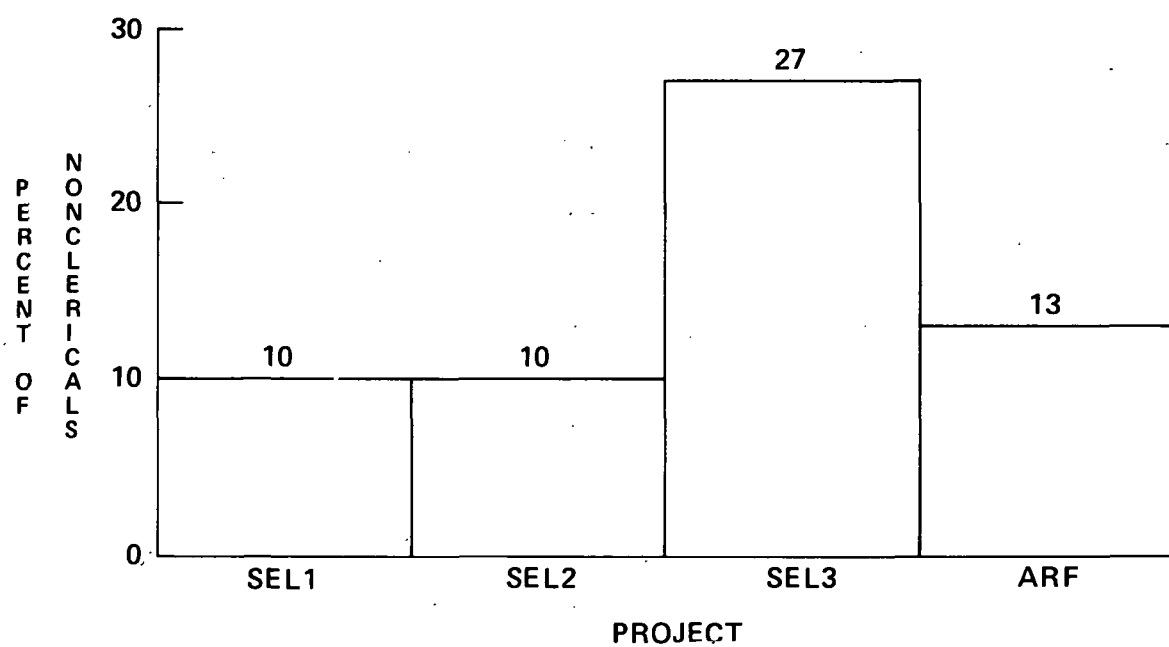
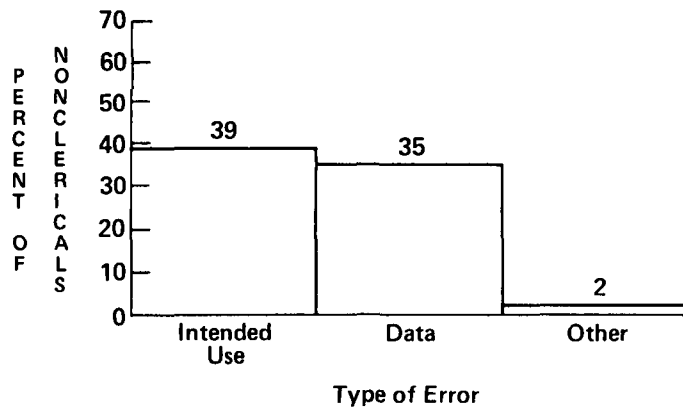
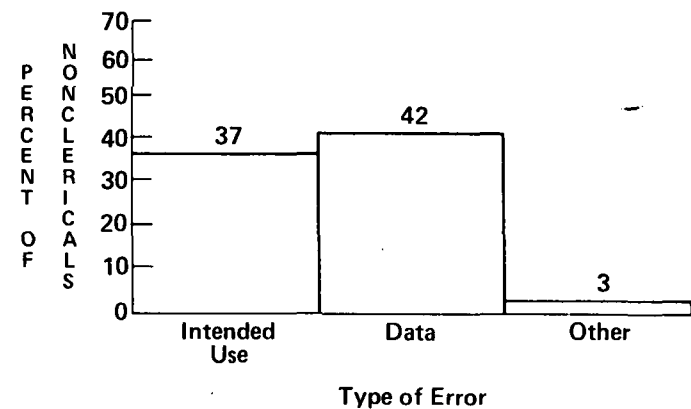


Figure 5.7 Interface Errors.



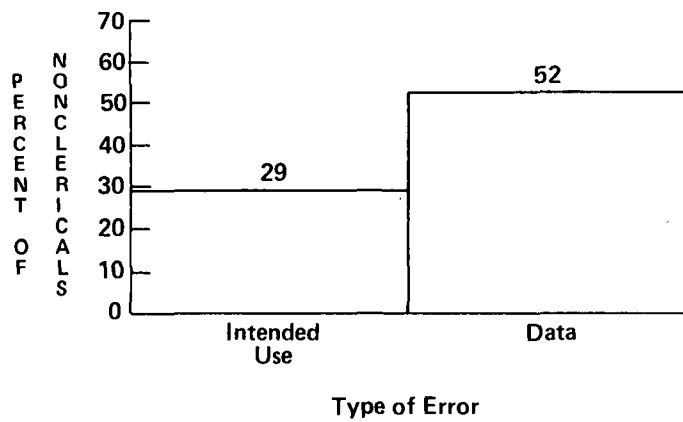
SEL1



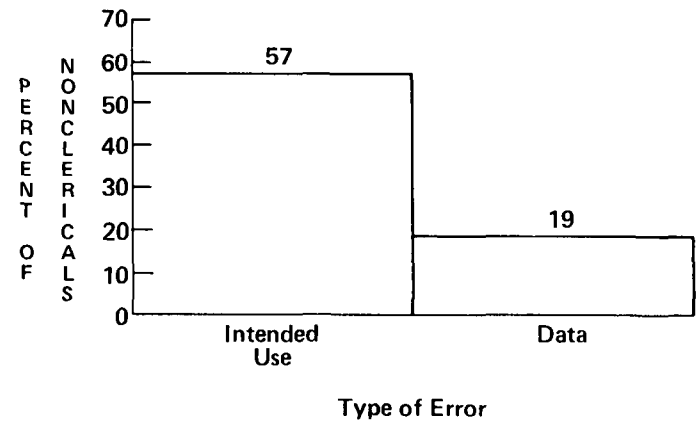
SEL2

Key to Figure 5.6

Data Error in the use of data
 Intended Use Error in intended function, i.e. program behavior does not correspond to the intended use of the program

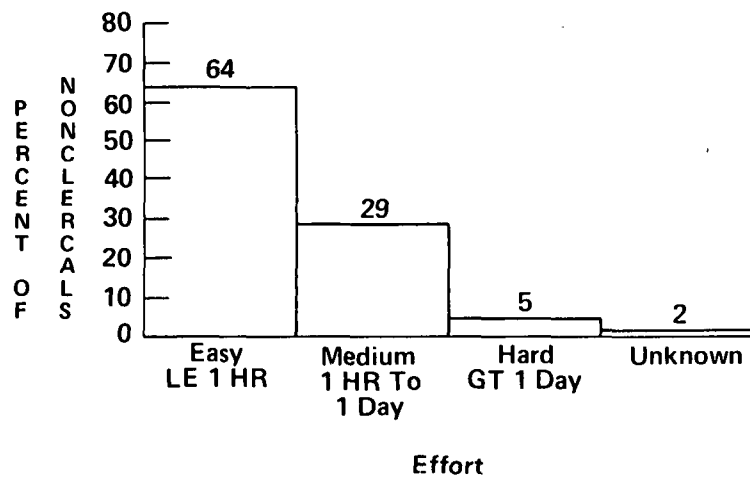


SEL3

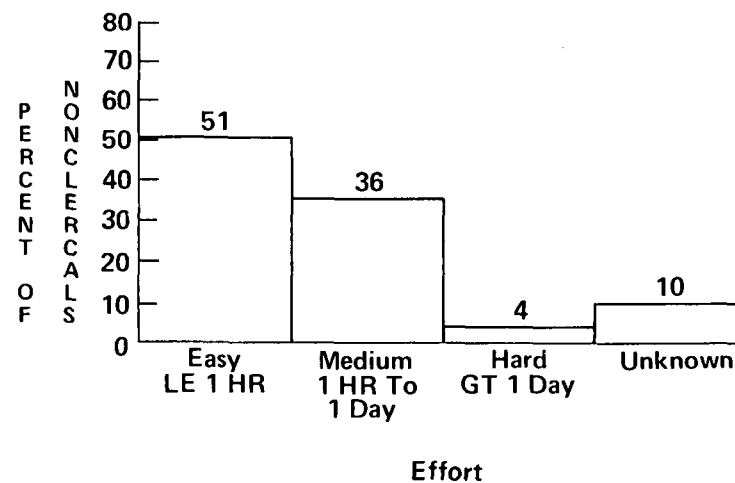


ARF

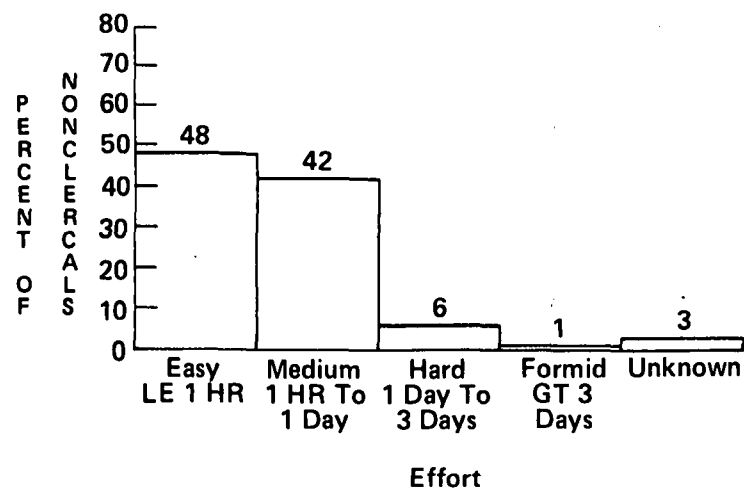
Figure 5.6 Sources of Design/Implementation Errors.



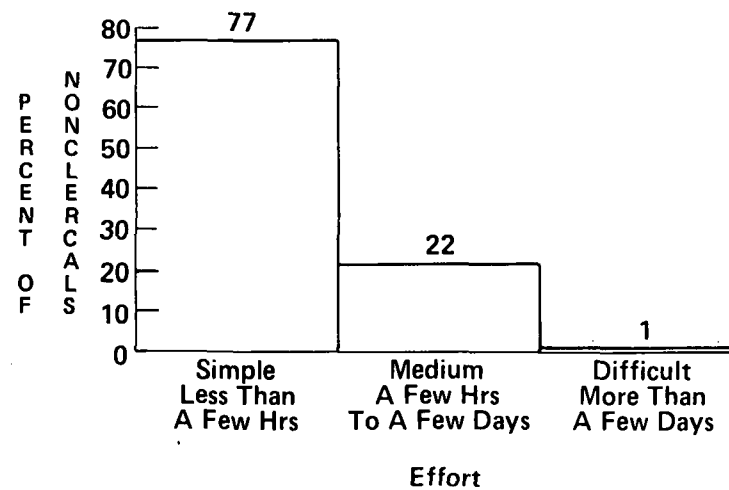
SEL1 Effort to Design Change



SEL2 Effort to Design Change



SEL3 Effort to Make Change



ARF Effort to Fix

Figure 5.10. Effort to Change Nonclerical Errors.

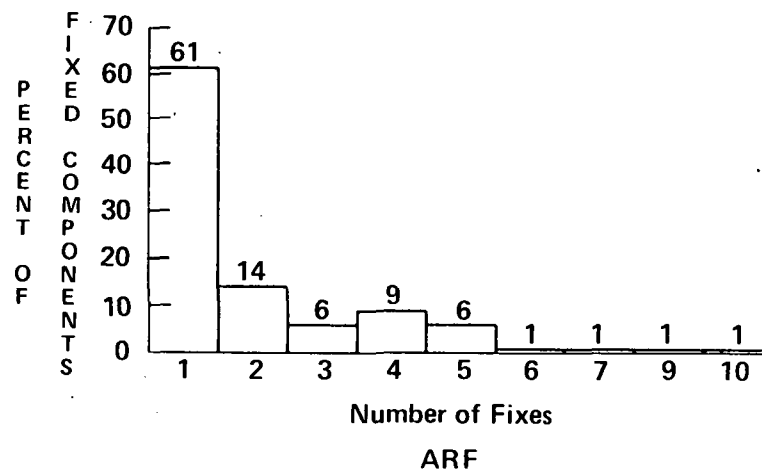
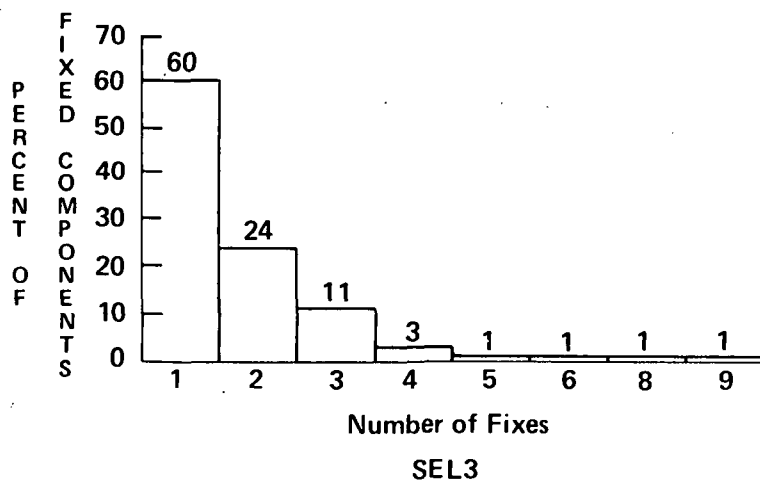
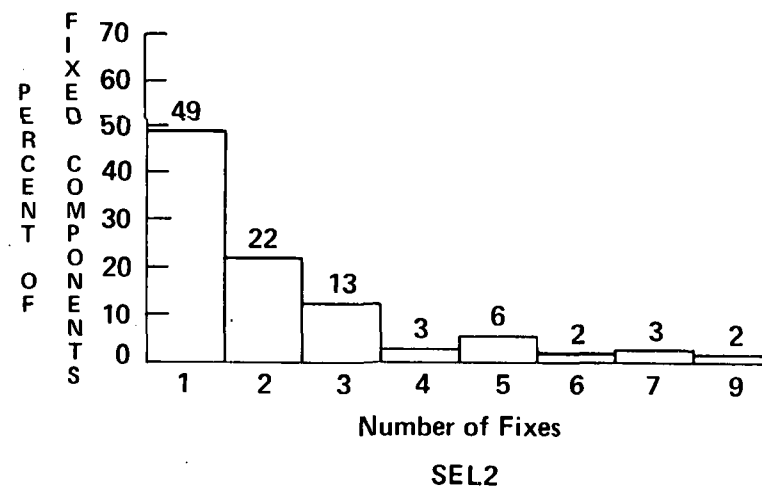
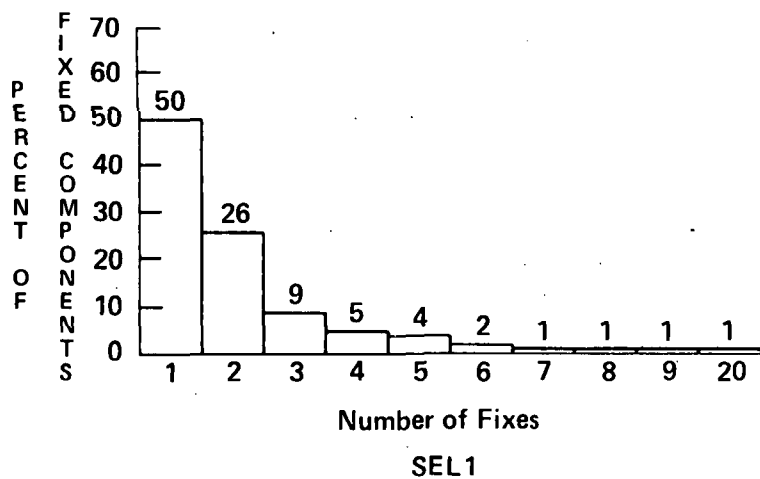


Figure 5.15. Frequency Distribution of Fixes.

CONCLUSIONS ABOUT SOFTWARE DEVELOPMENT COMMON TO NRL AND NASA/GSFC

- * PRINCIPAL ERROR SOURCE IS DESIGN AND IMPLEMENTATION OF SINGLE ROUTINES
REQUIREMENTS, SPECIFICATIONS, AND INTERFACE MISUNDERSTANDINGS ARE
MINOR SOURCES OF ERRORS.
- * FEW ERRORS ARE THE RESULT OF CHANGES, FEW ERRORS REQUIRE MORE THAN
ONE ATTEMPT AT CORRECTION, AND FEW ERROR CORRECTIONS RESULT IN OTHER
ERRORS.
- * RELATIVELY FEW ERRORS TAKE MORE THAN A DAY TO CORRECT.

DIFFERENCES BETWEEN ARF AND SEL SOFTWARE DEVELOPMENT

- * THE PROPORTION OF ARF ERRORS INVOLVING DATA IS CONSIDERABLY SMALLER
THAN THE CORRESPONDING PROPORTION FOR SEL ERRORS

METHODOLOGY EVALUATION:
EFFECTS OF INDEPENDENT VERIFICATION
AND INTEGRATION ON ONE CLASS OF
APPLICATION

Jerry Page
COMPUTER SCIENCES CORPORATION
and
GODDARD SPACE FLIGHT CENTER
SOFTWARE ENGINEERING LABORATORY

Prepared for the
NASA/GSFC
Sixth Annual Software Engineering Workshop

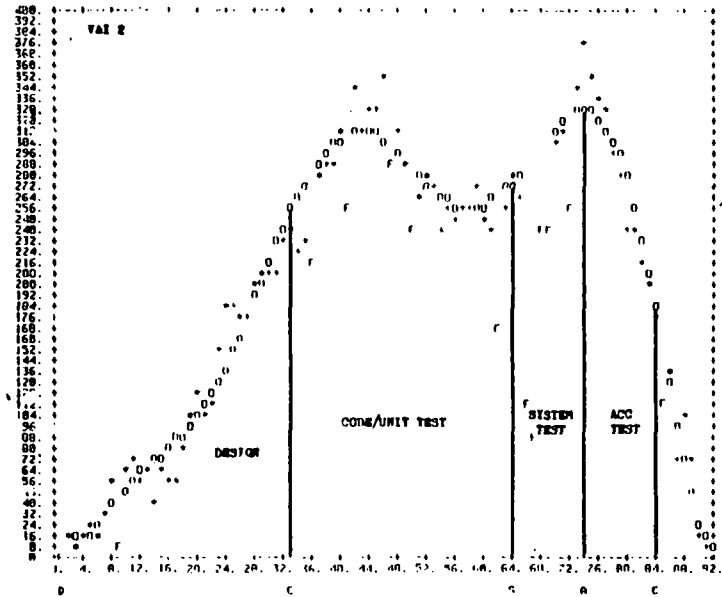
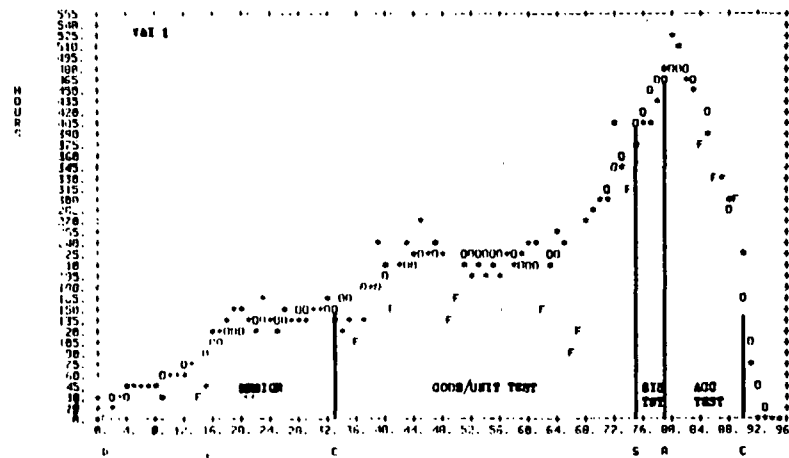
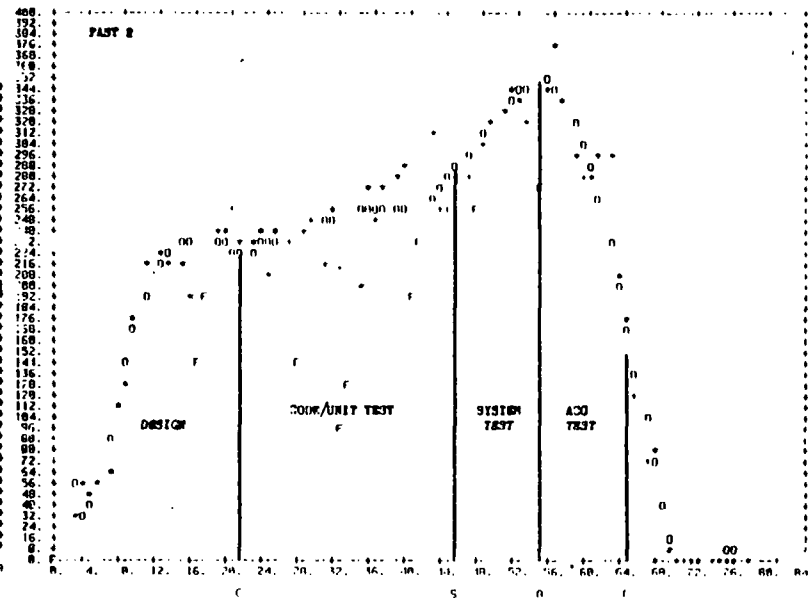
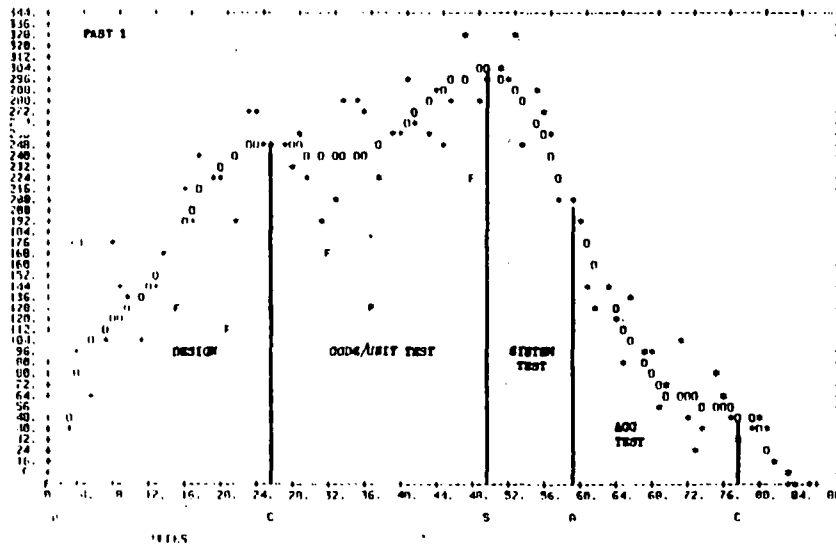
METHODOLOGY EVALUATION:

**EFFECTS OF
INDEPENDENT VERIFICATION
AND INTEGRATION ON
ONE CLASS OF APPLICATION**

Viewgraph 1: Title

One area of study in the Software Engineering Laboratory (SEL) is methodology.¹ This presentation describes the effects of an independent verification and integration (V&I) methodology on one class of application. V&I is the name that we will use for what some call independent verification and validation (IV&V) and others call verification and validation (V&V). "One class of application" means the development of solutions for a set of similar problems (ground-based support for satellite operations) that are developed in the same computing environment--simply put, a specific problem in a specific environment.

¹Goddard Space Flight Center, SEL-81-104, "The Software Engineering Laboratory" (Software Engineering Laboratory Series), D. N. Card et al., February 1982.



Viewgraph 2: Resource Profiles

Why use a V&I methodology? Why have we experimented with a V&I methodology? To introduce V&I methodology, let me show you resource profiles for four real projects developed for the Goddard Space Flight Center (GSFC) by Computer Sciences Corporation (CSC) and monitored closely by the SEL. These resource profiles show technical hours charged to the projects by week. Technical hours are those hours charged by the programmers and the first-line managers. First-line managers are those managers who make decisions, set priorities, and solve problems daily, as opposed to higher level managers who receive weekly or less frequent progress reports. These resource profiles also do not include service charges, which amount to approximately 13 percent of the hours charged to a project. Service hours include those hours charged by librarian, secretarial, technical publications, and data technician support groups.

In these profiles, design activity starts at the far left-hand side and continues throughout the project at decreasing levels. The first vertical line indicates the conclusion of a series of requirements analysis and critical design reviews. It is the point at which implementation and corresponding testing are allowed to begin. The second vertical line is the point at which implementation (coding) is supposed to be complete and system testing starts. The third vertical line is the point at which the software is supposed to be ready (for operation) and acceptance testing starts. The fourth vertical line indicates the end of acceptance testing and the beginning of maintenance (by another group).

Most people who measure software products apply many measures to the software product from the point at which it enters the maintenance and operation (M&O) phase. We do too, but since we have no responsibility for the software once it

is transferred to the maintenance group and because it is more difficult to collect data through another group, we apply many of our measures one or two phases earlier, i.e., from the beginning of acceptance testing or from the beginning of system testing.

As you can see from three of these four profiles (excluding the one in the upper left-hand quadrant), the peak effort is at the start of acceptance testing. Some of the reasons that the peak effort occurs at that point are

- All the projects grow between 15 and 40 percent after the start of implementation because of requirements escalation.
- These projects cross two or three funding periods. This puts some constraint on how much work can be done in any one funding period.
- Management problems exist. The profile in the lower left-hand quadrant shows the application of the "mythical man-month."
- There is a hard deadline (launch of a satellite).
- The computers are not very reliable (6- to 8-hour mean time to failure).

We know what we are doing during that peak effort (the peak at the third vertical line). A large fraction of our work there is correcting errors.

It is commonly accepted that the cost to correct an error approximately doubles as it enters each new phase of the development life cycle. For example, if an error originates in the requirements phase (the phase preceding design) and if that requirements error gets designed, the cost to correct the error during design will be one to two times more than to correct the error in the requirements phase. If the designed requirements error gets implemented, the cost to

correct the error during implementation will be two to four times more than to correct the error in the requirements phase. If the implemented requirements error enters the system testing phase, the cost to correct the error will be four to eight times more. If the implemented requirements error enters the acceptance testing phase, the cost to correct the error will be 8 to 16 times more. If it enters the M&O phase, the cost to correct the error will be 16 to 32 times more (for one simplified example, see Figure 1).

The same progression holds for errors that originate in design and implementation. Therefore, during the M&O phase, even implementation errors are costly to correct; they cost four to eight times more to correct during the M&O phase than during the implementation phase.

We do not need a general hypothesis to know that it costs more to correct errors in the later stages of development. Our own data collected over the last 5 years shows that some increase occurs in the cost of correcting errors from one phase of development to the next. SEL data shows that (regardless of error type) the average error discovered during the acceptance testing phase costs more to correct than the average error discovered during the system testing phase and that the average error discovered during the system testing phase costs more to correct than the average error discovered during the implementation phase. The increase in the average effort to correct the average error from one phase to the next varies from project to project, but it frequently approximates a doubling of effort.

Common sense indicates that there will be cost increases for changes to the evolving product as development progresses through the life cycle. Certainly, in this environment there are several transfers of responsibility: from the requirements team to the development team, from the

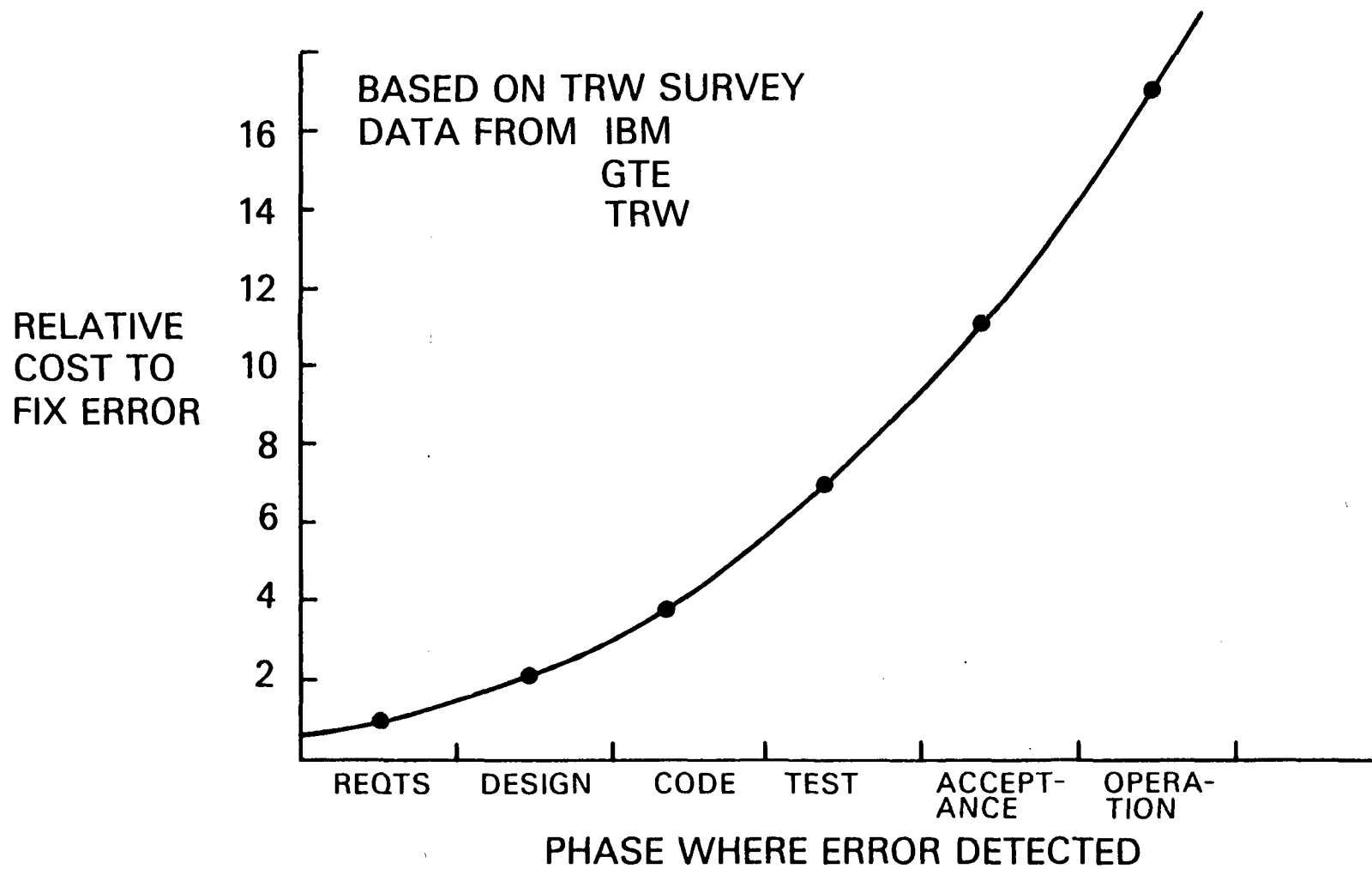


Figure 1. Cost of Correcting Software Errors

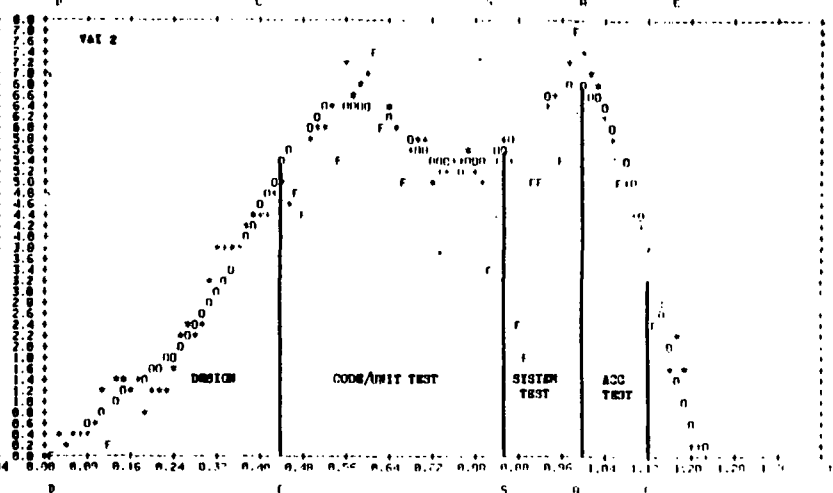
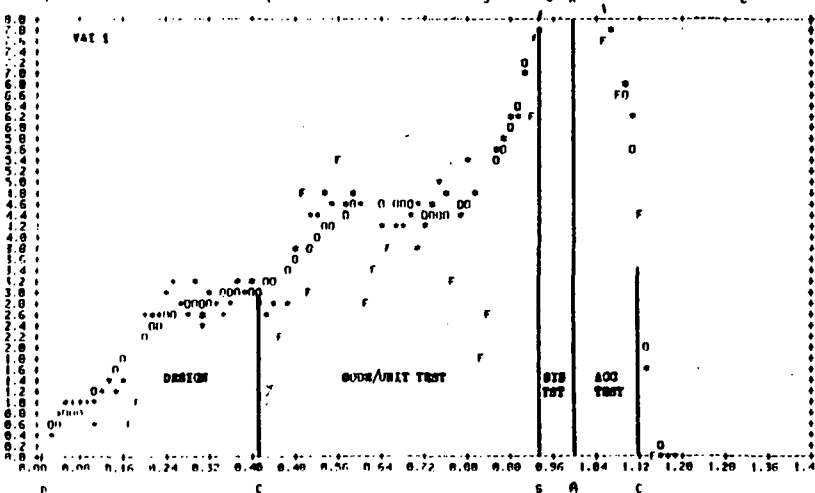
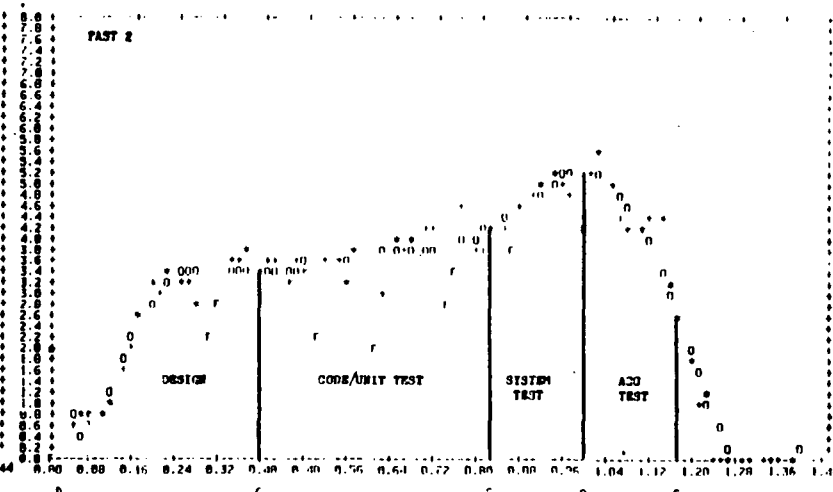
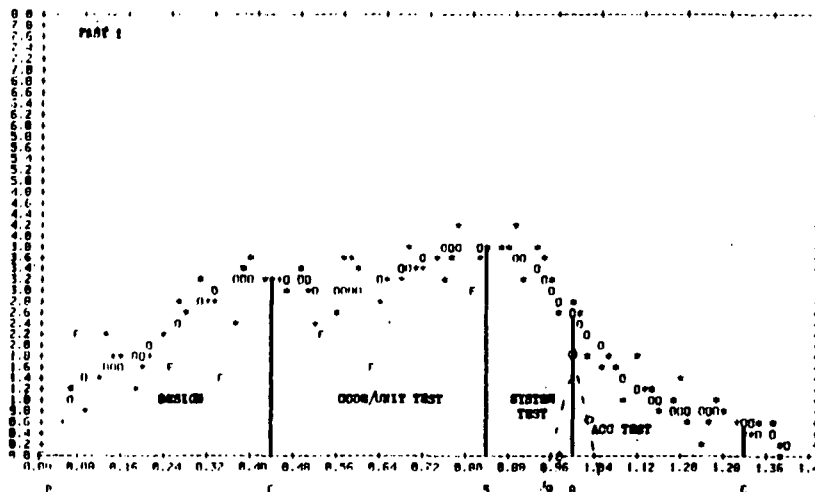
designers to the implementers, from the implementers to the testers, and finally, from the development team to the maintenance team. These are not complete transfers of responsibility; instead, the team size increases or decreases at different points in the development life cycle. Because a system is never 100-percent completely or accurately documented and because few people can instantaneously absorb the content of the documentation, new team members will require additional time to become familiar with the system. Therefore, functions will increase in cost when new members or groups become responsible for them.

Since the average development team size is six members, prematurely removing one member from the team always affects the schedule adversely. If the schedule cannot be adjusted (adjustments are more difficult late in the life cycle because of launch deadlines), then a replacement member must be added to the team. This replacement increases cost and it does not solve the schedule problem completely unless the replacement individual is more productive than the individual who was replaced.

We know that we have to improve our methodology, both in management and development practices, to move error-correction efforts earlier into the development life cycle, closer to the commission of the errors.

We know this from the advocates of V&I methodology, from our own SEL data, and from common sense. To save money, we must move the peak effort away from the start of acceptance testing (the third vertical line in the resource profile) and nearer to the design phase (between the first and second vertical lines in the resource profile). For example, we spend approximately 30 percent of our dollars for system and acceptance testing (the area between the second and fourth vertical lines). If 50 percent of that expenditure is for

error correction (15 percent of dollars), then by moving that error-correction effort into the implementation phase, we will reduce the cost of that effort by approximately one-half; i.e., we will save approximately 7.5 percent of our development cost.



Viewgraph 3: Scaled Resource Profiles

These resource profiles are scaled so that the start of acceptance testing is 1 on the x-axis. The technical hours spent each week (the y-axis) are scaled by the developed lines of code (in thousands). The scaled resource profiles show technical hours per thousand lines of developed code by fraction of development life cycle. The unscaled resource profiles (see viewgraph 2) show technical hours by week of development life cycle.

DEVELOPMENT ENVIRONMENT

CHARTER: DESIGN, IMPLEMENT, TEST, DOCUMENT
TYPE OF SOFTWARE: SCIENTIFIC, GROUND-BASED, NEAR-REAL-TIME, INTERACTIVE GRAPHIC
LANGUAGES: 85% FORTRAN, 15% ASSEMBLER MACROS
MACHINES: IBM S/360-75 AND -95, BATCH WITH TSO

PROCESS CHARACTERISTICS:	<u>AVERAGE</u>	<u>HIGH</u>	<u>LOW</u>
DURATION (MONTHS)	15.6	20.5	12.9
EFFORT (STAFF-YEARS)	8.0	11.5	2.4
SIZE (1000 LOC)			
DEVELOPED	57.0	111.3	21.5
DELIVERED	62.0	112.0	32.8
STAFF (FULL-TIME EQUIV.)			
AVERAGE	5.4	6.0	1.9
PEAK	10.0	13.9	3.8
INDIVIDUALS	14	17	7
APPLICATION EXPERIENCE			
MANAGERS	5.8	6.5	5.0
TECHNICAL STAFF	4.0	5.0	2.9
OVERALL EXPERIENCE			
MANAGERS	10.0	14.0	8.4
TECHNICAL STAFF	8.5	11.0	7.0

Viewgraph 4: Development Environment

I will talk about four projects today. Two went into operation about 2 years ago; the other two went into operation about 3 months ago. A V&I methodology was applied to the last two. The last two projects will be labeled V&I 1 and V&I 2 on the following viewgraphs. The projects that became operational 2 years ago will be labeled Past 1 and Past 2.

<u>Date</u>	<u>Past 1</u>	<u>Past 2</u>	<u>V&I 1</u>	<u>V&I 2</u>
Development start	May 1978	June 1978	Oct. 1979	Oct. 1979
Maintenance start	Oct. 1979	Aug. 1979	June 1981	May 1981
Operation start	Feb. 1980	Oct. 1979	Aug. 1981	Aug. 1981
M&O end	Active	Sept. 1980	Active	Active

This viewgraph shows the average value of each development characteristic and the high and low values of the development characteristics from 12 projects in one class of application. The high or the low values themselves do not represent one project but show the most and least of any characteristic attributed to any of the 12 projects. The four projects that I will talk about are included in these statistics.

What is our development environment like? Our development teams design, implement, test, and document software that is scientific, ground-based, near-real-time, and interactive graphic. The software is 85 percent FORTRAN, 1 percent assembler, and 14 percent assembler macros. The assembler macros are required for the graphics capability. The software is developed on the IBM S/360-75 and -95, which are batch oriented with a timesharing option (TSO).

This is an operations environment, not a development environment. In this environment, the developers have access to

the IBM S/360-95 via a Remote Job Processing (RJP) terminal and via TSO terminals. The developers use the IBM S/360-75 primarily in programmer-present blocks of time for integration and system testing via a graphics device. The IBM S/360-95 is the primary day-to-day satellite operations machine. When a hardware failure occurs, the developers lose access to the machine via the RJP and TSO terminals and must immediately relinquish their programmer-present time (if they have it) on the IBM S/360-75 so that operations activities can continue with minimal interruption. Since programmer-present blocktime is scheduled weekly and since the schedule is usually fully booked, IBM S/360-95 hardware failures always affect the development schedule adversely, especially late in the development life cycle.

In addition, the IBM S/360-75 is the primary satellite launch and launch-simulation operations machine. It is not unusual to have launches monthly, and frequently they are delayed on a day-by-day basis for 1 to 2 weeks or on a week-by-week basis for 2 to 4 weeks. When this happens, additional simulations are scheduled and/or additional mission planning machine time is required. Again, the developers must relinquish scheduled programmer-present blocktimes.

We estimate that 20 to 40 percent of scheduled programmer-present blocktime is lost because of hardware failures on both machines and because of launch delays. When frequent hardware failures and launches occur during the later stages of a development project, you can see how they can contribute significantly to the peak effort at the start of acceptance testing because of the need to make up lost machine time to complete the development project on schedule.

On the average, the development process takes 15.6 months, requires 8 staff-years of effort, develops 57,000 lines of

code, and delivers 62,000 lines of code. Some amount of old code is used in each of these projects. The average staff size is 5.4 people and peaks at 10 people (full-time equivalents). Fourteen individuals are usually involved; this figure includes the first-line managers, i.e., those managers who make decisions, set priorities, and solve problems on a daily basis. For this application, on the average, the managers have 5.8 years of experience and the technical staff has 4 years. The technical staff includes the managers (approximately 30 percent). The managers have 10 years of professional experience overall, and the technical staff has 8.5 years of professional experience.

V&I EXPERIMENT

DOES INDEPENDENT V&I IMPROVE DEVELOPMENT PROCESS AND PRODUCT?

<u>EXPECTATION</u>	<u>MEASURE</u>
DECREASE	REQUIREMENTS AMBIGUITIES AND MISINTERPRETATIONS
DECREASE	DESIGN FLAWS
DECREASE	COST OF CORRECTING FAULTS
DECREASE	COST OF SYSTEM AND ACCEPTANCE TESTING
INCREASE	EARLY DISCOVERY OF FAULTS
INCREASE	QUALITY OF SOFTWARE PUT INTO OPERATION
MAINTAIN	PRODUCTIVITY/COST

Viewgraph 5: V&I Experiment

Why use a V&I methodology? It has often been claimed that the use of a V&I team would solve some of our problems.

What we want to know from this experiment is "Does the use of an independent V&I team improve our development process and product?" To test this hypothesis, we will apply seven measures. These measures, however, are not completely independent of each other. They measure, in different ways, the occurrence of two basic properties:

1. When errors are discovered earlier, they are less costly to correct.
2. The use of a V&I methodology helps to discover errors earlier.

The seven measures with explanations follow.

1. Decrease requirements ambiguities and misinterpretations. This will save time and money, especially in later stages of development. Overall, these are the most expensive errors to correct because requirements are the starting point for the development life cycle.

To evaluate this measure, the development error data that is collected by the SEL from the development and V&I teams from the start of implementation through the completion of acceptance testing will be examined. In this experiment, the use of a V&I methodology is not expected to reduce the development error rate; rather, it is expected to help discover errors earlier. If the use of a V&I methodology provides this benefit, a larger fraction of requirements errors will be detected during the design phase, in which the SEL has no formal process for recording errors, and therefore, fewer requirements errors (a smaller percentage

of total errors) will remain to be discovered during the formal reporting period.¹ Compared with the past projects, a 50-percent decrease in the percentage of requirements errors reported by the development and V&I teams will be a clear indication of success for this measure. In addition, since the V&I team will pursue the resolution of unspecified and ambiguous requirements, fewer of these requirements problems are expected in the later stages of development.

2. Decrease design errors. This will save time and money in later stages of development. Design errors are the second most expensive to correct.

To evaluate this measure, the development error data will be used to compute the percentage of the design errors that are complex design errors. Complex design errors are many-component errors, whereas simple design errors are single-component errors. A component is a subroutine or shared block of code. Simple design errors are frequently related to (1) wrong assumptions about data values and structures, e.g., integer versus real variables, 2-byte versus 4-byte variables, location in buffer, or length of a format; (2) lapses in memory, e.g., missing items (declarations, dimensions, subscripts, statements, or counter incrementers) or incorrect variable names (not misspellings); or (3) incorrect interpretation of computations, e.g., wrong sense of direction (sign operator), factors of 2 or root 2, or wrong order of steps. Complex design errors are frequently

¹Formal error reporting for development is keyed to machine-readable code that, in this environment, is the executable source code. Therefore, formal error reporting occurs only from the start of implementation through the completion of acceptance testing. Maintenance error data is collected from the maintenance group in a slightly different form.

related to interfaces and operational considerations and, therefore, they affect modules (several components). Since interfaces and operational aspects receive more scrutiny and high-level attention, they are more likely to be discovered during design reviews, which for the most part occur outside the formal error reporting period. The simple design errors, which are found in the detail of the design, are less likely to be found by a small V&I team (approximately 15 percent of development effort). If the use of a V&I methodology helps to discover complex design errors earlier, a larger fraction of the complex errors will be detected during the design phase, and therefore, fewer complex design errors (a smaller percentage) will remain to be discovered during the formal reporting period. Compared with the past projects, a 50-percent decrease in the percentage of complex design errors reported by the development and V&I teams will be a clear indication of success for this measure.

3. Decrease the cost of correcting errors. According to those who advocate the use of a V&I methodology and from our own SEL data, we know that correcting errors one life cycle phase earlier will produce a significant savings.

To evaluate this measure, the relative cost of correcting errors before and after acceptance testing started will be computed.¹ If the use of a V&I methodology reduces the cost of correcting errors, the developers will spend less effort per error in the later stages of development. Compared with the past projects, a 20- to 25-percent reduction

¹Here, the relative cost of correcting errors is computed by tabulating the effort to correct errors (reported by the development teams) in each phase, computing the percentage of error-correction effort that occurred in each phase, and then dividing the error-correction effort percentage of each phase by the corresponding percentage of errors found in that phase.

in the relative cost of correcting errors after acceptance testing started will be a positive indication of success for this measure. Maintenance error data that is collected by the SEL from the maintenance groups will also be used.

4. Decrease the cost of system and acceptance testing. If the first three items occur, less effort will be required in these phases.

To evaluate this measure, the percentage of the development cost required to complete system and acceptance testing will be computed.¹ If the use of a V&I methodology helps to discover errors closer to the phase in which they originated, (1) the development teams will spend less time correcting errors during system testing and the system tests will be completed sooner, reducing the cost of system testing and (2) the development teams will need only to prepare for and to demonstrate the acceptance tests, reducing the cost of acceptance testing. Compared with the past projects, a smaller percentage of development cost for system and acceptance testing will be a positive indication of success for this measure. If the cost is less than the average cost for this application, it will be a clear indication of success.

5. Increase the early discovery of errors. This will save time and money in later stages of development as stated above. It will also improve the reliability of the software or at least improve confidence in the reliability of the software, since error rates will be less (or the mean time

¹The development cost is computed by weighting the hours charged to a project by the different responsibilities of the personnel assigned to the project. A manager's hours are multiplied by 1.5; a programmer's hours are multiplied by 1.0; support service personnel's hours are multiplied by 0.5.

between failures will be greater) in the later stages of development. To evaluate this measure, the development and maintenance error data will be used to compute the percentages of errors that were discovered before and after acceptance testing started. If the use of a V&I methodology helps to discover errors earlier, most of the errors will be discovered before acceptance testing starts. Compared with the past projects, a 50-percent reduction in the percentage of errors discovered after acceptance testing started will be a clear indication of success for this measure.

6. Improve the quality of the software put into operation. This will decrease maintenance costs. In general, the use of a V&I methodology will be most beneficial in the M&O phase, since systems with lifetimes greater than 1 or 2 years usually have maintenance costs that range from 30 to 100 percent of the development cost.

To evaluate this measure, the software and maintenance error data will be used to compute the error rate for the M&O phase. If the use of a V&I methodology improves the quality of the software put into operation, the error rate in the M&O phase will be smaller compared with the error rates of the past projects. An error rate less than the average error rate (0.5 to 0.6 errors per thousand lines of developed code) for this application will be a positive indication of success for this measure.

7. Maintain productivity and cost. Adding another interaction for the development team will slow them down and will, therefore, reduce their productivity and increase the cost of development. However, if requirements and complex design errors are reduced, if the cost of correcting errors is reduced, and if the time spent on system and acceptance testing is reduced, those reductions should offset the cost of interaction between the development and V&I teams.

Therefore, productivity and development costs should remain the same. We do not expect to offset the cost of the V&I team completely, but optimistically speaking, we hope to.

To evaluate this measure, the software and the weighted work hours charged to the projects by the development teams will be used to compute (in staff-months) the cost of 1000 lines of developed code. A cost less than or equal to the average cost (1.7 staff-months per thousand lines of developed code) for this application will be a clear indication of success for this measure. That is to say, an average cost for the development team plus an added cost for the V&I team is a clear indication of success; the development teams will have maintained productivity despite the interaction with the V&I team.

By one calculation, the cost of interaction with the V&I team is estimated to be 10 percent of the development effort. Therefore, if the development teams are average in performance and require only the average cost even though they are interacting with a V&I team, the use of a V&I methodology will have effected approximately a 10-percent savings in development cost. If the use of a V&I methodology works well, i.e., if the first six measures show positive indications of success, then the combined cost of the development and V&I teams will be close to the average cost of development for this application. Since the cost of the V&I effort will be approximately 15 percent of the development effort and the estimated cost of interaction with the V&I teams is 10 percent, a combined cost of the development and V&I teams that is near the average development cost will indicate approximately a 25-percent savings in development cost (15 percent real savings).

V&I TEAM¹

CHARTER:

VERIFY REQUIREMENTS AND DESIGN
PERFORM SEPARATE SYSTEM TESTING
VALIDATE CONSISTENCY END TO END
FIX NOTHING
REPORT ALL

PROCESS CHARACTERISTICS:

DURATION (MONTHS)	14-16
EFFORT	15-18 PERCENT OF DEVELOPMENT EFFORT
STAFF (FULL-TIME EQUIV.)	
AVERAGE	1.1
PEAK	3.0
INDIVIDUALS	6
APPLICATION EXPERIENCE	
MANAGERS	7
TECHNICAL STAFF	4
OVERALL EXPERIENCE	
MANAGERS	14
TECHNICAL STAFF	8

¹SAME CONTRACTOR AS DEVELOPMENT TEAMS, BUT IN DIFFERENT OPERATIONAL AREA.

Viewgraph 6: V&I Team

What did we expect the V&I team to do in this experiment?

The V&I team was supposed to

- Verify requirements and design.
- Perform separate system testing
- Validate the consistency from start to end (from requirements to product)
- Fix nothing
- Report all findings

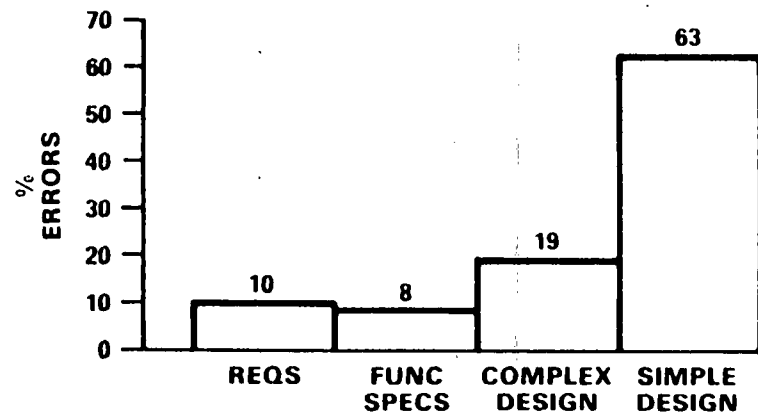
The V&I process lasted 14 to 16 months and required an effort of 16 to 18 percent of the development effort. The process required an average of 1.1 people and peaked at 3 people (full-time equivalents). Six individuals were involved, including the first-line managers. The application and overall experience of the technical staff was similar to that of the development teams (viewgraph 4); the managers, however, had a little more experience.

The V&I team was associated with the same contractor as the development teams but came from a different operational area.

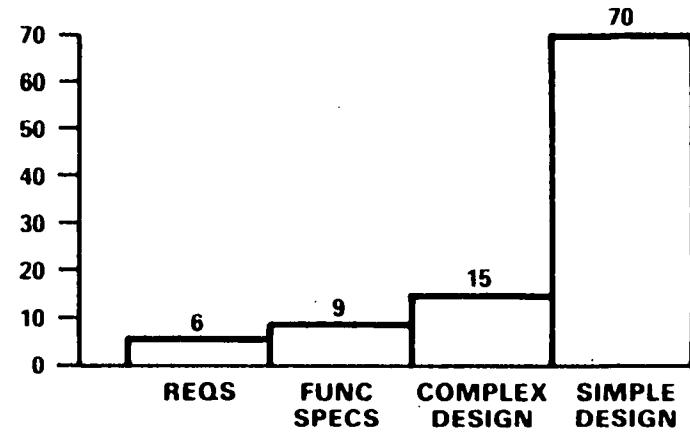
Next, we will examine the results of the experiment.

MEASURE 1—REQUIREMENTS PROBLEMS

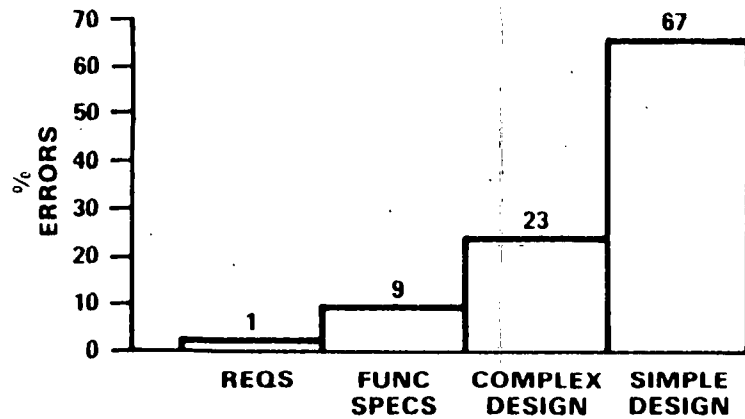
MEASURE 2—DESIGN FLAWS



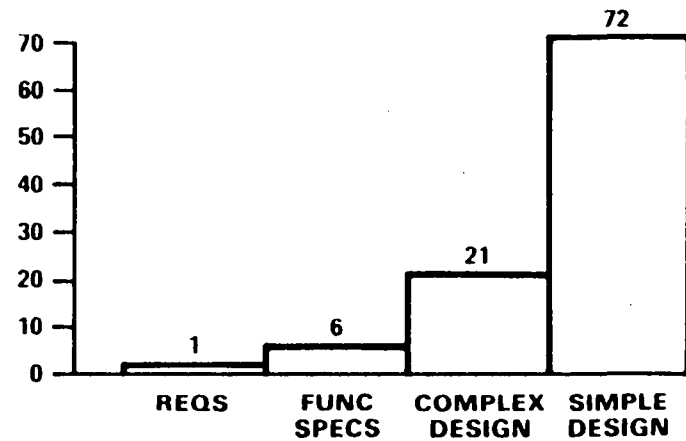
PAST 1



PAST 2



V&I 1



V&I 2

Viewgraph 7: Measure 1 - Requirements Problems and
Measure 2 - Design Flaws

This viewgraph shows the breakdown, by percentages, of all the requirements and design errors detected from the start of implementation through the end of acceptance testing.

1. Requirements Errors

Expectation:

For requirements errors, we expect to see a 50-percent decrease in the percentage of requirements errors.

Findings:

From the bar graphs, you can see that the percentage of requirements errors for both V&I projects was reduced 84 to 90 percent compared with the past projects. In addition, very few requirements remained unspecified in the later stages of development. Hence, there were very few late surprises in terms of requirements problems compared with the past projects.

Conclusion:

The use of a V&I methodology did significantly decrease requirements ambiguities and misinterpretations.

2. Design Errors

Expectation:

For design errors, we expect to see a 50-percent decrease in the percentage of complex design errors. Complex design errors are those involving many components. Simple design errors are single-component errors. A component is a subroutine or a shared block of code.

Findings:

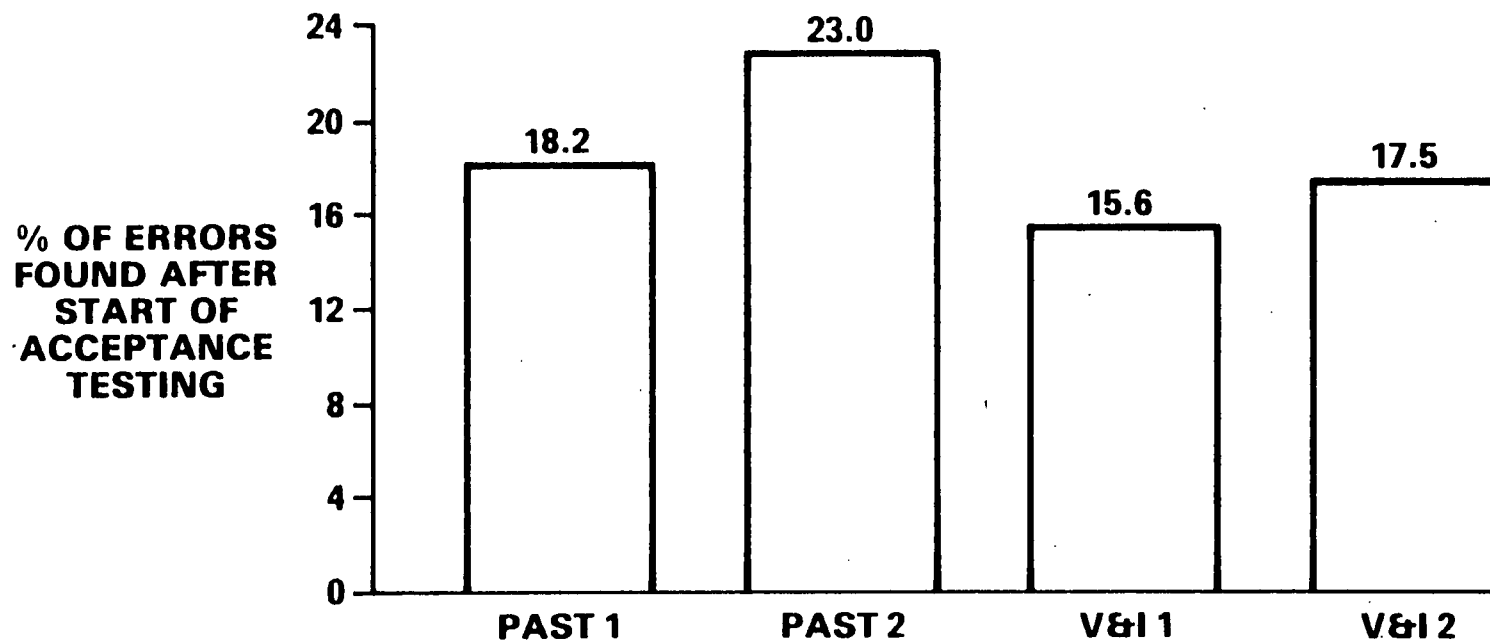
From the bar graphs, you can see that the percentages of complex design errors for the V&I projects are 26 and

23 percent of the total design errors. It is a little less for the two past projects (23 and 18 percent).

Conclusion:

The use of a V&I methodology did not decrease complex design errors.

MEASURE 5—EARLY DISCOVERY OF FAULTS



Viewgraph 8: Measure 5 - Early Discovery of Faults

This viewgraph shows the percentage of errors of the total that were found after acceptance testing started.

Expectation:

We expect to see a 50-percent reduction in the percentage of errors found after acceptance testing starts.

Findings:

You can see that for the two V&I projects there was a slight decrease (less than 30 percent) in the percentage of errors found after acceptance testing started.

Conclusion:

The use of a V&I methodology did not significantly increase the early discovery of errors.

Additional Data:

The percentage of errors found in each phase is as follows:

<u>Phase</u>	<u>Past 1</u>	<u>Past 2</u>	<u>V&I 1</u>	<u>V&I 2</u>
After Acceptance Testing Started	18.2	23.0	15.6	17.5
Before Acceptance Testing Started	81.8	77.0	84.4	82.5
Maintenance and Operation	3.4	5.3	5.0	6.9
Acceptance Testing	14.8	17.7	10.6	10.6
System Testing	14.8	4.8	8.2	18.9
Code/Unit Testing	67.0	72.2	76.2	63.6

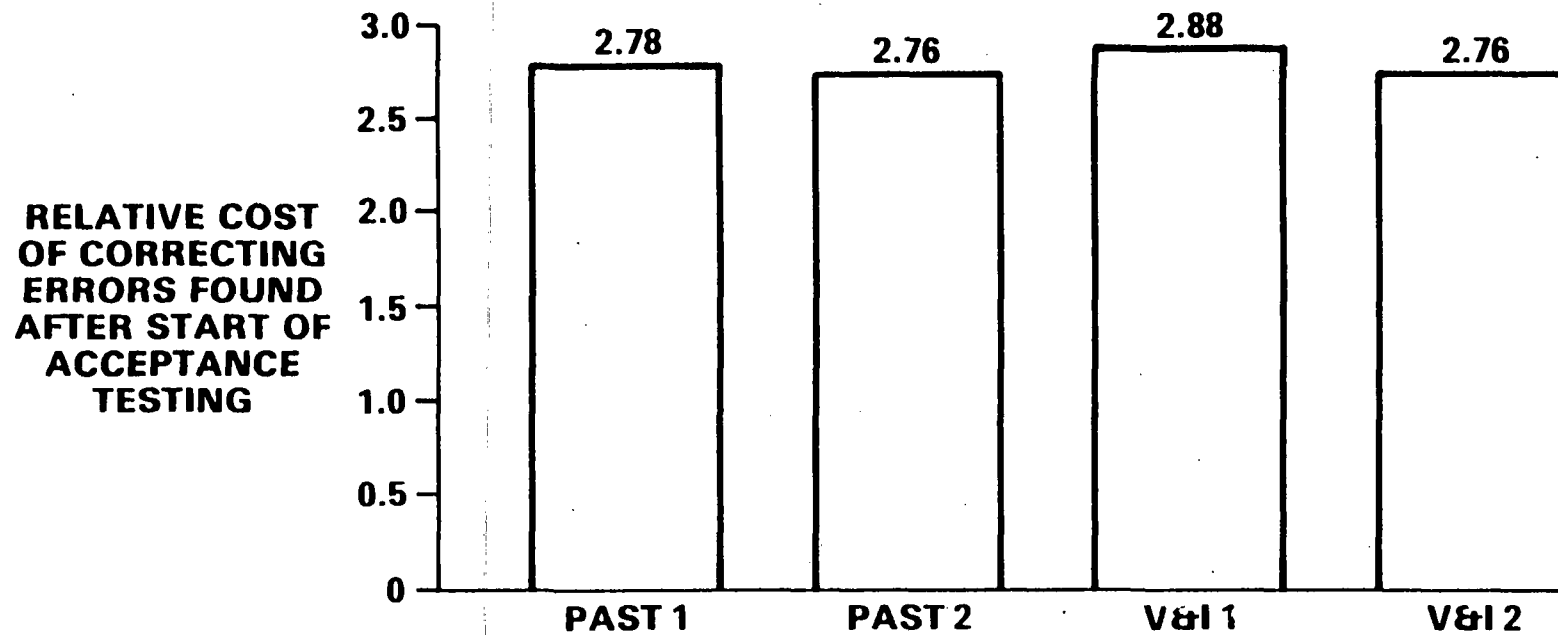
This viewgraph and viewgraphs 9 through 11 contain M&O data through November 20, 1981. The length and status of the M&O phases are as follows:

<u>M&O Phase</u>	<u>Past 1</u>	<u>Past 2</u>	<u>V&I 1</u>	<u>V&I 2</u>
Months	25	14	5	6
Status	Active	Complete	Active	Active

Except for project Past 2, which has ended, the results presented in viewgraphs 8 through 11 can only become worse with further operation. However, the results are not expected to change appreciably because of the characteristics of the environment. Typically, in this environment, 95 to 100 percent of the postacceptance error corrections and enhancements occur during the first 6 months of M&O. For example, the supposedly last-planned modification of the source code for both V&I projects occurred a few days before November 20, 1981.

After the first 6 months of M&O, typically, the software is changed only to support a degradation in satellite hardware performance, e.g., failure of a primary sensor. However, to support a launch, the software is engineered to support these types of contingencies but not always accurately enough for day-to-day operation. Since the usual lifetimes of these projects range from 1 to 3 years, the users must weigh the cost of extensive development to support serious or critical degradation in satellite hardware performance with the benefit to be gained during the expected (and usually shortened) life of the satellite. For example, about a year ago, the satellite of project Past 1 (25 months M&O) had a critical hardware failure that seemed to end the project prematurely; however, relatively simple modifications to the software allowed the users to keep the satellite active in a degraded mode of operation.

MEASURE 3—COST OF CORRECTING FLAWS



Viewgraph 9: Measure 3 - Cost of Correcting Flaws

This viewgraph shows the relative cost of correcting errors found after acceptance testing started. This number is the ratio of the fraction of effort required to correct the errors that occurred after acceptance testing started to the fraction of errors that occurred after acceptance testing started. For example, if 50 percent of the effort to correct errors was expended after acceptance testing started and if that effort was needed to correct 5 percent of the errors, this number would be 10.

Expectation:

We expect to see a 20- to 25-percent lower relative cost to correct errors after acceptance testing starts.

Findings:

From the bar graphs, you can see that the relative cost to correct errors after acceptance testing started was the same as that for the past projects. The relative cost to correct errors before acceptance testing started was approximately 0.5. This indicates that the cost to correct errors after acceptance testing started was between 4.4 and 4.9 times more costly than the cost to correct errors before acceptance testing started.

Conclusion:

The use of a V&I methodology did not decrease the cost of correcting errors in the acceptance testing and M&O phases combined.

Additional Data:

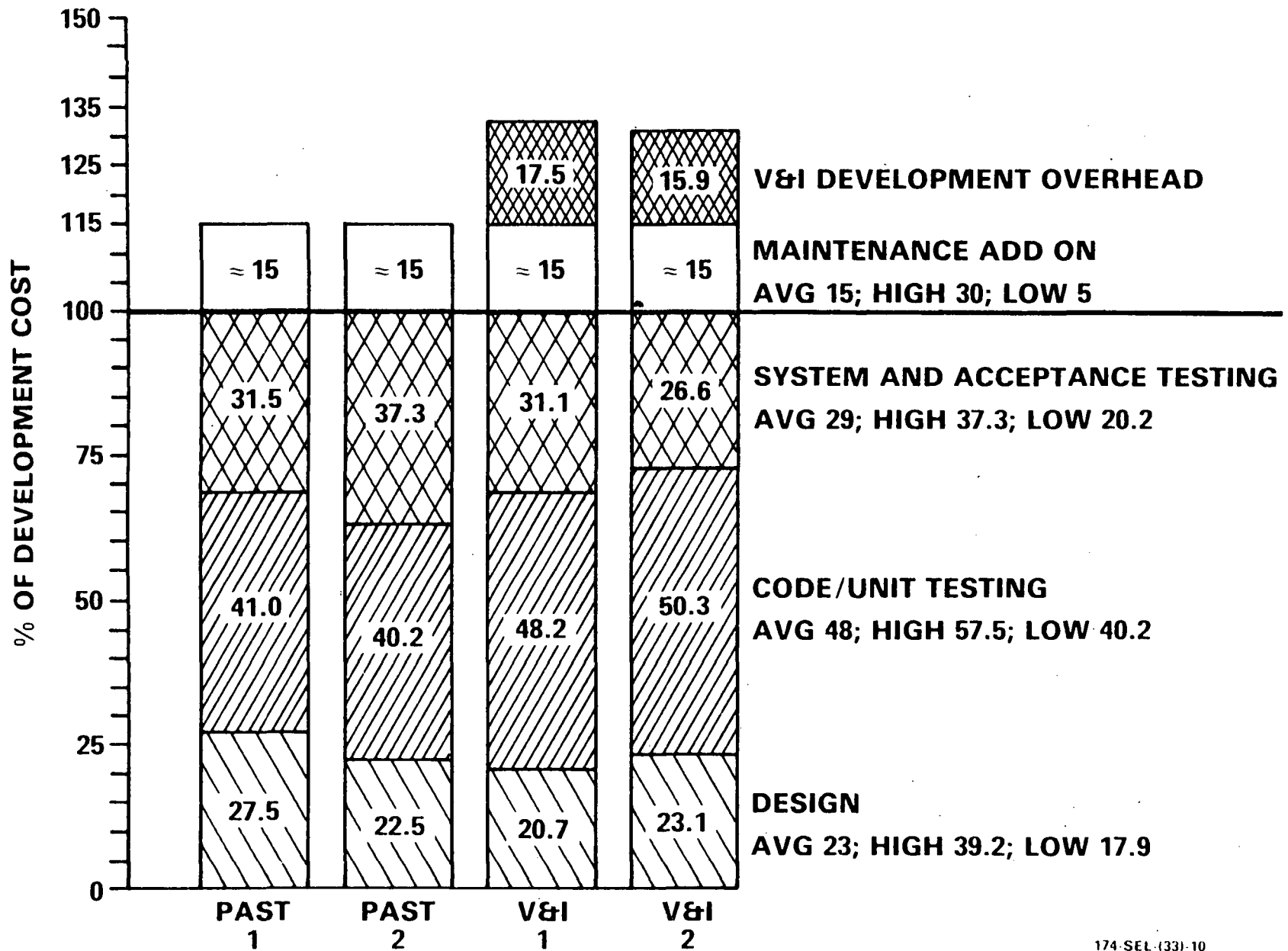
The relative cost of correcting errors in each phase is as follows:

<u>Phase</u>	<u>Past 1</u>	<u>Past 2</u>	<u>V&I '1</u>	<u>V&I '2</u>
After Acceptance Testing Started	2.78	2.76	2.88	2.76
Before Acceptance Testing Started	0.60	0.47	0.59	0.63
Maintenance and Operation	4.85	4.53	4.09	3.54
Acceptance Testing	2.31	2.23	2.31	2.26
System Testing	1.00	1.09	1.30	1.08
Code/Unit Testing	0.47	0.43	0.58	0.49

These figures, in part, validate the common belief (advanced by proponents of V&I methodology) that errors are more expensive to correct when they are discovered later in the development cycle. You can also see from these figures and from the figures in the previous viewgraph that the results are different for different phases; but, remember that we do not have responsibility for the maintenance phase, and data is more difficult to obtain from the group who has responsibility. Therefore, we measure things one or two phases earlier, i.e., during acceptance testing or system testing.

The relative cost of correcting errors in the M&O phase was less for the V&I projects mainly because of fewer requirements errors in that phase. The past projects had at least twice as many requirements errors in that phase.

MEASURE 4—COST OF SYSTEM AND ACCEPTANCE TESTING



Viewgraph 10: Measure 4 - Cost of System and Acceptance Testing

This viewgraph shows the cost for time spent in various development calendar phases (not activity phases). Design activity takes place in the design calendar phase, in the code/unit testing (implementation) calendar phase, and even in the system and acceptance testing calendar phase. Detailed SEL data shows that design activity ranges from 30 to 45 percent of the development effort. On the average, however, only 23 percent of the development effort occurs during the design calendar phase, i.e., the phase in which only design-related activity is performed. The remaining design activity is performed primarily during the implementation phase because requirements change, previously missing information is acquired, and design errors exist. Since it is not unusual to receive requirements changes during the system and acceptance testing phases, since some previously missing information may be acquired during these phases, and since design errors are also discovered in these phases, some design activity occurs here, too.

This viewgraph also contains the average cost for each phase and the highest and lowest cost for each phase for the 12 projects in our sample. The high or low costs themselves do not represent the cost of one project but show the most and least money spent for the various phases by any of the 12 projects.

Expectation:

We expect to see a reduction in the cost of the system testing and acceptance testing phases.

Findings:

On the average, we spend 29 percent of our dollars on system and acceptance testing. You can see that one V&I project was below the average (26.6 percent) and the other, above

(31.1 percent). Together, they were equal to the average. Both were less than our two projects from the past.

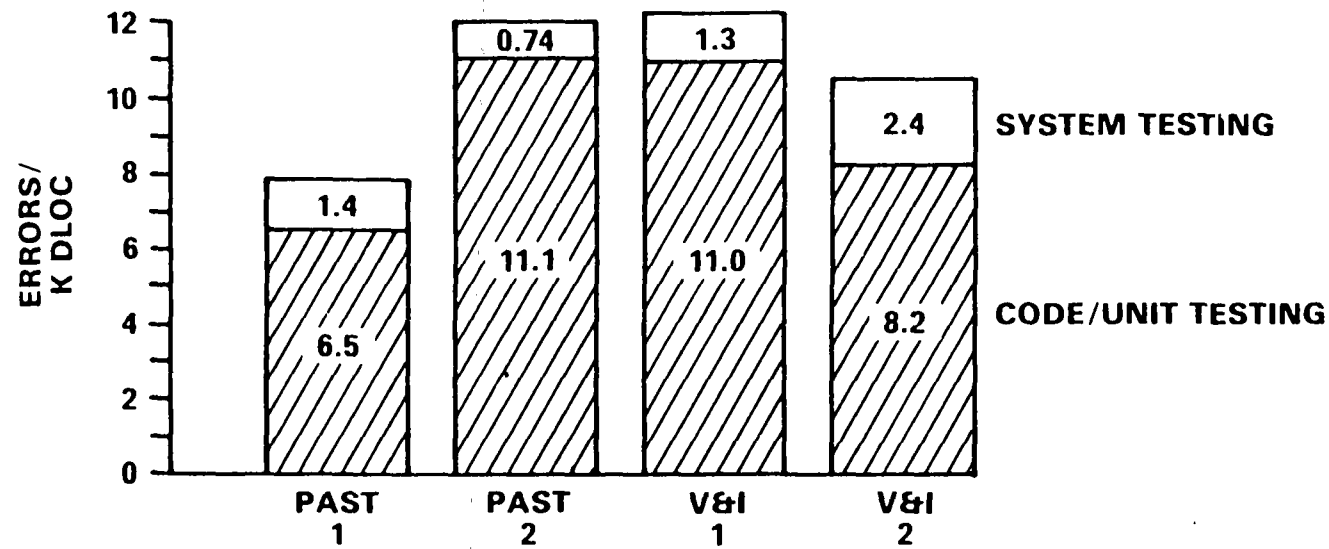
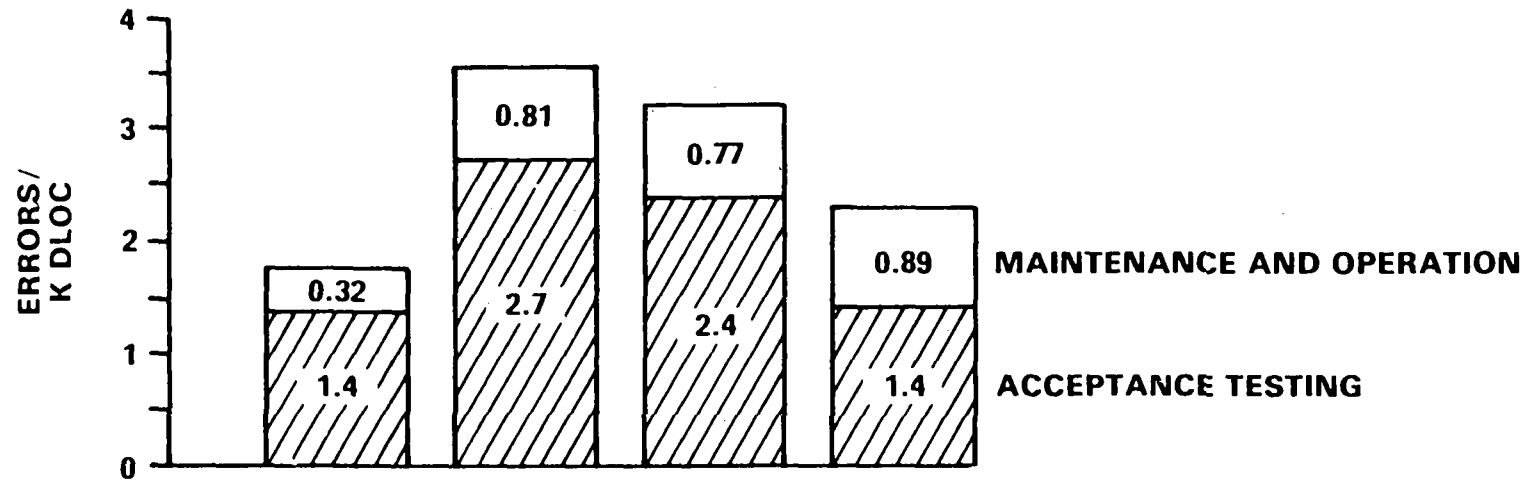
Conclusion:

The use of a V&I methodology did not significantly decrease the cost of system and acceptance testing.

Additional Data:

We do not have responsibility for the maintenance phase. Our best estimate is that the maintenance costs for the four projects are about 15 percent of the development costs. The V&I projects had approximately 16- to 18-percent overheads to pay for the V&I effort.

MEASURE 6—QUALITY OF SOFTWARE



**MONTHS
MAINTENANCE
OPERATION**

25
21

14
12

5
3

6
3

Viewgraph 11: Measure 6 - Quality of Software

This viewgraph shows the errors per thousand lines of developed code for various calendar phases. What is important here is the M&O phase.

Expectation:

We expect to see an error rate in the M&O phase less than the average error rate for this application.

Findings:

From the bar graphs, you can see that the error rates for the two V&I projects are not better than the error rates for the two past projects. The average error rate in the M&O phase is between 0.5 and 0.6 errors per thousand lines of developed code; both V&I projects had error rates higher than the average.

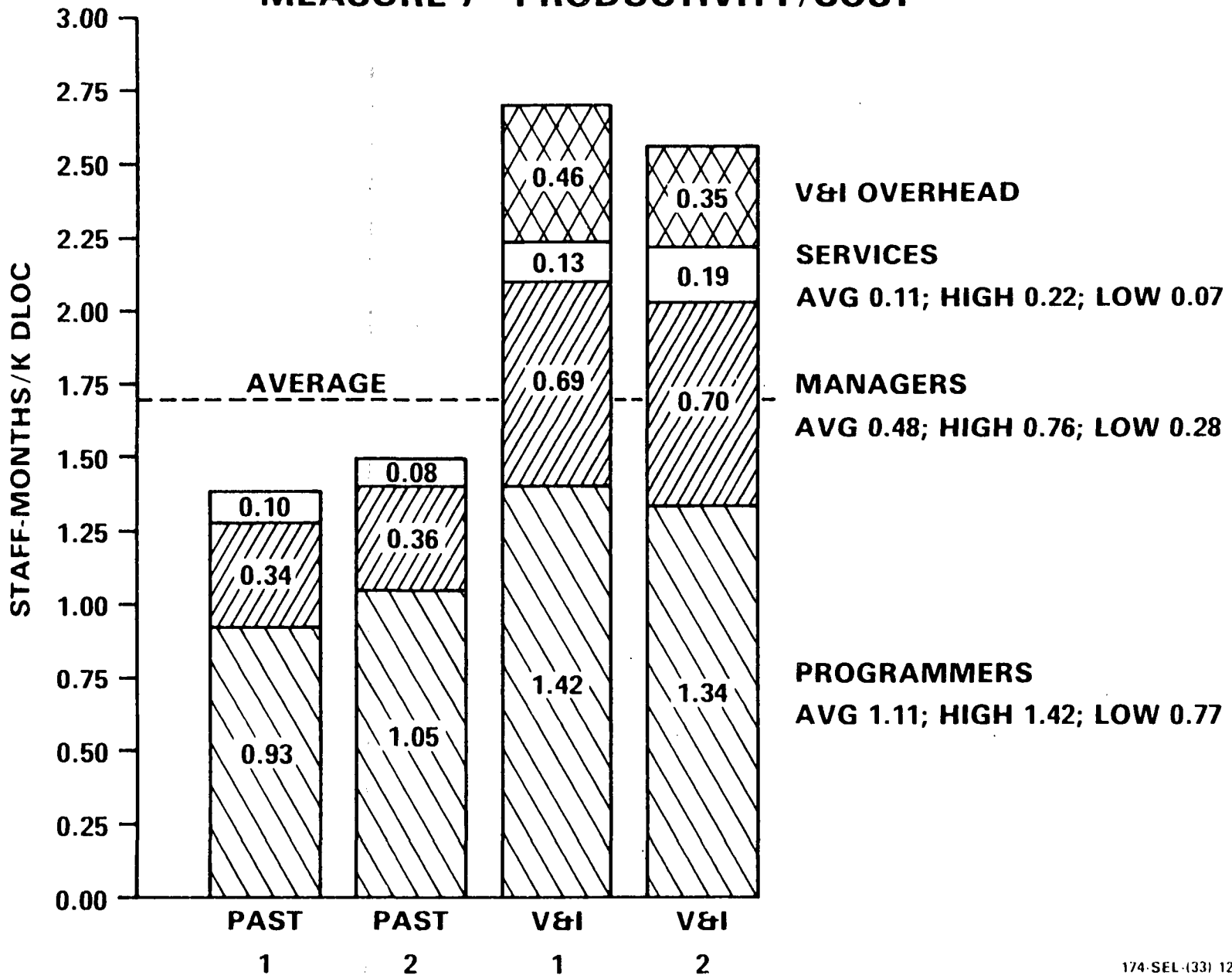
Conclusion:

The use of a V&I methodology did not improve the quality of the software put into operation.

Additional Data:

Error rates from the other phases are important track records. Hypothetically, let us say that projects Past 1 and V&I 2 were developing the same product. If we measured the acceptance testing error rates, we would see that both had error rates of 1.4 errors per thousand lines of developed code. We would not be able to tell too much about the projects from that viewpoint. However, if we examined those projects' error rates before acceptance testing, we would see that project Past 1 had a preacceptance testing error rate of 7.9 and project V&I 2 had a preacceptance testing error rate of 10.6. From this, we may be able to predict the worse M&O phase error rate for project V&I 2.

MEASURE 7—PRODUCTIVITY/COST



Viewgraph 12: Measure 7 - Productivity/Cost

This viewgraph shows the cost (in staff-months) per thousand lines of developed code (K DLOC).

Expectation:

We expect the V&I overhead costs to be an add-on cost to our average development cost.

Findings:

Because of the interaction with the V&I team and some other problems, we drove the productivity of the development teams to the low end of our productivity range. Together, the two V&I projects were about 85 percent more expensive than our two past projects. Since the quality of the products was not any better (see viewgraph 11), an 85-percent increase in cost for the same product is a very expensive penalty to pay. The cost of the development part of the V&I projects (2.2 staff-months per K DLOC) was approximately 30 percent higher than the average development cost (1.7 staff-months per K DLOC). This is three times as large as the estimated cost of interaction with the V&I team.

Conclusion:

The use of a V&I methodology is expensive.

RESULTS OF V&I EXPERIMENT

FROM THE DATA WE HAVE USED, WE HAVE

<u>FOUND</u>	<u>MEASURE</u>
LARGE DECREASE IN	REQUIREMENTS AMBIGUITIES AND MISINTERPRETATIONS
NO DECREASE IN	DESIGN FLAWS
NO DECREASE IN	COST OF CORRECTING FLAWS
SMALL DECREASE IN	COST OF SYSTEM AND ACCEPTANCE TESTING
SMALL INCREASE IN	EARLY DISCOVERY OF FAULTS
NO INCREASE IN	QUALITY OF SOFTWARE PUT INTO OPERATION
LARGE <u>DECREASE IN</u> <u>INCREASE IN</u>	PRODUCTIVITY COST

SCORE: 1 PLUS; 5 ZEROS; 1 DOUBLE MINUS

Viewgraph 13: Results of V&I Experiment

From the data we have used, which includes resource data, error data, and the software, we have found that a V&I methodology provided

1. A large decrease in requirements ambiguities and misinterpretations. There were very few late surprises in terms of requirements problems, and the number of requirements errors reported was significantly less than for the past projects.

2. No decrease in design errors. The fraction of complex design errors was similar to that of the past projects.

3. No decrease in the cost of correcting errors. The relative cost of correcting errors that occurred after acceptance testing started was the same as that for the past projects.

4. A small decrease in the cost of system and acceptance testing. One V&I project had a system and acceptance testing cost less than the average system and acceptance testing cost; the other V&I project was above the average cost. However, both V&I projects had costs below the costs of the past projects used in the comparison.

5. A small increase in early discovery of errors. For both V&I projects, the percentage of errors that occurred after acceptance testing started was less than the percentage of errors that occurred after acceptance testing started for the past projects.

6. No improvement in the quality of software put into operation. The error rates in the M&O phase for both V&I projects were higher than the average error rate for software put into operation for this class of application.

7. A decrease in productivity and an increase in cost. Because, in part, the interaction of the V&I and

development teams lowered productivity and because there was not a savings in correcting errors, the cost was high.

We scored a plus with the first measure (requirements problems); zero with the next five measures; and a double minus with the last measure (productivity/cost).

SUMMARY

- **FIRST APPLICATION OF V&I IN THIS ENVIRONMENT**
 - DID NOT IMPROVE PROCESS
 - WAS EXPENSIVE
 - WAS A MANAGEMENT HEADACHE

- **HOWEVER, WITH VARIATIONS, WE WILL ENCOURAGE ITS USE FOR**
 - THE RIGHT SIZE EFFORT
 - THE RIGHT RELIABILITY REQUIREMENT

Viewgraph 14: Summary

For our first application of a V&I methodology in this environment

- V&I did not improve the process
- V&I was very expensive
- V&I was a management headache

To qualify this, our experience with many methodologies has been as follows:

- The first time a methodology is applied, mistakes are made (and we made many mistakes), and many of the potential benefits or advantages of the methodology are not realized.
- The second time a methodology is applied, there is a tendency to overcompensate for the things that you did worst the first time, and the methodology still does not work as well as it potentially could.
- The third time a methodology is applied, you lower your expectations somewhat or modify them, and you home in on what is right for your environment.

In general, development teams are at the bottom of the totem pole in this environment. Because they work in an operations environment, they have low priority for accessing the machines. They have adversary relationships with the analysis/requirements team, the team that conducts acceptance testing, the people who schedule computer time, the computer operators, the programmer assistance center, and the customer. The V&I team members, who are like a development team but do not design or implement, have the same adversaries. Placing a V&I team in this environment creates another adversary for both the development team and the development-like V&I team. The manager who monitors both teams (the customer) has twice as many complaints, computer

problems, priority decisions, schedule problems, cost problems, reporting problems, and conflicts to deal with. The V&I experiment was a management headache.

However, we believe that we know what changes are needed and how to moderate them to make the use of a V&I methodology more cost effective in this environment for

- The right size effort
- The right reliability requirement

Most of our projects require 8+4 staff-years of effort. We believe that a V&I methodology will be cost effective in the 10- to 12-staff-year range and that cost savings will be achieved for larger efforts. All our completed projects have been for ground-based software, but we have started to develop some onboard (flight) prototype systems. For these systems, which have a more stringent reliability requirement, we believe that a V&I methodology will be cost effective for 5- to 6-staff-year efforts. In both these cases, we believe that a V&I effort of approximately 15 percent of the development effort is sufficient for our work.

THE VIEWGRAPH MATERIALS
for the
J. PAGE PRESENTATION WERE
INCORPORATED IN PAPER

EVALUATING SOFTWARE DEVELOPMENT CHARACTERISTICS:
ASSESSMENT OF SOFTWARE MEASURES IN THE
SOFTWARE ENGINEERING LABORATORY

Victor R. Basili
University of Maryland
College Park, MD 20742

The purpose of this presentation is to discuss some of the work done on metrics in the Software Engineering Laboratory. To put things in perspective, there are many factors that affect software quality and each of these factors has several criteria which define it. Metrics represent some sort of measurement as to whether or not we have achieved a particular criteria. For example, one factor that we would like the software to possess is reliability. One of the many criteria that goes to make up this generalized factor of reliability might be fault tolerance. One of the metrics that can be used to evaluate fault tolerance might be the number of crashes of the system.

There are many views of metrics. We can think of metrics as being subjective or objective. Subjective metrics normally do not involve any exact measurement; they tend to be an estimate of extent to a degree in the application of some technique or a classification or qualification of a problem or experience. Subjective metrics are usually done on a relative scale; e.g., they may be binary (yes or no), or discrete numbers (zero, 1, 2, 3). Examples of subjective metrics would be a qualitative judgment on the use of Process Design Language or an evaluation of the experience of programmers in a particular application.

Objective metrics, on the other hand, tend to be absolute measures taken on the product or process. For example, the time of development,

the number of lines of code delivered, the productivity in lines of code per staff month, the number of errors or changes associated with the project. The distinction between subjective and objective metrics is typically a little bit fuzzy. Very often we make a metric subjective because we don't know how to quantify it.

Another characterization of metrics is as product or process metrics. Product metrics measure the developed product, such as the source code, the object code, or the documentation. Such metrics might be lines of code (objective metric) or readability of the source code (subjective metric). Process metrics tend to measure the process model used for developing the product. Metrics such as use of methodology (subjective metric) and effort and staff months (objective metric) are two metrics that measure the process.

Another characterization is to think of metrics as being cost or quality metrics. It is clear that cost can be a quality metric. However, typically a goal in software development is to minimize cost and maximize quality. So for that reason we will consider these as separate views. Cost normally involves the expenditure of resources in dollars, which might include some capital investment, and this metric is usually normalized according to some value component. For example, we measure staff months or productivity in terms of dollars received for dollars spent, or output for dollars spent, or size per time slice. Quality metrics, on the other hand, measure some form of the value of the product. For example, trying to measure the mean time to failure of the product, the ease of change, the correctness, or the number of errors remaining are all quality measures.

Use of Metrics

We use metrics in varying ways. We can use them to characterize,

evaluate, or predict. Almost all metrics fit in the characterizing category. In that sense, the metric helps to distinguish the product and process or environment. For example, we may categorize an environment by the use of a methodology, the number of externally-generated changes, or the size. This allows us to compare environments or products or processes.

Not all characterizing metrics are evaluative. Metrics are considered evaluative if the metric correlates with or shows directly the quality of the process or the product. For example, the number of errors recorded during acceptance testing or the productivity involved in the development of a software project give us some way of evaluating whether the product has some reasonable reliability or the development is cost effective.

The most powerful capability a metric can have is prediction; that is, the measure is estimable or calculable and is used to predict another measure. For example, estimating size as a predictor of effort is a way to use an estimable metric to predict some desired information.

To demonstrate that a particular metric evaluates or predicts, requires some validation. Too often metrics are proposed in the literature which are meant to be evaluative or predicted, but that capability is not established by experiment or case study.

Analyzing Objective Metrics in the Software Engineering Laboratory

In a paper presented at the Sigmetrics Workshop (Basili/Phillips), we tried to use the laboratory project data to study the relationship between various metrics of size and complexity. One of the questions raised was could we predict effort, which was a cost measure, and the number of errors, a quality metric, using the various size and complexity metrics that appear in the literature. A second question was to be able to check the internal

consistency of several of those size and complexity metrics. The metrics used are given in Table 1. The relationship between the various complexity metrics appears in Table 2, which gives the Pearson correlation coefficient. As can be seen from this table, several of the complexity and size metrics

OBJECTIVE SIZE AND COMPLEXITY MEASURES STUDIED

SRC : SOURCE LINES OF CODE INCLUDING COMMENTS

XQT : EXECUTABLE STATEMENTS

SOFTWARE SCIENCE METRICS

N : LENGTH IN OPERATORS AND OPERANDS

V : VOLUME

V* : POTENTIAL VOLUME

L : LEVEL

E : EFFORT

CYC : CYCLOMATIC COMPLEXITY

CLS : NUMBER OF CALL STATEMENTS

CAJ : CALLS AND JUMPS

CHG : CHANGES TO THE SOURCE CODE

REV : NUMBER OF REVISIONS (VERSIONS) IN THE LIBRARY

EFF : NUMBER OF HOURS EXPENDED IN DEVELOPMENT

ERR : NUMBER OF ERRORS ASSOCIATED WITH COMPONENT

Table 1

RELATIONSHIP BETWEEN SIZE AND COMPLEXITY METRICS

	REV	CHG	XQT	SRC	CAJ	CYC	CLS
E	.6750	.2407	.8390	.8706	.8742	.8906	.7966
CLS	.6427	.3579	.7594	.8186	.9648	.8651	
CYC	.7921	.2534	.9253	.9519	.9666		
CAJ	.7439	.3158	.8734	.9176			
SRC	.8415	.2942	.9896				
XQT	.8560	.2920					
CHG	.4229						

Table 2

correlate well with one another. On the other hand, the change metrics do not correlate well. In trying to use combinations of these metrics to predict effort and errors, we see by Table 3 that there is some success in accounting for effort with some of the metrics, but less success in accounting for errors.

PREDICTING EFFORT AND ERRORS USING
SIZE AND COMPLEXITY METRICS

	EFF	ERR
EFF		.6346
CLS	.7977	.5704
CYC	.7399	.5592
CAJ	.7957	.5848
SRC	.7583	.5576
XQT	.7400	.5485
REV	.7122	.6734
E	.6612	.5432
CHG	.4799	

Table 3

Another study was to look at the internal validation of some of the metrics. Specifically, the software science metrics were examined to see whether predicted values for some of the metrics and actual values related in some way. Again, Pearson's correlation was used; the results are given in Table 4. One can see from this table that metrics like length, that is, N and \hat{N} , do correlate. There is not a bad relationship between V and V^* , although in the group of metrics, that relationship is probably the worst. It should be noted that projects are broken up into two groups--those of small components which were 50 lines or less, and large components which were more than 50 lines.

Based on this study, we made the following conclusions: First of all, there does exist some relationship between complexity metrics and effort and errors. However, most of the complexity metrics do not do much better at estimation than lines of code or executable statements. On the other hand, many of the metrics related very well with each other, which seems to imply that they really are measuring the same thing. The goal, therefore, should be concentrated on looking at orthogonal metrics. We are currently investigating data metrics in the SEL.

Using Subjective and Objective Metrics to Predict Cost

In a paper presented at the 5th International Conference on Software Engineering (Bailey/Basili), we inverted that experiment by examining the relationship between productivity and various factors. Basically, we used nonparametric statistics. The results were as follows: We found no significant relationship between productivity and size. However, there was a large set of methodology factors that showed varying degrees of positive correlation with productivity. A combined methodology factor that was used to pre-

INTERNAL VALIDATION

SMALL COMPONENTS	50 LINES	(280)
LARGE COMPONENTS	50 LINES	(285)

	LARGE	SMALL
$N \sim \hat{N}$.79	.83
$V \sim V^*$.52	.50
$L \sim \hat{L}$.71	.62
$E \sim \hat{E}$.61	.42

PEARSON CORRELATION

Table 4

dict cost or effort in the cost model showed a significant positive correlation with productivity as might have been expected. In this study, projects with high methodology rating were shown to have come from a different population than those with a low methodology rating. No other factor showed a significant positive correlation with productivity and we were able to show, at least in the SEL environment, that methodology does correlate with productivity and therefore has been an effective approach to software development.

Using Subjective Metrics to Predict Quality

Based on the study to predict productivity but changing the statistical approach to factor analysis, we compressed three sets of metrics into three factors--quality, methodology, and complexity. Methodology and complexity were not significantly correlated in the study. However, quality was significantly correlated with methodology with a correlation (R) of .67 and quality was also significantly correlated with complexity with a correlation (R) of -.64. In both cases, the correlation was less than a .001 significance level.

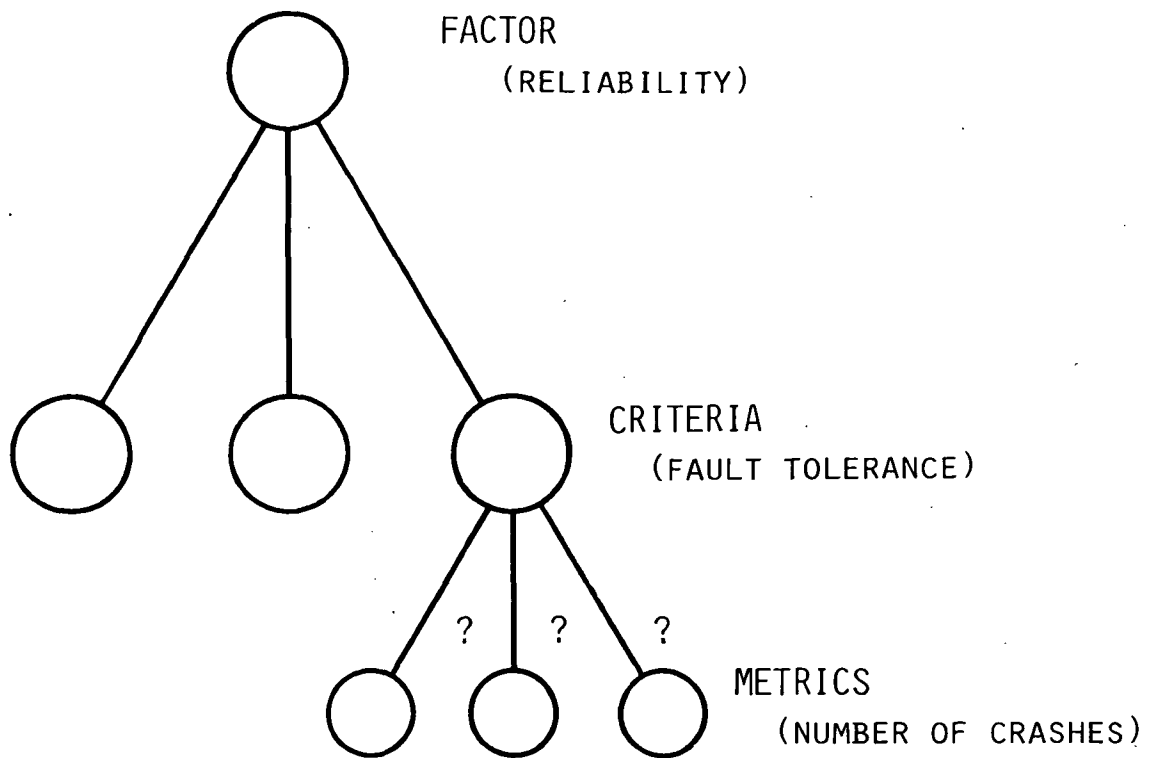
Using methodology alone to predict quality, the coefficient of determination (R^2) is equal to .45. This means that methodology accounted for essentially 45% of the quality rating. Using methodology and complexity both, we got an R^2 of .65. This implies that there is some evidence that we can predict quality from methodology and complexity and that methodology is again highly correlated, not with just productivity as we saw in the previous study, but also with quality. Work in this particular area is just beginning and we plan to make tremendous use of the subjective metrics, not just for evaluation, but also for prediction.

REFERENCES

(Basili/Phillips) - Basili, V. and Phillips, T., "Validating Metrics on Project Data" - Submitted to special issue of Software Metrics, Transactions on Software Engineering.

(Bailey/Basili) - Bailey, J. and Basili, V., "A Meta-Model for Software Development Resource Expenditures," Proceedings of the Fifth International Conference on Software Engineering, March 1981, pp. 107-116

**THE VIEWGRAPH MATERIALS
for the
V. BASILI PRESENTATION FOLLOW**



VIEWS OF METRICS

SUBJECTIVE

VS.

OBJECTIVE

SUBJECTIVE:

NO EXACT MEASUREMENT

AN ESTIMATE OF EXTENT OR DEGREE IN THE APPLICATION
OF SOME TECHNIQUE

A CLASSIFICATION OR QUALIFICATION OF PROBLEM OR
EXPERIENCE

USUALLY DONE ON A RELATIVE SCALE

E.G., USE OF A PDL

EXPERIENCE OF THE PROGRAMMERS IN THE APPLICATION

OBJECTIVE:

AN ABSOLUTE MEASURE TAKEN ON THE PRODUCT OR PROCESS

E.G., TIME FOR DEVELOPMENT

NUMBER OF LINES OF CODE

PRODUCTIVITY

NUMBER OF ERRORS OR CHANGES

VIEWS OF METRICS

PRODUCT

VS.

PROCESS

PRODUCT:

MEASURE OF THE ACTUAL DEVELOPED PRODUCT

I.E., SOURCE CODE, OBJECT CODE, DOCUMENTATION

E.G., LINES OF CODE, READABILITY OF THE SOURCE CODE

PROCESS:

MEASURE OF THE PROCESS MODEL USED FOR DEVELOPING
THE PRODUCT

E.G., USE OF METHODOLOGY, EFFORT IN STAFF MONTHS

- - - - -

COST

VS.

QUALITY

COST:

EXPENDITURE OF RESOURCES IN DOLLARS INCLUDING
CAPITAL INVESTMENT USUALLY NORMALIZED ACCORDING
TO SOME VALUE COMPONENT

E.G., STAFF MONTHS, PRODUCTIVITY, SIZE/TIME SLICE

QUALITY:

SOME FORM OF VALUE OF THE PRODUCT

E.G., RELIABILITY, EASE OF CHANGE, CORRECTNESS,
NUMBER OF ERRORS REMAINING

USE OF METRICS

PREDICTIVE VS. EVALUATIVE VS. CHARACTERIZING

CHARACTERIZING:

MEASURE HELPS DISTINGUISH THE PRODUCT OR PROCESS
OR ENVIRONMENT

E.G., USE OF A METHODOLOGY, NUMBER OF EXTERNALLY
GENERATED CHANGES, SIZE

EVALUATIVE:

MEASURE CORRELATES WITH OR SHOWS DIRECTLY THE QUALITY
OF THE PROCESS OR PRODUCT

E.G., NUMBER OF ERRORS REPORTED DURING ACCEPTANCE
TESTING, PRODUCTIVITY

PREDICTIVE:

MEASURE IS ESTIMATABLE OR CALCULABLE AND IS USED TO
PREDICT ANOTHER MEASURE

E.G., ESTIMATING SIZE AS A PREDICTOR OF EFFORT

USE REQUIRES VALIDATION

ANALYZING OBJECTIVE MEASURES
IN THE SEL

USING SEL PROJECT DATA TO STUDY THE RELATIONSHIP BETWEEN
VARIOUS METRICS OF SIZE AND COMPLEXITY

PREDICTING EFFORT (A COST MEASURE) AND NUMBER OF ERRORS
(A QUALITY METRIC) USING SIZE AND COMPLEXITY METRICS

CHECKING THE INTERNAL CONSISTENCY OF SEVERAL SIZE AND
COMPLEXITY METRICS

OBJECTIVE SIZE AND COMPLEXITY MEASURES STUDIED

SRC : SOURCE LINES OF CODE INCLUDING COMMENTS

XQT : EXECUTABLE STATEMENTS

SOFTWARE SCIENCE METRICS

N : LENGTH IN OPERATORS AND OPERANDS

V : VOLUME

V* : POTENTIAL VOLUME

L : LEVEL

E : EFFORT

CYC : CYCLOMATIC COMPLEXITY

CLS : NUMBER OF CALL STATEMENTS

CAJ : CALLS AND JUMPS

CHG : CHANGES TO THE SOURCE CODE

REV : NUMBER OF REVISIONS (VERSIONS) IN THE LIBRARY

EFF : NUMBER OF HOURS EXPENDED IN DEVELOPMENT

ERR : NUMBER OF ERRORS ASSOCIATED WITH COMPONENT

PREDICTING EFFORT AND ERRORS USING
SIZE AND COMPLEXITY METRICS

	EFF	ERR
EFF		.6346
CLS	.7977	.5704
CYC	.7399	.5592
CAJ	.7957	.5848
SRC	.7583	.5576
XQT	.7400	.5485
REV	.7122	.6734
E	.6612	.5432
CHG	.4799	

RELATIONSHIP BETWEEN SIZE AND COMPLEXITY METRICS

	REV	CHG	XQT	SRC	CAJ	CYC	CLS
E	.6750	.2407	.8390	.8706	.8742	.8906	.7966
CLS	.6427	.3579	.7594	.8186	.9648	.8651	
CYC	.7921	.2534	.9253	.9519	.9666		
CAJ	.7439	.3158	.8734	.9176			
SRC	.8415	.2942	.9896				
XQT	.8560	.2920					
CHG	.4229						

INTERNAL VALIDATION

SMALL COMPONENTS	50 LINES	(280)
LARGE COMPONENTS	50 LINES	(285)

	LARGE	SMALL
$N \sim \hat{N}$.79	.83
$V \sim V^*$.52	.50
$L \sim \hat{L}$.71	.62
$E \sim \hat{E}$.61	.42

PEARSON CORRELATION

CONCLUSION

- **CAN USE COMMERCIALLY-OBTAINED DATA TO VALIDATE COMPLEXITY METRICS**
- **VALIDITY CHECKS AND ACCURACY RATINGS ARE VITAL**
- **THERE EXIST RELATIONSHIPS BETWEEN COMPLEXITY METRICS AND EFFORT AND ERROR COUNTS**
- **THE BETTER THE DATA, THE BETTER THE RESULTS**
- **DON'T DO MUCH BETTER THAN LINES OF CODE ON EXECUTABLE STATEMENTS**
- **METRICS RELATE WELL WITH EACH OTHER
(MEASURING THE SAME THING)**

USING SUBJECTIVE AND OBJECTIVE METRICS
TO PREDICT COST (EFFORT)

A META-MODEL WAS DEVELOPED FOR DERIVING AN INDIVIDUALIZED
COST MODEL FOR THE LOCAL ENVIRONMENT

IT ASSUMES EACH ENVIRONMENT IS DIFFERENT AND IS CLASSIFIABLE
BY A SET OF FACTORS (CAPTURED USING SUBJECTIVE METRICS)

SOME FACTORS ARE CONSTANT ACROSS THE ENVIRONMENT AND ARE
HIDDEN IN A BASIC SIZE/EFFORT EQUATION BASED UPON
PAST HISTORY WITHIN THE ENVIRONMENT

OTHER FACTORS CAUSE DIFFERENCES BETWEEN PROJECTS AND CAN BE
USED TO EXPLAIN THE DIFFERENCE BETWEEN ACTUAL EFFORT
AND EFFORT AS PREDICTED BY THE BASIC SIZE/EFFORT
EQUATION

CAN PREDICT COST (EFFORT) WITH THE USE OF SUBJECTIVE METRICS

EVALUATING THE EFFECT OF VARIOUS FACTORS ON PRODUCTIVITY

WE EXAMINED THE RELATIONSHIP BETWEEN PRODUCTIVITY AND VARIOUS
FACTORS

FOUND NO SIGNIFICANT RELATIONSHIP BETWEEN PRODUCTIVITY AND SIZE

A LARGE SET OF METHODOLOGY FACTORS SHOWED VARYING DEGREES OF
POSITIVE CORRELATION WITH PRODUCTIVITY

A COMBINED METHODOLOGY FACTOR SHOWED A SIGNIFICANT POSITIVE
CORRELATION WITH PRODUCTIVITY

[PROJECTS WITH HIGH METHODOLOGY RATING CAME FROM A DIFFERENT
POPULATION THAN THOSE WITH A LOW METHODOLOGY RATING]

NO OTHER FACTORS SHOWED A SIGNIFICANT POSITIVE CORRELATION
WITH PRODUCTIVITY

METHODOLOGY IS CORRELATED WITH PRODUCTIVITY

USING SUBJECTIVE METRICS TO PREDICT QUALITY

WE COMPRESSED THREE SETS OF METRICS INTO THREE FACTORS:
QUALITY, METHODOLOGY, AND COMPLEXITY

METHODOLOGY AND COMPLEXITY WERE NOT SIGNIFICANTLY
CORRELATED

QUALITY WAS SIGNIFICANTLY CORRELATED WITH
METHODOLOGY ($R = .67$) AND COMPLEXITY ($R = .64$)
AT LESS THAN .001 SIGNIFICANCE LEVEL

USING METHODOLOGY ALONE TO PREDICT QUALITY, $R^2 = .45$

USING METHODOLOGY AND COMPLEXITY WE GET $R^2 = .65$

THERE IS EVIDENCE WE CAN PREDICT QUALITY FROM
METHODOLOGY AND COMPLEXITY

METHODOLOGY IS CORRELATED WITH QUALITY

PANEL #2

SOFTWARE METRICS

J. Gaffney/R. Judge, IBM

J. Post, Boeing Aerospace

D. Card, Computer Sciences Corporation

SOFTWARE METRICS
The Quantitative Impact of Four
Factors on Work Rates Experienced During
Software Development

John E. Gaffney, Jr.

Robert W. Judge

IBM Corporation

Federal Systems Division

Manassas, Virginia 22110

Abstract

This paper describes a model of the software development process which is being used at the IBM, Federal Systems Division. The model considers the software development process to consist of a sequence of activities, such as "program design" and "module development" (or coding). A manpower estimate is made by multiplying code size by the rates (man months per thousand lines of code) for each of the activities relevant to the particular case of interest and summing up the results. The effect of four objectively determinable factors (organization, software product type, computer type, and code type) on the productivity values for each of nine principal software development activities has been assessed. The analysis indicates that four factors can be identified which account for 39% of the observed productivity variation.

Software Cost Analysis By Work Components

Software development costs may be estimated by considering each of the activities or work components that constitute a particular software development process.⁽¹⁾ These components are the basis for a software engineering management model⁽²⁾ used by the Federal Systems Division of IBM. Sixteen work components have been identified from which the software organization or the engineering organization involved in a software development project can structure its particular activities. Data on 9 of them served as the basis for the work reported upon here. This information was based on experience at the IBM, Manassas, Virginia facility. These work components are:

Software Requirements Definition - This work component includes the definition and/or analysis of functional, operational, and other software system requirements.

Software Development Planning - This work component includes all tasks necessary to generate the plans necessary for the implementation of the software system.

Functional Design - This work component covers the documentation of the functions the software must perform to meet the requirements imposed upon it.

Program Design - This work component covers the documentation of the software system from an internal viewpoint.

Module Development - This work component covers the tasks associated with the detailed design of the software modules and their coding and test.

Software Integration and Test (SWIT) - This work component covers the integration and testing of the software system and the analysis to determine if it meets the system requirements.

SWIT Problem Analysis and Error Correction - This work component covers the analysis and correction of software problems uncovered during SWIT.

System Test - This work component covers the hardware/software integration and test effort.

Acceptance Test - This work component covers the demonstration to the customer that the software system satisfies the requirements imposed upon it.

A cost estimate can be made by considering the nature of the particular software development job and the work components (such as program design, coding, etc.) that constitute it. Then, the labor (man months) for each component is estimated. The sum of these man month figures is the amount required for the given job. The labor for each work component is estimated as the product of the productivity rate (in man months per thousand source lines of code = MM/KSLOC) and the amount of source lines of code. Thus;

$$\text{Total labor (man months)} = \sum_{i=1}^n Pe_i \times S = SP_T$$

Where; n = number of work components
 Pe_i = work rate # i
 S = amount of source lines of code (=KSLOC).

The approach to considering the software development process as a sequence of activities with well-ordered time precedence relationships is a model long used by industrial engineers, and has been applied

recently to modern electronic systems development.^(3,4) Considering the development process in terms of its constituents enables the estimator to achieve a greater degree of intellectual control than if he were to evaluate the process overall. For example, it may not be clear how the availability of a new process that facilitates unit testing would impact overall development productivity. However, its effect on the work component that covers unit test would be much easier to discern. Then, the effect on overall productivity can be readily calculated by simply reviewing the appropriate rate (e.g. the proper " Pe_1 " in the equation given above).

The Impact of Four Factors on Work Component Productivities

Earlier work has considered the effect on overall productivity of various factors relating to the complexity of the code to be developed, the skills of the software development work force, and other factors representative of the software development environment.^(5, 6, 7, 8) This paper provides a quantitative assessment of the impact of several significant factors on the work rates of 9 specific work components.

A linear regression model was structured to relate the values of work rate in man months per KSLOC (MM/KSLOC), experienced in a reasonably large number of cases (typically more than 30 data samples), to variables representative of the factors; organization, software product type, computer type, and code type, involved in each case. The multiple correlation coefficient between the MM/KSLOC value and the encoded values of each of the variables was determined in each case. The square of this value times 100 is equal to the amount of variation in the given cost component 'explainable' by these four variables. Table 1 tallies their percentages, together with the sample size for each of the 9 work components that were evaluated.

Table 1 - Percentage of Variation in Work Rate
Explainable by Four Factors (1)

Work Component	Percentage of Variation In Work Rate Explained By The Four Factors (1)	Number of Samples Used
Software Requirements	15.12	30
Software Development		
Plan	17.81	38
Functional Design	15.53	45
Program Design	38.43	66
Module Development	55.87	60
Software Integration		
and Test (SWIT)	46.90	51
SWIT-Problem Analysis		
and Error Correction	60.33	51
System Test	26.13	39
Acceptance Test	49.40	42
Average	36.17	47

(1); organization (2 alternatives); product types (2 alternatives); computer type (3 alternatives); code type (3 alternatives)

Table 1 shows that, on a work component basis, the percentage of variation explained by the four factors is 36.17%. However, on an overall project basis, this percentage increases to 39% value. This is because the percentage of variation explained is larger for those work components which represent a greater proportion of the overall software product development effort.

Conclusion

The methodology of 'bottom-up' or 'micro' software development cost estimation and analysis has been described. The definitions of the sixteen cost components used by the IBM Federal Systems Division were presented. The effects of knowledge of four factors in resolving the uncertainty of nine of these cost components were presented.

Bibliography

1. Cruickshank, R. D., and Lesser, M., "An Approach To Estimating and Controlling Software Development Costs in "The Economics of Information Processing," R. Goldberg and H. Lorin (eds); Wiley, 1981.
2. Quinnan, R. E., "The Management of Software Engineering, Part V," "IBM Systems Journal," Volume 19, No. 4, 1980
3. Maynard, H. B. (ed.), "Industrial Engineering Handbook," McGraw-Hill, 1956.
4. Norden, P. V. , "On the Anatomy of Development Projects," "I.R.E. PGE.M. Transactions," Vol. EM-7; No. 1, pg. 41.
5. Walston, C. E. and Felix, C. P., "A Method of Programming Measurement and Estimation," IBM Systems Journal, Vol. 16, 1977, pg 54-73.
6. Gaffney, Jr., J. E., "Maximize Design Effort and Minimize Program Control Complexity - To Maximize Software Development Productivity," "Proceedings, IEEE Computer Software and Applications Conference," October, 1980, pg 225-228, IEEE catalog # 80CH1607-1.
7. Basili, V. R., and Reiter, R. W., Jr., "An Investigation of Human Factors in Software Development," "IEEE Computer," December 1979, pg 21-38.
8. Brooks, W. D., "Software Technology Payoff; Some Statistical Evidence," IBM Software Engineering Exchange (IBM Federal Systems Division, Bethesda, Md), Volume 2, No. 1, April, 1980.

THE VIEWGRAPH MATERIALS
for the
J. GAFFNEY/R. JUDGE PRESENTATION FOLLOW

THE QUANTITATIVE IMPACT OF FOUR FACTORS ON
WORK RATES EXPERIENCED DURING SOFTWARE DEVELOPMENT

J. E. GAFFNEY, JR. R. W. JUDGE

IBM CORPORATION
MANASSAS, VIRGINIA

SIXTH ANNUAL SOFTWARE ENGINEERING WORKSHOP
NASA
GODDARD SPACE FLIGHT CENTER

DECEMBER 2, 1981

WORK RATE

- o WORK RATE IS AN INDICATOR OF PRODUCTIVITY WHICH USES SOURCE LINES OF CODE (SLOC) AS THE MEASURABLE.

$$\text{LABOR (MAN MONTHS)} = \text{WORK RATE (MM/SLOC)} \bullet \text{WORK(SLOC)}$$

SOFTWARE WORK COMPONENTS

- o SOFTWARE REQUIREMENTS DEFINITION
- o SOFTWARE DEVELOPMENT PLANNING
- o FUNCTIONAL DESIGN
- o PROGRAM DESIGN
- o MODULE DEVELOPMENT
- o SOFTWARE INTEGRATION AND TEST
- o PROBLEM ANALYSIS AND ERROR CORRECTION
- o SYSTEM TEST
- o ACCEPTANCE TEST

ESTIMATION METHODOLOGY

$$\text{TOTAL LABOR (MAN MONTHS)} = \sum_{I=1}^N P_{E_I} \times S = M$$

WHERE:

M = MAN MONTHS

N = NUMBER OF WORK COMPONENTS

P_{E_I} = WORK RATE #I

S = NUMBER OF SOURCE LINES OF CODE

THE FOUR FACTORS

WHOSE EFFECT WAS ANALYZED

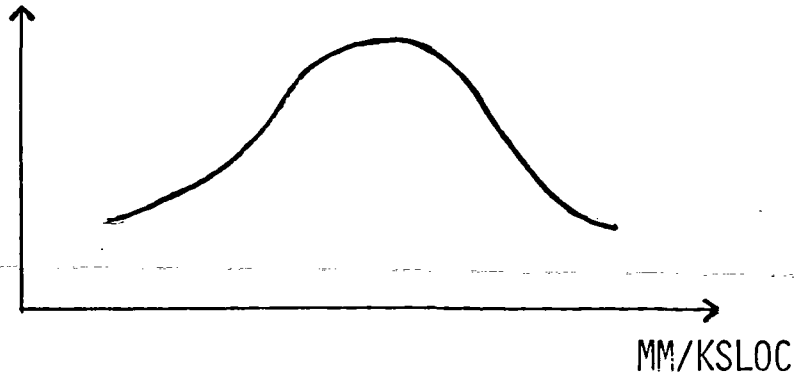
- o ORGANIZATION (2 ALTERNATIVES)
- o PRODUCT TYPE (2 ALTERNATIVES)
- o COMPUTER TYPE (3 ALTERNATIVES)
- o CODE TYPE (3 ALTERNATIVES)

OVERALL MAN MONTHS/KSLOC

DISTRIBUTION

(ONE PER WORK COMPONENT)

NO. OF CASES



(SIMULATED)

PERCENTAGE OF VARIATION IN WORK RATE
EXPLAINABLE BY FOUR FACTORS

WORK COMPONENT	PERCENTAGE OF VARIATION IN WORK RATE EX- PLAINED BY THE FOUR FACTORS	NUMBER OF SAMPLES USED
SOFTWARE REQUIREMENTS	15.12	30
SOFTWARE DEV. PLAN	17.81	38
FUNCTIONAL DESIGN	15.53	45
PROGRAM DESIGN	38.43	66
MODULE DEVELOPMENT	55.87	60
SOFTWARE INTEGRATION AND TEST (SWIT)	46.90	51
SWIT-PROBLEM ANALYSIS AND ERROR CORRECTION	60.33	51
SYSTEM TEST	26.13	39
ACCEPTANCE TEST	49.40	42
AVERAGE	36.17	
WEIGHTED AVERAGE	39.00	

SUMMARY

- o DESCRIBED WORK COMPONENT APPROACH TO ESTIMATION
- o ASSESSED IMPACT OF FOUR FACTORS ON WORK RATE
- o DETERMINED THAT THESE FOUR FACTORS ACCOUNTED FOR 39% OF THE VARIABILITY IN THE OVERALL WORK RATE
- o EXPLAINED WHY THE RESULTS DEMONSTRATE THE POWER OF THE WORK COMPONENT APPROACH

SOFTWARE METRICS:
SOFTWARE QUALITY METRICS FOR DISTRIBUTED SYSTEMS

by

Jonathan V. Post
Boeing Aerospace Company

ABSTRACT

Recent publication of numerous books and papers indicates the growing importance of Software Quality Metrics [1]. Studies at the Boeing Aerospace Company [2,3] have extended this field to cover Distributed Computer Systems. Emphasis is placed on studying Embedded computer systems, and on viewing them within a system life cycle [4]. The approach of J.A.McCall, et.al. [5,6], at General Electric was pursued and extended, maintaining the hierarchy of quality factors, criteria, and metrics [fig.1]. New software quality factors have been added, including Survivability, Expandability, and Evolvability [fig.2].

KEYWORDS

Software, Quality, Metrics, Distributed, Survivability, Life Cycle, Expandability, Evolvability, Virtuality

INTRODUCTION

What is a distributed computer system? Enslow [7] requires such a system to meet five criteria, while LeLann [8] requires it to be a collection of entities participating in system performance. Mauchley and Eckert built the first distributed computer, BINAC, circa 1947, and acknowledged [9] that the structure of the human brain, with its two cerebral hemispheres, was a guiding design metaphor. Dr. Roger Sperry's Nobel Prize in Medicine was for experiments performed at Caltech which established that the human brain is a distributed computer [10]. We consider a distributed system to be formed by the interconnection of potentially autonomous systems to accomplish system functions cooperative-

ly.

There are several ways the term "distributed" may be interpreted. Data may be distributed, processors may be distributed, processes may be distributed, users may be distributed, communications may link geographically dispersed clusters of components, or some combination of these strategies may be imposed on system architecture. Each of these types of distributedness leads to design tradeoffs, and to qualitative distinctions between centralized and distributed systems. No single model allows analysis of all such tradeoffs; data is either specialized, anecdotal, or condensed to "lessons learned" or scenario form. The application of Software Quality Metrics should help to provide a unifying framework for all such distributed systems. As Norbert Wiener first emphasized [11], it is possible to build a reliable system out of unreliable parts.

It will be increasingly important to understand distributed computer systems. Some of their characteristics will emerge more extensively in future configurations. One characteristic peculiar to distributed systems, and of importance in the 80's, is Geographic Dispersion. The extent to which computers within a distributed system can be physically displaced from each other, range from the centimeter to the multi-thousand-kilometer. Computers will indeed be "tightly-coupled" over intercontinental distances by fiber-optics technology currently under research. This technology complements that of the communications satellite. Interconnection of even a very small percentage of available computers will be able to form distributed systems of complexity beyond those of today, since by 1999 there will be on the order of one billion computers in the world [13].

QUALITY METRICS APPROACH

The approach chosen to evaluate distributed systems is the Software Quality Metrics methodology, which has been fruitfully applied to the study of a broad range of uniprocessor computers and embedded computer systems [1]. Since the 1970's, additional factors have been judged necessary in evaluating the performance of software and systems besides that of classic Reliability which was a factor closely identified with software and system quality. McCall and others [5,6] identified eleven software quality factors and developed a system of metrics to predict and assess the degree of presence of these factors. As shown in fig.1, each factor is composed of a number of criteria which are further broken down into quantitative metrics. The eleven Factors identified : Correctness, Reliability, Efficiency, Integrity, Usability, Maintainability, Testability, Flexability, Portability, Reusability, and Interoperability. The extension of this approach to distributed systems was introduced at last year's workshop by Robert W. Lawler of Boeing Aerospace Company [15]. The research conducted during the past year, as reported to RADC[2], has concentrated on identifying unique characteristics of distributed systems, and on

definition or redefinition of factors and criteria which can measure these characteristics. Three new software factors, four new system factors, twelve new software criteria, and two new system criteria have been described, and the factor of Testability has been generalized into the factor of Verifiability. Examples of these new factors and criteria are described below and in fig.2.

DISTRIBUTED SYSTEM CHARACTERISTICS

How do we approach the identification of the characteristics of distributed systems? Distributed System characteristics are identified and classified, along with rationales for the selection of Distributed Systems. 58 rationales are grouped into 9 reasons in fig.3. The rationales given for selection of a distributed system over a uniprocessor system indicate the characteristics which people imagine distributed systems, as a whole, exhibit. No one system meets more than a fraction of these identifications, just as no system life cycle for a distributed system quite fits into the system life cycle models for uniprocessor systems. Instead, we find the distributed system to be distributed through time in a distributed life cycle of concurrent phases of Operation, Revision, and Transition [fig.4].

NEW QUALITY FACTORS

The main difference between software metrics for a distributed system and software metrics for a uniprocessor system is that the quality of software in a distributed environment depends upon the design and performance characteristics of the entire system. We therefore distinguish between Software Quality Factors and System Quality Factors, although these have impact upon each other. The quality factor of Survivability, for example, reflects system performance when one or more nodes or communication links become totally nonoperational. The concepts of Reliability and Redundancy in a uniprocessor are not broad enough to describe Survivability.

Survivability is a factor which measures the capability of a system to operate when one or more components are destroyed. For a non-distributed system, Survivability is not a very meaningful measure. A single unit computer, depending on the degree of hardening and the damage received in the tactical environment, will usually either continue to operate, or else be completely incapacitated. For a geographically dispersed system, it is desirable that damage or destruction of individual components shall allow the system to continue functioning, albeit at a somewhat lower level of performance. Survivability, then, might measure the likelihood of a distributed system to exhibit this "graceful degradation". The 5 criteria within the system quality factor of Survivability are Autonomy, Distributedness, Anomaly Management, Modularity, and Reconfigurability. (See fig. 2)

Distributed Systems also require metrics to evaluate the capaci-

ty of expanding and upgrading the system, so we have identified and defined the corresponding factors of Expandability and Evolveability. Expandability is the extent to which the system capability can be expanded to enhance current functions or to add new functions. The criteria within the factor of Expandability include: Virtuality, Generality, Modularity, Augmentability, Clarity, Specificity, and Simplicity. Evolvability is the extent to which the system performance could be enhanced by the incorporation of new technology. Criteria within Evolvability are Virtuality, Generality, Modularity, Clarity, Specificity, and Simplicity. In addition, we have defined four new system quality factors, Availability, Safety, Transportability, and Interchangeability.

NEW CRITERIA

Twelve new software criteria were identified during investigation of characteristics for distributed systems [2]. These criteria are: Compliance, Validity, Clarity, Specificity, Virtuality, Comprehensibility, Reconfigurability, Distributedness, Autonomy, Supportability, Augmentability, and Compatibility [fig.5]. In addition, two new system criteria were identified: Self-containedness (an attribute of Transportability) and Homogeneity (an attribute of Interchangeability). A majority of these system and software criteria are applicable to uniprocessors as well. The following brief discussion on one of the new software criteria, Virtuality, shows how the entire system, including the human users, needs to be measured to evaluate the system quality.

For Distributed Systems, there is a new criterion within the quality factor for Usability. We refer to this criterion as Virtuality. The structure of a distributed system can be quite complex, and it is not always desirable for the user to be appraised of this structure. The user may perceive the system in terms of a virtual architecture, and be shielded from knowing the actual internal representation and location of data.

Virtuality is a measure of the extent to which the system appears to the user as it is intended to appear to the user. The user (or users) of a system is not expected or intended to see the system's logical, topological, or physical structure. Instead, an abstract "virtual" system is designed. The "real" system supports, emulates, and embodies the designed appearance and "feel" of the virtual system.

Theodor H. Nelson [12] explains the relationship between Virtuality and other criteria such as Conceptual Simplicity, Machine Independance, and Network File Availability. "Our approach to computer design we call 'the design of virtuality.' By virtuality we mean the seeming of an object or system, its conceptual structure, its atmospherics and its feel.... What counts is effects, not techniques.... The design of an interactive computer environment, similarly, should not be based on particular

hardware, or a particular display device, or a programming technique.... the systems analysis for an interactive system should deal with the mental space of the user's experience."

Virtuality also measures the subjective component of the user interface. In the special case of flight training simulators [14], the "feel" of the system has long been regarded as crucial to Usability. "Feel" is evaluated by expert pilots (superusers). This goes beyond Human Engineering, which concentrates on one display/sensory modality at a time, or on total bits per second. "Feel", and therefore Virtuality, involves gestalt perception, with an emphasis on right-brain holistic activity. Virtuality, and the human brain, cannot be ignored when studying distributed systems.

NEW METRICS

During the next year of this research effort there will be a set of metrics developed within the criteria and factors discussed above. The existing metrics [6] will be added to, deleted, and modified in accordance with results to date. The work yet to be performed may be summarized as follows:

- (1) Select Quality Metrics for Validation (Identify those metrics that will make the greatest contribution to validating the quality measurement framework previously developed);
- (2) Develop Scenarios and Collect Data (Design the data collection methodologies and gather relevant data from Boeing Aerospace Company projects which use distributed embedded computer systems);
- (3) Validate Metrics (Validation techniques consistent in concept and methodology with McCall, et.al. [6], but with multivariate regression analysis and other numerical analysis and correlation methods; conduct interviews with engineering and management personnel to supplement empirical data);
- (4) Produce a Report and Handbook (Final Report to be published by RADC. A Handbook will be prepared that describes the step-by-step procedures required to implement the quality measurements for distributed systems).

SUMMARY

Software Quality Metrics may be applied to the evaluation of distributed computer systems. Exactly what constitutes a distributed system is disputed in the literature. They have been built in various configurations for thirty years, but the human brain shares some of the characteristics of these systems and provides a valuable model. The approach of McCall et.al., with factors, criteria, and metrics, has been extended. New factors and new criteria have been defined. New metrics will be devised and validated as the research described in this paper is continued.

BIBLIOGRAPHY

- [1] Perlis, A.; Sayward, F.; Shaw, M.; "Software Metrics", MIT Press, 1981, includes 130 page annotated bibliography on Software Metrics, compiled by Mary Shaw
- [2] Post, Jonathan, and Bowen, Thomas P. "Interim Report for Quality Metrics for Distributed Systems", Boeing Document D180-26748-1, November 1981, prepared for RADC under contract F30602-80-C-0330
- [3] Henrick, John, "Performance Modeling of Distributed Computing Systems: A Literature Search", Boeing Document D182-1,0827-1 1 December 1981
- [4] Post, Jonathan; "Software Systems Engineering", Boeing Document D180-25488-5, January 1980, prepared for ASD USAF under contract number F33657-76-C-0723
- [5] McCall, J., Richards, P., Walters, G.; "Metrics for Software Quality Evaluation and Prediction", Proceedings of the NASA/Goddard Second Summer Engineering Workshop, September 1977
- [6] McCall, J.A., Matsumoto, M.T., "Software Quality Metrics Enhancements Final Report", prepared for RADC under contract number F-30602-78-C-0216
- [7] Enslow, Philip. H., "What is a 'Distributed' Data Processing System?", Computer, January 1978, p.13-21
- [8] LeLann, Gerard, "Distributed Systems -- Towards a Formal Approach," 1977 IFIP Congress Proceedings, p.155-160
- [9] Mauchley, John; Personal communication, Philadelphia, PA, June 1978
- [10] Sperry, Dr.Roger; Personal communication, Caltech, 1973
- [11] Weiner, Norbert, "Cybernetics", MIT Press, 1947
- [12] Nelson, Theodor H., "Replacing the Printed Word", Information Processing 80, Proceedings of the IFIPS Congress 1980, North-Holland Publishing Co.
- [13] Post, Jonathan V., "Quintillabit: Parameters of a Hyperlarge Database", Proceedings of the 6th International Conference on Very Large Databases, Montreal, 1-3 October 1980
- [14] Post, Jonathan V., "Software Development and Maintenance Facilities Guidebook", Boeing Document D180-25488-3, Sep. 1979, Prepared for USAF ASD, Contract F33657-76-C-0723
- [15] Lawler, R.L., "Software Quality Tradeoff Measurement", Proceedings from the Fifth Annual Software Engineering Workshop, 24 Nov 1980, Goddard Space Flight Center, NASA, Greenbelt, Maryland

**THE VIEWGRAPH MATERIALS
for the
J. POST PRESENTATION FOLLOW**

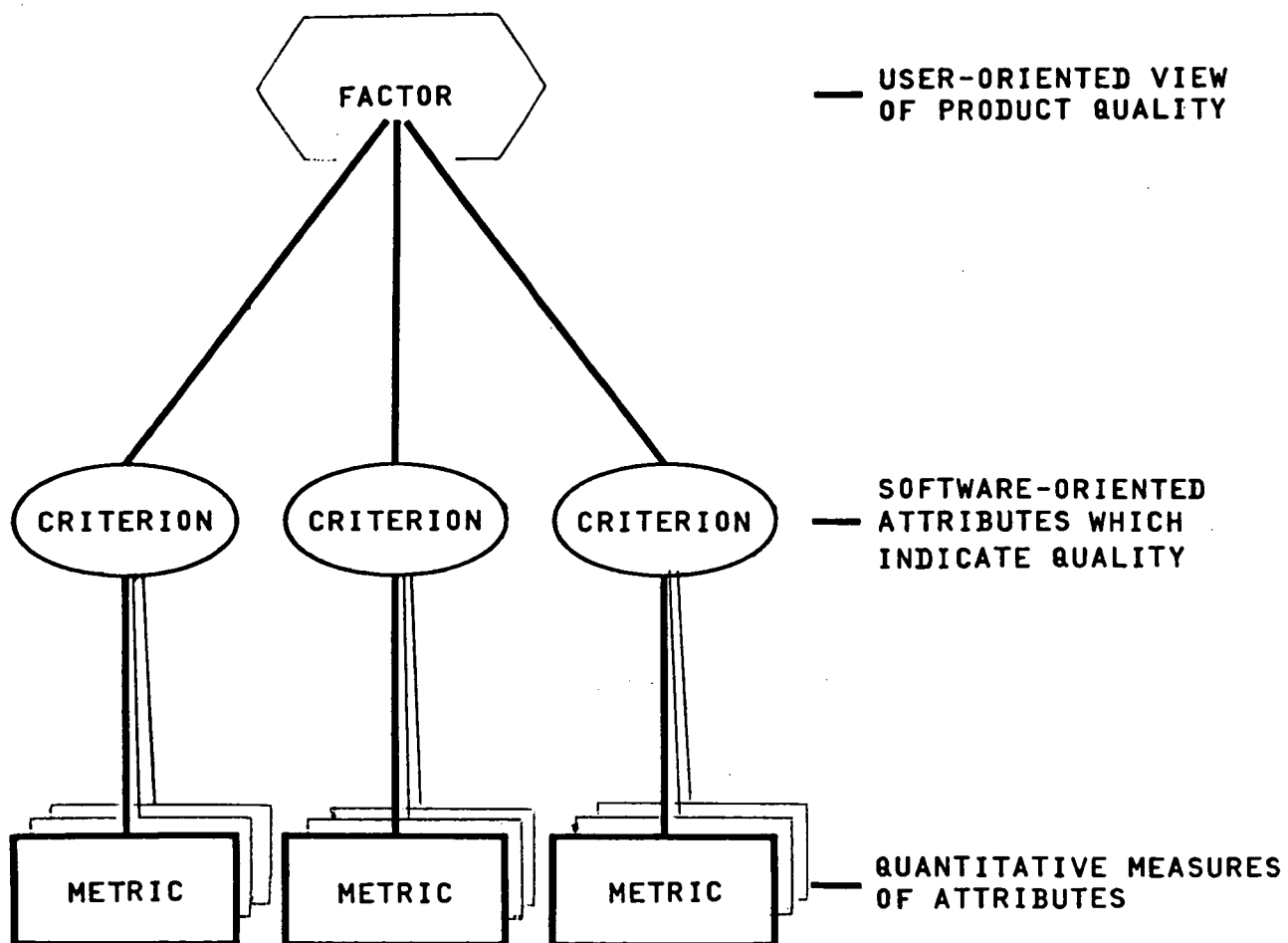


Figure 1 Software Quality Model

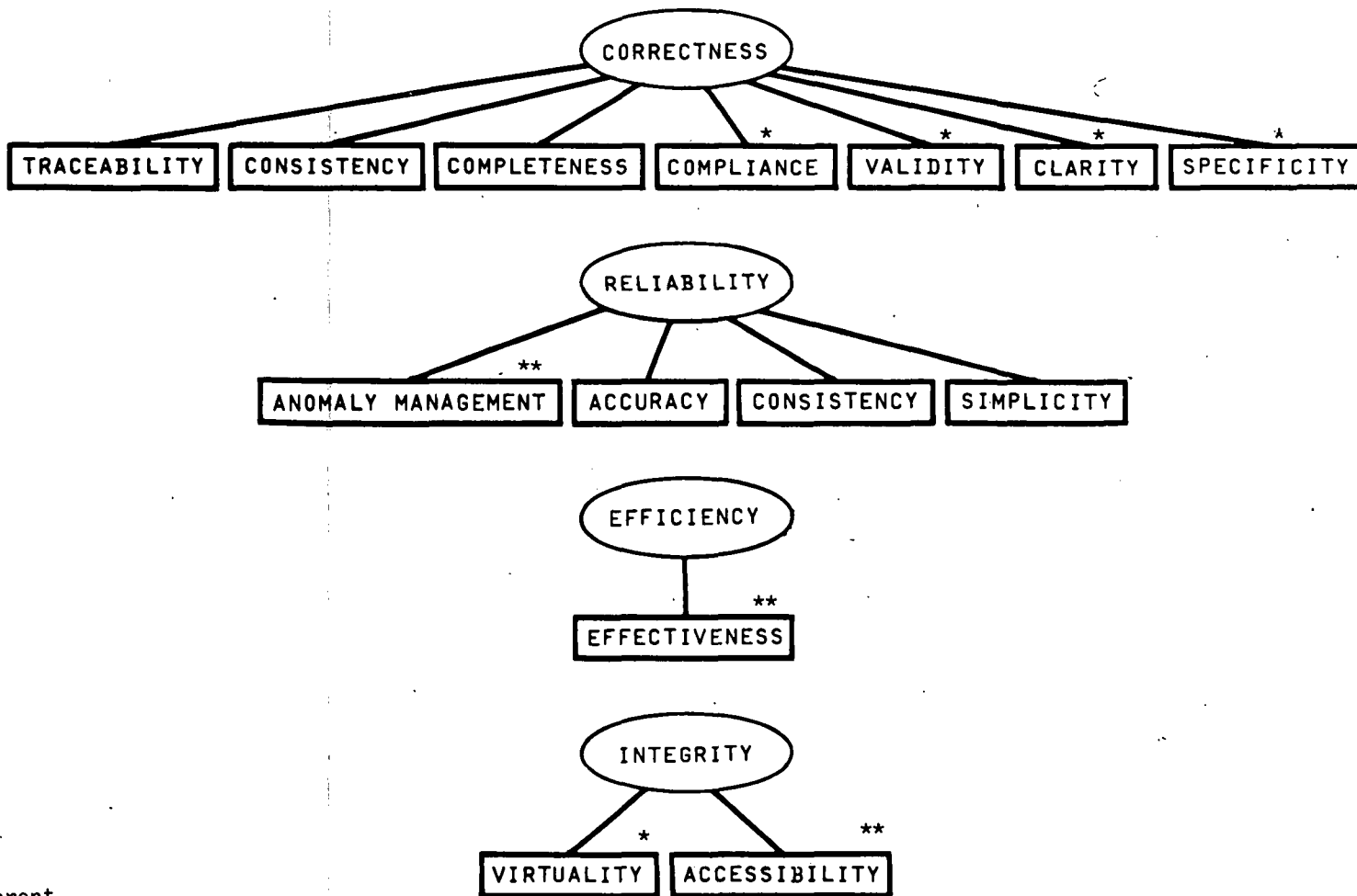
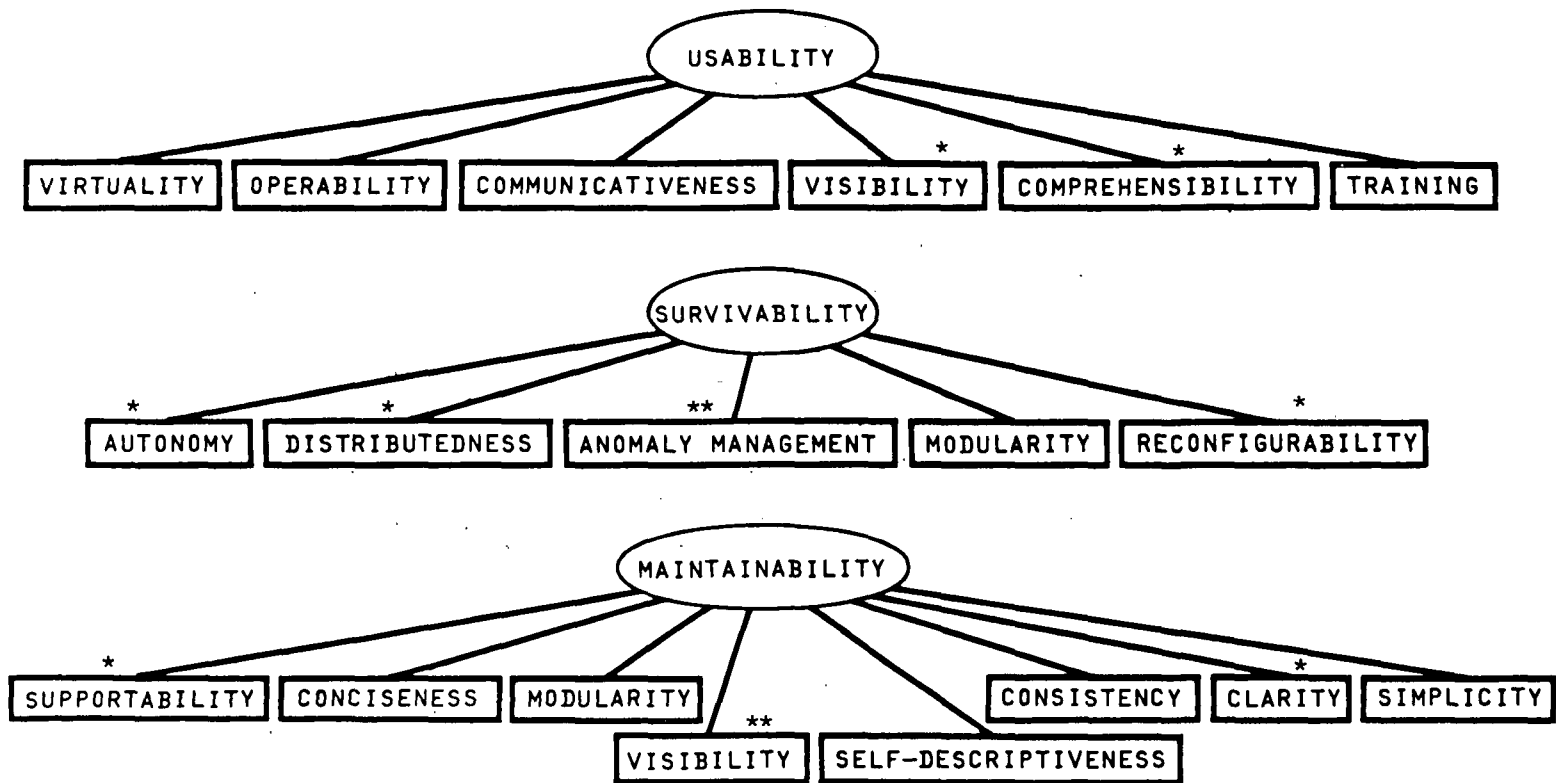


Figure 2 Relationship of Criteria to Software Quality Factors

* = New
 ** = Different



* = New
 ** = Different

Figure 2 Relationship of Criteria to Software Quality Factors

REASON NO.	REASONS FOR SELECTION OF DISTRIBUTED SYSTEMS	CORRECTNESS	MAINTAINABILITY	RELIABILITY	FLEXIBILITY	TESTABILITY	PORTABILITY	REUSEABILITY	EFFICIENCY	USABILITY	INTEGRITY	INTEROPERABILITY	SURVIVABILITY
1	IMPROVE RESPONSE TIME												
	• CONCURRENCY OF DIAGNOSIS WITH NORMAL OPERATION	X	X	X		X							X
	• ENHANCED DATA PARALLELISM				X				X	X			
	• MINIMIZE MEMORY/PROCESSOR COMMUNICATION TIME								X				
	• ALLOW OPTIMAL PARTITIONING OF WORKLOAD				X			X		X		X	
	• LOAD LEVELING				X				X	X			
	• REAL-TIME COORDINATION OF MULTIPLE SUBSYSTEMS												
2	PROVIDE GREATER PROCESSING AND ACCESSING CAPABILITIES												
	• AUTOMATIC JOB SEGMENTING				X			X	X				
	• PARTITIONING OF FUNCTIONALITY	X	X		X		X		X			X	X
	• INCREASED VARIETY OF PROCESSING MODES				X					X		X	
	• RESOURCE UNIFORMITY	X	X			X	X	X					
	• SPECIALIZED HARDWARE: DATABASE MACHINE					X			X				
	• INTEROPERABILITY WITH EXISTING SYSTEMS						X	X				X	
3	REDUCE COST												
	• LOWER COST TO UPGRADE (EXPANDABILITY)				X	X		X					
	• LOCAL ADMINISTRATIVE APPROVAL OF COMPONENTS									X		X	
	• NEW TOPOLOGICAL CONFIGURATIONS ON DEMAND		X	X	X					X			X
	• LOWER INITIAL COST								X	X			
	• INCREASED PROCURABILITY									X			
	• INCREASED DEPLOYABILITY		X		X								X
	• LOWER TOTAL WEIGHT		X		X				X				
	• LOWER TOTAL POWER CONSUMPTION		X		X				X				X
	• NETWORK TOPOLOGY OPTIMIZATION			X	X				X				X
	• RESOURCE SHARING			X	X			X		X		X	
4	REDUCE VULNERABILITY TO HARDWARE ERROR												
	• REDUNDANCY AT EACH NODE		X	X									X
	• TOLERANCE TO NODE FAILURE	X	X	X		X				X	X		X
	• TOLERANCE TO COMMUNICATIONS LINK FAILURE		X	X	X	X					X		X
	• CAPABILITY FOR ISOLATING FAILED COMPONENTS		X	X		X					X		X
	• DIAGNOSIS OF FAILURE TO LEAST REPLACEABLE UNIT		X	X		X							
	• REPAIR WITHOUT INTERRUPTION			X	X				X	X	X		X
5	REPLACE HARDWIRED LOGIC WITH MICROPROCESSOR												
	• RESOURCE UNIFORMITY	X	X			X	X	X					
	• RECONFIGURABILITY		X	X	X							X	X
	• MACHINE INDEPENDENCE	X					X	X				X	
	• DELAYED COMMITMENT TO SPECIFIC NODE HARDWARE		X		X							X	
	• MULTIPLICITY OF VENDORS		X	X	X							X	
	• RECONFIGURABILITY THROUGH LOW-COST HARDWARE				X				X	X			

Figure 3 Relationship Between Reasons, Rationales, and System Quality Factors (page 1 of 2)

<u>ACTIVITY</u>	<u>USER CONCERN</u>	<u>QUALITY FACTOR</u>
PRODUCT OPERATION	DOES IT DO WHAT IT'S SUPPOSED TO?	CORRECTNESS
	WHAT CONFIDENCE CAN BE PLACED IN WHAT IT DOES?	RELIABILITY
	HOW WELL DOES IT UTILIZE THE RESOURCES?	EFFICIENCY
	HOW SECURE IS IT?	INTEGRITY
	HOW EASY IS IT TO USE?	USABILITY
PRODUCT REVISION	HOW WELL WILL IT PERFORM UNDER ADVERSE CONDITIONS?	SURVIVABILITY*
	CAN IT BE REPAIRED?	MAINTAINABILITY
	CAN ITS OPERATION AND PERFORMANCE BE VERIFIED?	VERIFIABILITY*
PRODUCT TRANSITION	CAN IT BE CHANGED?	FLEXIBILITY
	CAN IT BE USED IN ANOTHER ENVIRONMENT?	PORTABILITY
	CAN IT BE USED IN ANOTHER APPLICATION?	REUSABILITY
	CAN IT BE INTERFACED WITH ANOTHER SYSTEM?	INTEROPERABILITY
	CAN ITS CAPABILITY BE EXPANDED?	EXPANDABILITY*
	CAN ITS PERFORMANCE BE UPGRADED WITH NEW TECHNOLOGY?	EVOLVABILITY*

* = NEW OR DIFFERENT

Figure 4 Quality Life Cycle Scheme

CRITERION	DEFINITION
• TRAINING	• THOSE ATTRIBUTES OF THE SOFTWARE WHICH PROVIDE TRANSITION FROM CURRENT OPERATION OR PROVIDE INITIAL FAMILIARIZATION.
• COMMUNICATIVENESS	• THOSE ATTRIBUTES OF THE SOFTWARE WHICH PROVIDE USEFUL INPUTS AND OUTPUTS WHICH CAN BE ASSIMILATED.
• OPERABILITY	• THOSE ATTRIBUTES OF THE SOFTWARE WHICH DETERMINE OPERATIONS AND PROCEDURES CONCERNED WITH THE OPERATION OF THE SOFTWARE.
• MODULARITY	• THOSE ATTRIBUTES OF THE SOFTWARE WHICH PROVIDE A STRUCTURE OF HIGHLY COHESIVE MODULES WITH OPTIMUM COUPLING.
• RECONFIGURABILITY*	• THOSE ATTRIBUTES OF THE SOFTWARE WHICH PROVIDE FOR CONTINUITY OF SYSTEM OPERATION WHEN ONE OR MORE PROCESSORS, STORAGE UNITS, OR COMMUNICATIONS LINKS FAIL.
• DISTRIBUTEDNESS*	• THOSE ATTRIBUTES OF THE SOFTWARE WHICH DETERMINE THE DEGREE TO WHICH SOFTWARE FUNCTIONS ARE GEOGRAPHICALLY OR LOGICALLY SEPARATED WITHIN THE SYSTEM.
• AUTONOMY*	• THOSE ATTRIBUTES OF THE SOFTWARE WHICH DETERMINE ITS DEPENDENCY ON INTERFACES.
• CONCISENESS	• THOSE ATTRIBUTES OF THE SOFTWARE WHICH PROVIDE FOR IMPLEMENTATION OF A FUNCTION WITH A MINIMUM AMOUNT OF CODE.
• SUPPORTABILITY*	• THOSE ATTRIBUTES OF THE SOFTWARE WHICH PROVIDE FOR EASE IN CREATION OF NEW SOFTWARE VERSIONS (e.g., USE OF HOL, VERSION UPDATE SCHEME).
• SELF-DESCRIPTIVENESS	• THOSE ATTRIBUTES OF THE SOFTWARE WHICH PROVIDE EXPLANATION OF THE IMPLEMENTATION OF A FUNCTION.
• GENERALITY	• THOSE ATTRIBUTES OF THE SOFTWARE WHICH PROVIDE BREADTH TO THE FUNCTIONS PERFORMED.
• INDEPENDENCE**	• THOSE ATTRIBUTES OF THE SOFTWARE WHICH DETERMINE ITS DEPENDENCY ON THE SOFTWARE ENVIRONMENT (COMPUTING SYSTEM, OPERATING SYSTEM, UTILITIES, INPUT/OUTPUT ROUTINES, LIBRARIES).
• AUGMENTABILITY*	• THOSE ATTRIBUTES OF THE SOFTWARE WHICH PROVIDE EXPANSION CAPABILITY FOR FUNCTIONS AND DATA.
• COMPATIBILITY*	• THOSE ATTRIBUTES OF THE SOFTWARE WHICH PROVIDE INTERFACE PROTOCOLS AND ROUTINES THAT ARE APPROPRIATE TO THE INTERFACE EQUIPMENT FEATURES AND CAPABILITIES.
• COMMONALITY**	• THOSE ATTRIBUTES OF THE SOFTWARE WHICH PROVIDE FOR THE USE OF INTERFACE STANDARDS FOR PROTOCOLS, ROUTINES, AND DATA REPRESENTATIONS.

* = New
** = Different

Figure 5 Software Quality Criteria Definitions

CRITERION	DEFINITION
• TRACEABILITY	• THOSE ATTRIBUTES OF THE SOFTWARE WHICH PROVIDE A THREAD OF ORIGIN FROM THE IMPLEMENTATION TO THE REQUIREMENTS WITH RESPECT TO THE SPECIFIED DEVELOPMENT ENVELOPE AND OPERATIONAL ENVIRONMENT.
• CONSISTENCY	• THOSE ATTRIBUTES OF THE SOFTWARE WHICH PROVIDE FOR UNIFORM DESIGN AND IMPLEMENTATION TECHNIQUES AND NOTATION.
• COMPLETENESS	• THOSE ATTRIBUTES OF THE SOFTWARE WHICH PROVIDE FULL IMPLEMENTATION OF THE FUNCTIONS REQUIRED.
• COMPLIANCE*	• THOSE ATTRIBUTES OF THE SOFTWARE WHICH PROMOTE IMPLEMENTATIONS THAT CONFORM TO THE REQUIREMENTS.
• VALIDITY*	• THOSE ATTRIBUTES OF THE SOFTWARE WHICH CONSTRAIN IMPLEMENTATIONS TO A RANGE OF ACCEPTABLE SOLUTIONS.
• CLARITY*	• THOSE ATTRIBUTES OF THE SOFTWARE WHICH PROVIDE NON-AMBIGUOUS DESCRIPTIONS OF FUNCTIONS AND IMPLEMENTATIONS.
• SPECIFICITY*	• THOSE ATTRIBUTES OF THE SOFTWARE WHICH PROVIDE SINGULARITY IN THE DEFINITION AND IMPLEMENTATION OF FUNCTIONS.
• SIMPLICITY	• THOSE ATTRIBUTES OF THE SOFTWARE WHICH PROVIDE FOR THE DEFINITION AND IMPLEMENTATION OF FUNCTIONS IN THE MOST NON-COMPLEX AND UNDERSTANDABLE MANNER.
• ANOMALY MANAGEMENT**	• THOSE ATTRIBUTES OF THE SOFTWARE WHICH PROVIDE FOR CONTINUITY OF OPERATIONS UNDER AND RECOVERY FROM NON-NOMINAL CONDITIONS.
• ACCURACY	• THOSE ATTRIBUTES OF THE SOFTWARE WHICH PROVIDE THE REQUIRED PRECISION IN CALCULATIONS AND OUTPUTS.
• EFFECTIVENESS**	• THOSE ATTRIBUTES OF THE SOFTWARE WHICH PROVIDE FOR MINIMUM UTILIZATION OF RESOURCES (PROCESSING TIME, STORAGE, OPERATOR TIME) IN PERFORMING FUNCTIONS.
• ACCESSIBILITY**	• THOSE ATTRIBUTES OF THE SOFTWARE WHICH PROVIDE FOR CONTROL AND AUDIT OF ACCESS TO THE SOFTWARE AND DATA.
• VIRTUALITY*	• THOSE ATTRIBUTES OF THE SOFTWARE WHICH PRESENT A SYSTEM THAT DOES NOT REQUIRE USER KNOWLEDGE OF THE PHYSICAL CHARACTERISTICS (e.g., NUMBER OF PROCESSORS/DISKS, STORAGE LOCATIONS)
• VISIBILITY**	• THOSE ATTRIBUTES OF THE SOFTWARE WHICH PROVIDE STATUS MONITORING OF THE DEVELOPMENT AND OPERATION (e.g., INSTRUMENTATION).
• COMPREHENSIBILITY*	• THOSE ATTRIBUTES OF THE SOFTWARE WHICH ENHANCE UNDERSTANDING OF THE OPERATION OF THE SOFTWARE.

Figure 5 Software Quality Criteria Definitions

* = New
** = Different

IDENTIFICATION AND EVALUATION
OF SOFTWARE MEASURES

David N. Card

COMPUTER SCIENCES CORPORATION

and

GODDARD SPACE FLIGHT CENTER
SOFTWARE ENGINEERING LABORATORY

Prepared for the

NASA/GSFC

Sixth Annual Software Engineering Workshop

INTRODUCTION

The purpose of this presentation is to describe and demonstrate a large-scale, systematic procedure for identifying and evaluating measures that meaningfully characterize one or more elements of software development. The background of this research, the nature of the data involved, and the steps of the analytic procedure are discussed. The presentation concludes with an example of the application of this procedure to data from real software development projects.

As the term is used here, a measure is a count or numerical rating of the occurrence of some property. Examples of measures include lines of code, number of computer runs, person-hours expended, and degree of use of top-down design methodology. Measures appeal to the researcher and the manager as a potential means of defining, explaining, and predicting software development qualities, especially productivity and reliability.

Measures may be classified into four groups as illustrated by the software development model presented in Figure 1. It shows these components: a problem, a solution-generating process, the environment in which that process takes place, and the solution (or software product). Measures can be employed to characterize the components of this model and to show their interrelationships. Some examples of appropriate measures for each component are also shown in the figure.

The Goddard Space Flight Center (GSFC) Software Engineering Laboratory (SEL) is engaged in an effort, part of which this presentation describes, to develop a concise set of such characteristic measures. The SEL and its activities are discussed in more detail in Reference 1.

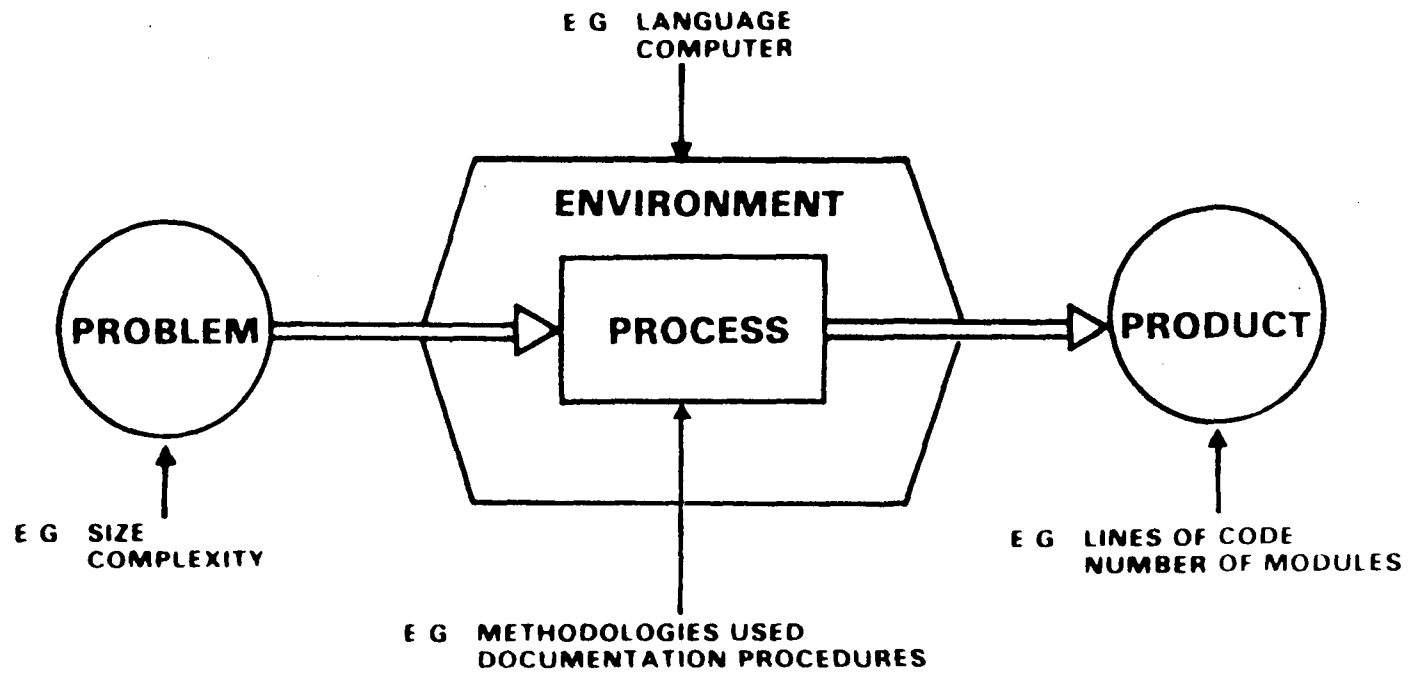


Figure 1. SEL Software Development Model

The approach to software measurement adopted in this presentation is different from that generally followed. The usual procedure is to select high-level "qualities" and then to seek numerical criteria or measures of these qualities. McCall (Reference 2) has developed a comprehensive system of such qualities and appropriate measures. However, the goal of the approach followed here is to identify the qualities being measured by the data collected rather than to attempt to associate measures with previously specified qualities. The measures considered in this analysis are described in the next section.

DATA DESCRIPTION

Clearly, the number of potentially useful measures is large; the SEL has selected more than 200 for study. These measures cover the entire range of software development activity as experienced by the SEL. However, the analysis described here will focus on the relationships among measures of the process and product components of the software development model (see Figure 1).

Therefore, a data subset containing only the 60 measures relevant to those two components was used. The measures (or variables) used are listed in Table 1 (see Appendix A). This list does not necessarily exhaust the possibilities for measures in those areas; however, this group of measures is believed to form a comprehensive set. The process measures class is represented by three subclasses: methodology (Table 1a), tools (Table 1b), and documentation (Table 1c). Note that the methodology class is further subdivided by development phase into design, code, and test measures. The product class (Table 1d) includes size and resource measures.

The data used in this analysis were collected by the SEL from 22 actual medium-scale, scientific software development projects. Values for all these measures were determined for each project. The values are ratings of the degree of use, counts, or rates per line of code, as indicated in Table 1. Degree-of-use process measures are expressed as relative scores on a scale from zero to five. The exact derivation of these scores will be explained in a forthcoming SEL document (Reference 3).

ANALYTIC PROCEDURE

The 60 measures just described are not unique or independent. Some may, in fact, measure the same or related qualities. The object of the analytic procedure is to identify the most basic set of qualities (or properties) being measured by the group of 60. A "basic" quality is defined to be one that is independent of all other such qualities. This subset, then, defines the basic quality characteristics describing the projects from which the data were obtained.

The procedure to be proposed is "large scale." That is, it is appropriate when a large number of measures (or variables) are to be evaluated. The researcher interested in studying the relationships of only a few specific measures can probably get better results from regression and hypothesis testing techniques. Nevertheless, this procedure can be useful as a screening tool for detecting confounding effects in the data before selecting other statistical techniques.

The analytic procedure followed in this experiment has two steps, as indicated in Figure 2. These are the application of a test of normality to the candidate measures (data), followed by a factor analysis of those not rejected by the test. The result of this procedure is a descriptive, rather than a predictive, model of the data. The procedure identifies the descriptive factors common to the set of measures. Thus, the original measures are organized into a number of groups (or factors) smaller than the number of measures input to the procedure. These factors correspond to the basic qualities sought for in the data. The steps of this procedure are discussed in more detail in the following sections.

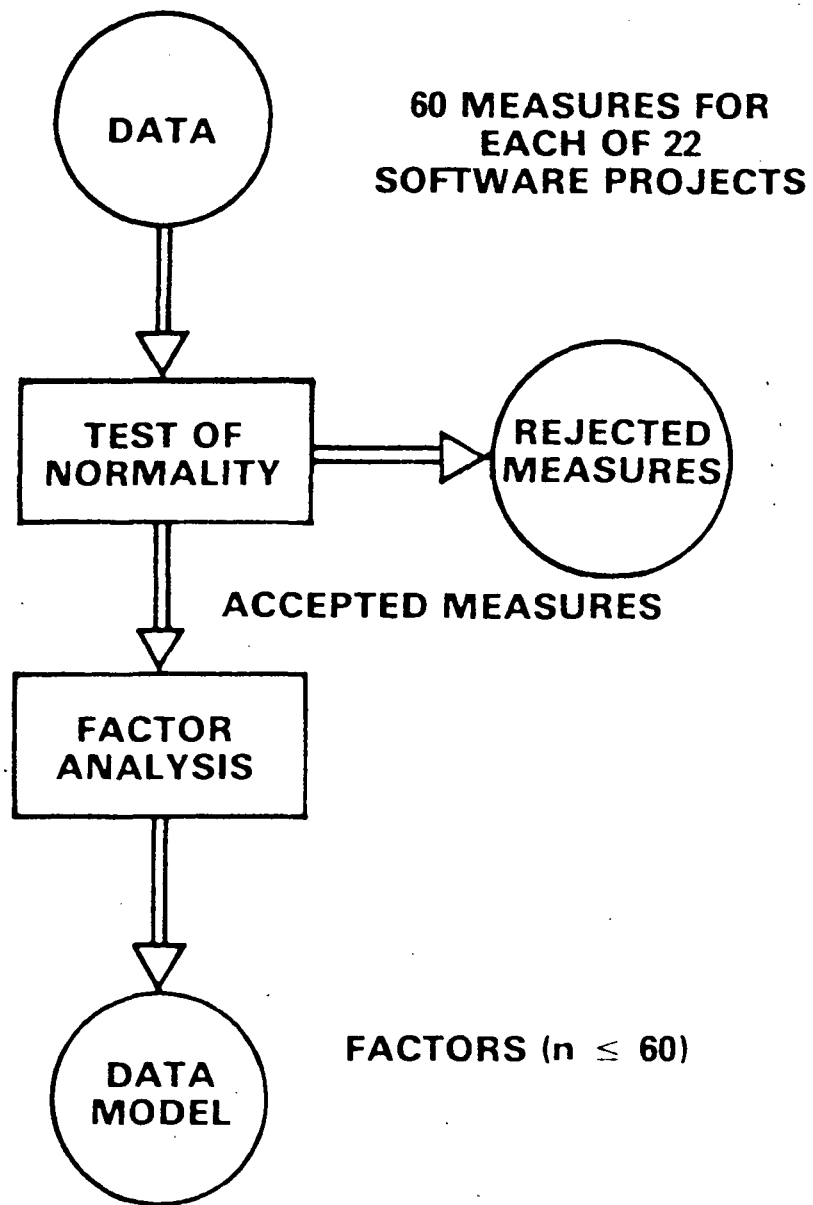


Figure 2. Analytic Procedure

TEST OF NORMALITY

The test of normality analyzes the probability distribution of a measure. The observed values of each measure are distributed over some range. The normal distribution is readily identifiable in Figure 3. The test of normality will detect measures whose values are distributed in a pattern significantly different from the normal. For example, it would reject a measure with values clustered at one end of the range (skewed) rather than distributed symmetrically across it.

This is not a very powerful test. It will accept any approximately symmetrical distribution even if that distribution is not truly normal. However, the test is important because approximate normality of the data is an assumption of step two, the factor analysis.

Six measures from the set of 60 were rejected by the test of normality using the 0.05 level of significance. These are measures of techniques for which insufficient examples of use were available. Consequently, most projects had scores of zero for these degree-of-use measures, a result that produced dramatically skewed distributions. They are

- HIPO Design Technique
- Verification and Validation Team (two measures)
- Requirements Language Tool
- Configuration Management Tool
- Unit Development Folders

These measures could, however, be used in some other types of analyses not considered here.

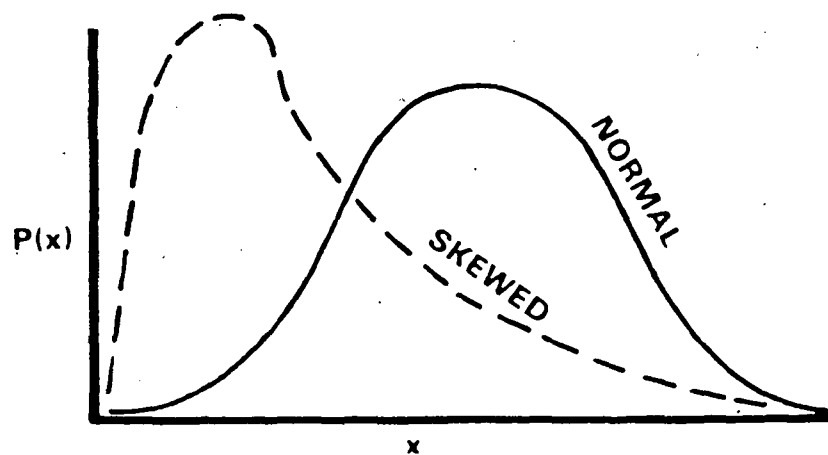


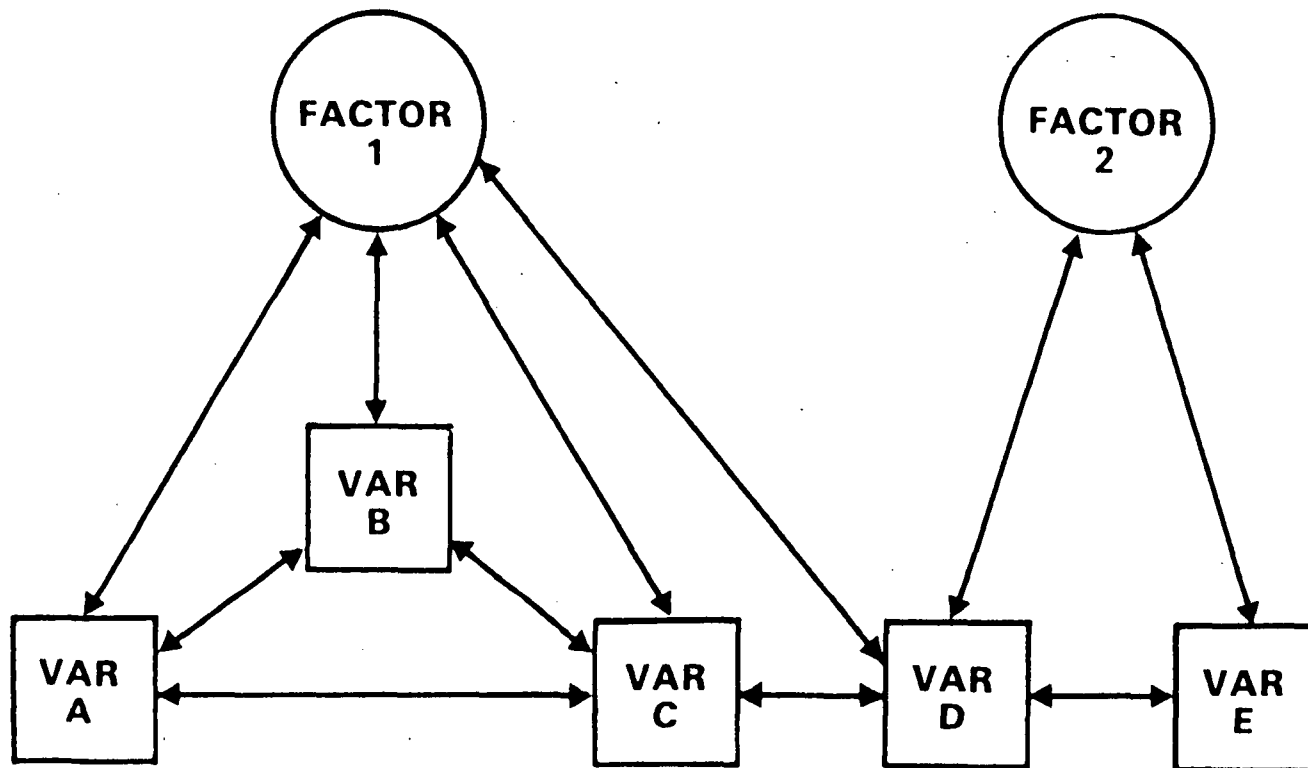
Figure 3. Test of Normality

FACTOR ANALYSIS

The 54 remaining measures were included in the factor analysis. The goal of the factor analysis is to "discover" the underlying structure of the data. Factor analysis hypothesizes the existence of a set of statistically independent "factors" that are not directly measurable by the experimenter. Measures (or variables) are the quantities that are observed in practice. However, the apparent correlations among measures can be interpreted to be due to their joint correlation with common factors (see Figure 4). That is, two or more measures correlated with the same factor will be correlated with each other. The desirable result of a factor analysis is the extraction of a smaller set of factors whose relationships are known (they are independent) from the larger set of measures whose relationships are more complex.

Consider this example of the factor concept. The number of errors in a piece of software and its mean time to failure are measures related to reliability and are correlated with each other. However, neither measure by itself is a full description of reliability. Such things as the location of the error and the severity of the failure must also be considered. Therefore, the reliability quality factor is not directly measurable although a number of measurable variables are correlated with it.

A successful factor analysis will explain such groups of related measures. Thus, each factor defined will correspond to a distinct basic quality being measured by the original set of variables. These qualities are the sources of variation (or differentiation) among the projects studied.



NOTE: VARIABLES MAY BE CORRELATED.
FACTORS ARE INDEPENDENT.

Figure 4. Concept of Factor Analysis

The principles of factor analysis are explained in detail in the text by Harman (Reference 4). A number of software implementations of factor analysis are available. The specific software used in this analysis was the principal components factor procedure of the Statistical Analysis System (Reference 5).

SUMMARY OF RESULTS

Further analysis of the 54 process and product measures that passed the test of normality produced a factor model containing 5 factors that explained 77 percent of the variance of the original measures. The meaning of each factor is determined by examining the measures that are closely correlated with it. These factors and the amount of variance accounted for by each are as follows:

- Methodology Intensity (31%)
- Project Size (25%)
- Computer Usage (9%)
- Quality Assurance (8%)
- Change Rate (5%)

The variance associated with a factor is a measure of the degree to which that factor differentiates among the projects (or cases) studied. Thus, it is a measure of information content. A larger portion of the total variance could have been accounted for by using a larger number of factors. The relationship of the number of factors to the variance explained by the factor model is illustrated in Table 2 of Appendix A. The interpretation of additional factors is difficult because none of the original measures are highly correlated with them. Therefore, they are not included in this preliminary definition of the factor model.

The correlations of the original measures with the five factors are listed in Table 3 of Appendix A. Only correlations greater than 0.526 (the 0.01 level of significance) are reproduced. The measure showing the highest correlation with a factor can be taken as the best estimator of that quality factor from among the original measures included in the analysis. These "best" estimators are indicated by asterisks in the tables.

Remember that, although the factors are mutually independent, any given measure may be correlated with more than one factor and/or with other measures. The factor model does, however, identify the strongest relationships in the data. Some specific observations are made below about each of the factors defined by the analysis.

Factor 1 - The first and most powerful factor (Table 3a in Appendix A) is highly correlated with degree-of-use process measures; thus, this factor may be interpreted to represent the degree to which formal methodology was applied during development. The most strongly correlated measure, methodology reinforcement (the extent to which adherence to specified methodologies was enforced by management), supports this interpretation. The strong correlation of so many methodology, tool, and documentation measures with a common factor suggests that simple regression and hypothesis testing techniques are inappropriate for analyzing such effects because of their inability to isolate the action of a single technique from among the actions of other techniques.

Factor 2 - The second factor (Table 3b in Appendix A) is clearly related to the size of the development effort and product. Its "best" estimator is person-hours. The correlation of top-down coding with this factor illustrates the descriptive, rather than predictive, nature of factor analysis. The proper conclusion based on this observation is that more top-down coding tends to be used in small projects than in large ones, not that top-down coding necessarily reduces the size of a development effort.

Factor 3 - The third factor (Table 3c in Appendix A) contains a number of measures related to the pattern of computer usage. This factor indicates that the manner and degree of computer usage reflect the use of certain development tools and techniques. The "best" estimator of this factor is top-down design.

Factor 4 - The fourth factor (Table 3d in Appendix A) has only one measure, semiformal quality assurance, significantly correlated with it. Thus, its meaning is difficult to establish. However, a substantial amount of variance (8 percent) is associated with this factor. The preceding factor contained five variables but explained only slightly more variance (9 percent). Thus, this factor and measure deserve closer examination in future analyses.

Factor 5 - The last factor (Table 3e in Appendix A) clearly describes the change rate. The interpretation of this factor is important since, as a consequence of the mutual independence of factors, it is independent of the four factors previously defined. Hence, methodology intensity, project size, and computer usage do not appear to be related to each other or to code stability (reliability), as measured by the change rate.

Another feature of this model should be noted. Although productivity was most strongly correlated with factor 4, it was not significantly correlated with any factor. Productivity may still be related to specific methodologies but not to the general factors just defined. Thus, the information provided by this procedure about productivity and reliability is negative in this example because unrelated qualities and measures were identified rather than related ones.

CONCLUSION

The results presented here are preliminary. Conclusions based on the factor model just developed may change as more data become available and as the procedure is refined. However, the analysis has demonstrated its capacity to resolve some important questions about the data. The conclusions are as follows: the basic qualities being quantified by the original measures can be identified and enumerated; their relative importance or strength (in terms of percentage of variance accounted for) can be established; and a "best" estimator can be selected for each quality.

Therefore, we can define a concise set of quality measures that meaningfully characterizes the process and product components of the software development model and that can serve as a framework for further research. These qualities and associated measures can be studied in greater detail with other techniques to determine their relationships to productivity and reliability more exactly. Hence, these results are a first step toward defining, explaining, and predicting software reliability and productivity in the SEL environment.

APPENDIX A - SUMMARY OF FACTOR ANALYSIS

This appendix consists of a series of three tables that summarize the factor analysis procedure described in the preceding discussion. Table 1 describes the measures evaluated in this analysis. Table 2 identifies the variances associated with factors. Table 3 lists the significant correlations (at the 0.01 level of significance) of measures with factors.

Table 1a. Methodology Measures

(DEGREE OF USE)

ORGANIZATION — CHIEF PROGRAMMER

DESIGN	— WALKTHROUGHS
DESIGN	— FORMAL REVIEWS
DESIGN	— FORMALISMS
DESIGN	— TREE CHARTS
DESIGN	— PROGRAM DESIGN LANGUAGE (PDL)
DESIGN	— HIERARCHICAL INPUT PROCESSING OUTPUT (HIPO)
DESIGN	— TOP-DOWN
DESIGN	— ITERATIVE ENHANCEMENT
CODE	— STUBS
CODE	— TOP-DOWN
CODE	— STRUCTURED
CODE	— WALKTHROUGHS
CODE	— READ
CODE	— CONFIGURATION CONTROL
TEST	— FORMALISM
TEST	— FOLLOWTHROUGH
TEST	— BATCH
TEST	— V&V PRESENCE
TEST	— V&V USE

Table 1b. Tools Measures

(DEGREE OF USE)

FORMAL TRAINING IN METHODOLOGY
INFORMAL TRAINING
METHODOLOGY REINFORCEMENT
REQUIREMENTS LANGUAGE (MEDL-R)
DESIGN LANGUAGE (PDL)
PRECOMPILER (SFORT)
SOFTWARE AIDS (e.g., EXREF, MAP, LIST)
LIBRARIAN
DATA GENERATORS
TERMINALS (TSO)
REMOTE JOB PROCESSING (RJP)
CONFIGURATION ANALYSIS (CAT)

Table 1c. Documentation Measures

(DEGREE OF USE)

**SEL FORMS
DESIGN DOCUMENT
DESIGN DECISIONS
SEMIFORMAL QUALITY ASSURANCE
ACTIVITY NOTEBOOKS
UNIT DEVELOPMENT FOLDERS
TEST PLANS
USER'S GUIDE/SYSTEM DESCRIPTION
FORMAL TREATMENT OF USER'S GUIDE
WEEKLY/MONTHLY PROGRESS REPORTS**

Table 1d. Resource/Product Measures

(COUNTS AND RATES)

NUMBER OF COMPONENTS
TOTAL MODULES
NEW MODULES
MODIFIED MODULES
TOTAL LINES OF CODE (INCLUDES COMMENTS)
NEW LINES OF CODE (INCLUDES COMMENTS)
MODIFIED LINES OF CODE
NUMBER OF COMPUTER RUNS
NUMBER OF CHANGES
PAGES OF DOCUMENTATION
TOTAL MANHOURS
TOTAL COMPUTER HOURS
PERCENT OF NEW CODE
CHANGES PER LINE OF CODE
CHANGES PER LINE OF NEW CODE
NEW LINES + 20% OF REVISED LINES
LINES OF CODE PER MANHOUR
COMPUTER HOURS PER LINE OF CODE

Table 2. Preliminary Eigenvalues and Variances Associated With Factors

FACTOR	1	2	3	4	5	6	7	8	9	10	11
EIGENVALUES	16.492786	13.305087	4.744286	4.063959	2.855640	2.438981	1.738979	1.555128	1.469211	1.101198	0.931850
PORTION	0.305	0.246	0.088	0.075	0.053	0.045	0.032	0.029	0.027	0.020	0.017
CUM PORTION	0.305	0.552	0.640	0.715	0.768	0.813	0.845	0.874	0.901	0.922	0.939
FACTOR	12	13	14	15	16	17	18	19	20	21	
EIGENVALUES	0.685056	0.621503	0.495917	0.427863	0.397183	0.255911	0.209853	0.153974	0.055635	0.000000	
PORTION	0.013	0.012	0.009	0.008	0.007	0.005	0.004	0.003	0.001	0.000	
CUM PORTION	0.952	0.963	0.972	0.980	0.987	0.992	0.996	0.999	1.000	1.000	

NOTE: Only five factors were retained in the analysis.

Table 3a. Factor 1

<u>MEASURE</u>	<u>CORRELATION</u>
CHIEF PROGRAMMER ORGANIZATION	.62
DESIGN WALKTHROUGHS	.75
FORMAL DESIGN REVIEWS	.75
DESIGN FORMALISMS	.83
DESIGN TREE CHARTS	.65
PROGRAM DESIGN LANGUAGE (METHODOLOGY)	.63
CODE STUBS	.86
CODE WALKTHROUGHS	.69
CODE READING	.60
CONFIGURATION CONTROL (METHODOLOGY)	.62
TEST FORMALISMS	.74
TEST FOLLOWTHROUGH	.72
FORMAL TRAINING IN METHODOLOGY	.78
INFORMAL TRAINING	.61
METHODOLOGY REINFORCEMENT	.89*
DESIGN LANGUAGE (TOOL)	.64
SOFTWARE (CODING) AIDS	.68
LIBRARIAN	.85
DATA GENERATORS	.71
REMOTE JOB ENTRY	.54
SEL FORMS	.76
DESIGN DOCUMENT	.73
DESIGN DECISION (DOCUMENTATION)	.75
ACTIVITY NOTEBOOKS	.76
USER'S GUIDE/SYSTEM DESCRIPTION	.69
WEEKLY/MONTHLY PROGRESS REPORTS	.69

NOTE: VARIANCE ACCOUNTED FOR: 31%.

Table 3b. Factor 2

<u>MEASURE</u>	<u>CORRELATION</u>
NUMBER OF COMPONENTS	.89
TOTALS MODULES	.89
NEW MODULES	.85
MODIFIED MODULES	.80
TOTAL LINES	.91
NEW LINES	.92
MODIFIED LINES	.77
NUMBER OF RUNS	.91
NUMBER OF CHANGES	.93
PAGES OF DOCUMENTATION	.94
PERSON HOURS	.96*
COMPUTER HOURS	.88
DELIVERED LINES	.93
TOP-DOWN CODING	— .56

NOTE: VARIANCE ACCOUNTED FOR: 25%.

Table 3c. Factor 3

<u>MEASURE</u>	<u>CORRELATION</u>
COMPUTER HOURS/LINES OF CODE	.60
TOP-DOWN DESIGN	.88*
BATCH TESTING	.70
REMOTE JOB ENTRY	.69
TEST PLANS	-.57

NOTE: VARIANCE ACCOUNTED FOR: 9%.

Table 3d. Factor 4

<u>MEASURE</u>	<u>CORRELATION</u>
SEMIFORMAL QUALITY ASSURANCE	.60*
(PRODUCTIVITY	— .30)

NOTE: VARIANCE ACCOUNTED FOR: 8%.

Table 3e. Factor 5

<u>MEASURE</u>	<u>CORRELATION</u>
CHANGES/LINES OF CODE	.73*
CHANGES/LINES OF NEW CODE	.64

NOTE: VARIANCE ACCOUNTED FOR: 5%.

REFERENCES

1. Computer Sciences Corporation, CSC/TM-81/6104, The Software Engineering Laboratory, D. N. Card, et al., October 1981
2. Rome Air Development Center, RADC-TR-77-369, Factors in Software Quality, J. A. McCall, P. K. Richards, and G. F. Walters, November 1977
3. Computer Sciences Corporation, Evaluation and Application of Subjective Measures of Software Development, D. Card and G. Page (in preparation)
4. H. H. Harman, Modern Factor Analysis, Chicago: University of Chicago Press, 1976
5. J. T. Sall, et al., Statistical Analysis System User's Guide, SAS Institute, 1979

**THE VIEWGRAPH MATERIALS
for the
D. CARD PRESENTATION WERE
INCORPORATED IN THE PAPER**

PANEL #3

SOFTWARE MODELS

B. Littlewood/A. Sofer, George Washington University

H. Sayani/C. Svoboda, Advanced Systems Technology Corporation

SOFTWARE MODELS:

A BAYESIAN APPROACH TO PARAMETER ESTIMATION IN THE JELINSKI-MORANDA SOFTWARE RELIABILITY MODEL

by

Bev Littlewood, The City University, London, England
Ariela Sofer, The George Washington University, Washington, D.C.

Abstract

Maximum likelihood estimation procedures for the Jelinski-Moranda software reliability model often give misleading answers. We show here that a reparameterization and a Bayesian analysis eliminate some of the problems incurred by MLE methods and often give better predictions on sets of real and simulated data.

Practical difficulties in estimating the initial number of errors N and the failure rate of each error ϕ by the method of maximum likelihood are:

1. \hat{N} , the MLE of N , is occasionally infinite (i.e., the routines for calculating \hat{N} and $\hat{\phi}$ do not converge). Littlewood and Verrall show that \hat{N} is finite if and only if the regression line of the interevent times t_i vs. i has positive slope.
2. A serious problem is that often $\hat{N} \approx n$, the sample size, and sometimes $\hat{N} = n$. Thus the MLE predicts that the program is perfect even when it is far from being so. Forman and Singpurwalla have shown that \hat{N} and $\hat{\phi}$ can only be trusted near the end of debugging, i.e., when almost all failures have been removed.

3. Even when these problems are not encountered, the results obtained from the model are too optimistic; it predicts the reliability to be greater than it really is.

In view of these deficiencies, we are led to consider a Bayesian approach to the estimation problem. It seems plausible that it is easier to correctly estimate the initial program failure rate $\lambda = N\phi$ than the initial number of bugs N , since small errors in $\hat{\phi}$ could lead to large errors in \hat{N} . It is therefore plausible to reparameterize the model to (λ, ϕ) instead of (N, ϕ) .

Using now the Bayesian approach, letting $\text{prior}(\lambda, \phi) = \text{prior}(\lambda) \cdot \text{prior}(\phi)$, where $\text{prior}(\lambda)$ and $\text{prior}(\phi)$ are gamma distributed, and using

$$\begin{aligned} R_{n+1}(t) &= P(T_{n+1} < t) = P(T_{n+1} < t \mid t_1, \dots, t_n) \\ &= \int P(T_{n+1} < t \mid \lambda, \phi) \text{post}(\lambda, \phi \mid t_1, \dots, t_n) d\lambda d\phi \end{aligned}$$

we obtain an explicit estimate of the program's current reliability.

Similarly, we can get in closed form the distributions of the number of bugs remaining in the program, the number of bugs that have to be removed in order to attain a given reliability, and the times between future consecutive failures (provided they are well defined, i.e., the program is not perfect).

The quality of these estimations was examined for the special case when λ and ϕ have an (improper) uniform prior distribution over $[0, \infty)$ (i.e., a noninformative prior distribution). The predictions were examined both for real and for simulated sets of data. In all cases where ML erroneously predicts the program to be perfect, the Bayesian method gives a positive probability that the program is not perfect. Moreover,

since the predicted reliability is given in closed form, problems of convergence of the computer program are not encountered.

To examine the quality of prediction, we use a goodness of fit procedure. Suppose that from the data t_1, \dots, t_n we predict the distribution of T_{n+1} , the time to next failure. We then observe t_{n+1} . Define $U_n = \Pr(T_{n+1} < t_{n+1})$. If the model is correct, then U_n are uniform variables on $(0, 1)$. We compare the sample c.d.f. of the u_n 's with a line of unit slope which is the uniform c.d.f.

When applying the goodness of fit procedure to real data sets, the Bayesian approach is almost always better than the MLE method. For the simulated data, the goodness of fit procedure on the Bayesian estimates give very good results; this, however, is not always true for the real data sets.

There seems to be evidence that the J-M model is intrinsically optimistic in its estimate of software reliability. This could be a consequence of the assumption that all errors contribute equally to the failure rate. A new model by Littlewood relaxes this assumption with the result that earlier fixes tend to involve larger reductions in the failure rate than the later ones. It can be shown that this model is less optimistic than the J-M model and we hope to examine its performance on real and simulated data in future work.

THE VIEWGRAPH MATERIALS
for the
B. LITTLEWOOD/A. SOFER PRESENTATION FOLLOW

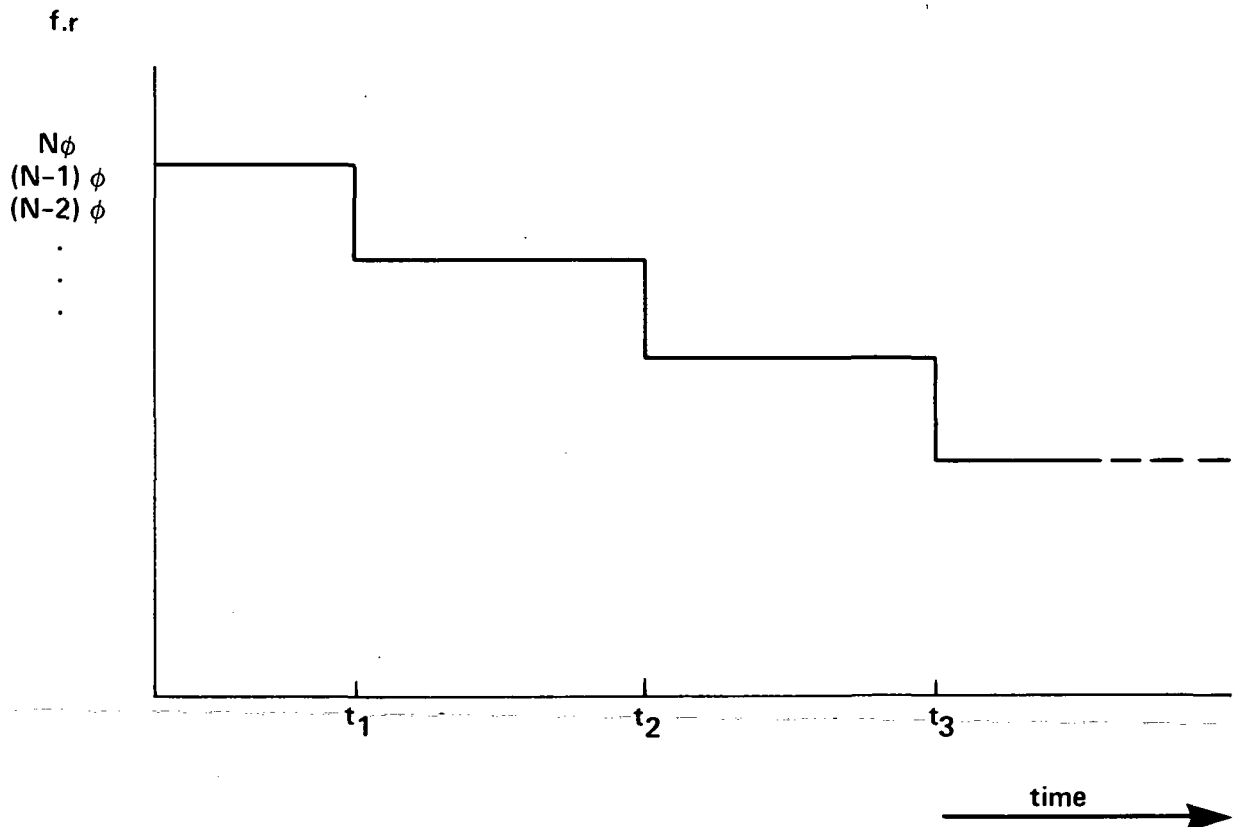
JELINSKI-MORANDA model assumptions:

1. Successive inter-failure times T_1, T_2, \dots are independent.

$$\text{pdf}(t_i | \lambda_i) = \lambda_i e^{-\lambda_i t_i}$$

2. $\lambda_i = (N - i + 1) \phi$ where

N is “initial number of faults” ϕ is “contribution to program failure rate from each fault”



Note that

1. All fixes have same effect.
2. Same model by SHOOMAN and MUSA. Same assumptions for NHPP model by GOEL-OKUMOTO.

There seems to be 3 problems with J-M:

1. \hat{N} occasionally infinite ($\hat{\phi} = 0$)

Nec. & Suff. conditions: "Regression line of t_i versus i has negative slope"
(Littlewood, Verrall: 1981 IEEE TR)

This can also occur with simulated data from J-M with finite N , $\phi \neq 0$,

However $\hat{\lambda} = \hat{N}\hat{\phi}$ is finite, non-zero.

2. Reliability predictions always(?) too optimistic
3. \hat{N} usually too small, sometimes equal to sample size (i.e. program is "perfect")

Table 7.
Failure Intervals – System 3 System Test Phase

i	T_i	
1	115,	1
2	0,	1
3	83,	3
4	178,	3
5	194,	3
6	136,	3
7	1077,	3
8	15,	3
9	15,	3
10	92,	3
11	50,	3
12	71,	3
13	606,	6
14	1189,	8
15	40,	8
16	788,	18
17	222,	18
18	72,	18
19	615,	18
20	589,	26
21	15,	26
22	390,	26
23	1863,	27
24	1337,	30
25	4508,	36
26	834,	38
27	3400,	40
28	6,	40
29	4561,	42
30	3186,	44
31	10571,	47
32	563,	47
33	2770,	47
34	652,	48
35	5593,	50
36	11696,	54
37	6724,	54
38	2546,	55
39	-10175,	56

SYSTEM 3

FAILURE NUMBER	\hat{N} ESTIMATED FAILURES	ESTIMATED INITIAL MTTF	$\hat{\phi}$ MORANDA PHI
2	999999	0.5750E+02	0.173913E-07
3	999999	0.6600E+02	0.151515E-07
4	5	0.5900E+02	0.338983E-02
5	6	0.6480E+02	0.257202E-02
6	8	0.7275E+02	0.171821E-02
7	7	0.7884E+02	0.181206E-02
8	8	0.8845E+02	0.141318E-02
9	12	0.1196E+03	0.696972E-03
10	19	0.1396E+03	0.377017E-03
11	55	0.1609E+03	0.112990E-03
12	999999	0.1688E+03	0.592304E-08
13	22	0.1387E+03	0.327621E-03
14	15	0.1125E+03	0.592367E-03
15	18	0.1306E+03	0.425447E-03
16	18	0.1306E+03	0.425294E-03
17	21	0.1476E+03	0.322715E-03
18	25	0.1616E+03	0.247463E-03
19	25	0.1622E+03	0.246615E-03
20	25	0.1612E+03	0.248210E-03
21	31	0.1807E+03	0.178535E-03
22	33	0.1854E+03	0.163413E-03
23	26	0.1609E+03	0.239046E-03
24	26	0.1606E+03	0.239457E-03
25	25	0.1520E+03	0.263205E-03
26	26	0.1628E+03	0.236199E-03
27	27	0.1764E+03	0.210001E-03
28	28	0.1876E+03	0.190384E-03
29	29	0.2023E+03	0.170456E-03
30	30	0.2182E+03	0.152766E-03
31	31	0.2427E+03	0.132935E-03
32	32	0.2642E+03	0.118265E-03
33	33	0.2853E+03	0.106202E-03
34	34	0.3041E+03	0.967196E-04
35	35	0.3248E+03	0.879556E-04
36	36	0.3519E+03	0.789439E-04
37	37	0.3804E+03	0.710397E-04
38	38	0.4073E+03	0.646041E-04



How well does model perform?

Simplest problem is estimation of current reliability:

Given data t_1, \dots, t_{i-1} , what can we say about T_i ?

What is cdf $F_i(t)$?

Obtain ML estimates of N, ϕ , based on t_1, \dots, t_{L-1} and use "Predictor distribution"

$$\hat{F}_i(t) = 1 - e^{-(\hat{N}-i+1)\hat{\phi}t}$$

If prediction is "good"

$U_i = \hat{F}_i(T_i)$ is approx.

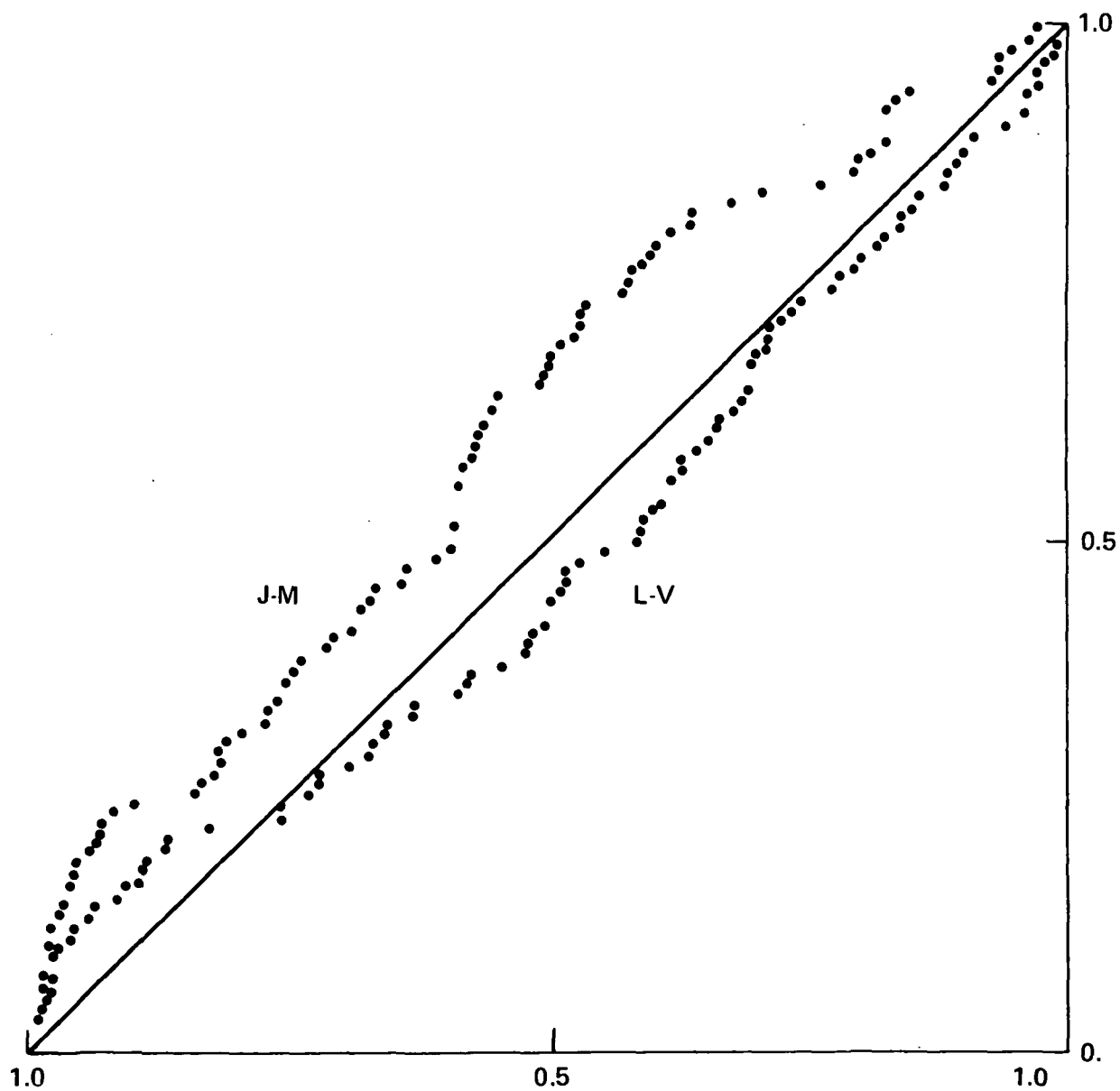
$U(0,1)$. Examine Q-Q plots of realizations U_i

EXAMPLE

Data: MUSA "System 1", range of i:30-129

Jelinski-Moranda: poor prediction, *optimistic*

Littlewood-Verrall: good prediction, slight pessimism



Bayesian J-M

Reparameterize to (λ, ϕ) from (N, ϕ) where $\lambda \equiv N\phi$ "initial failure rate".

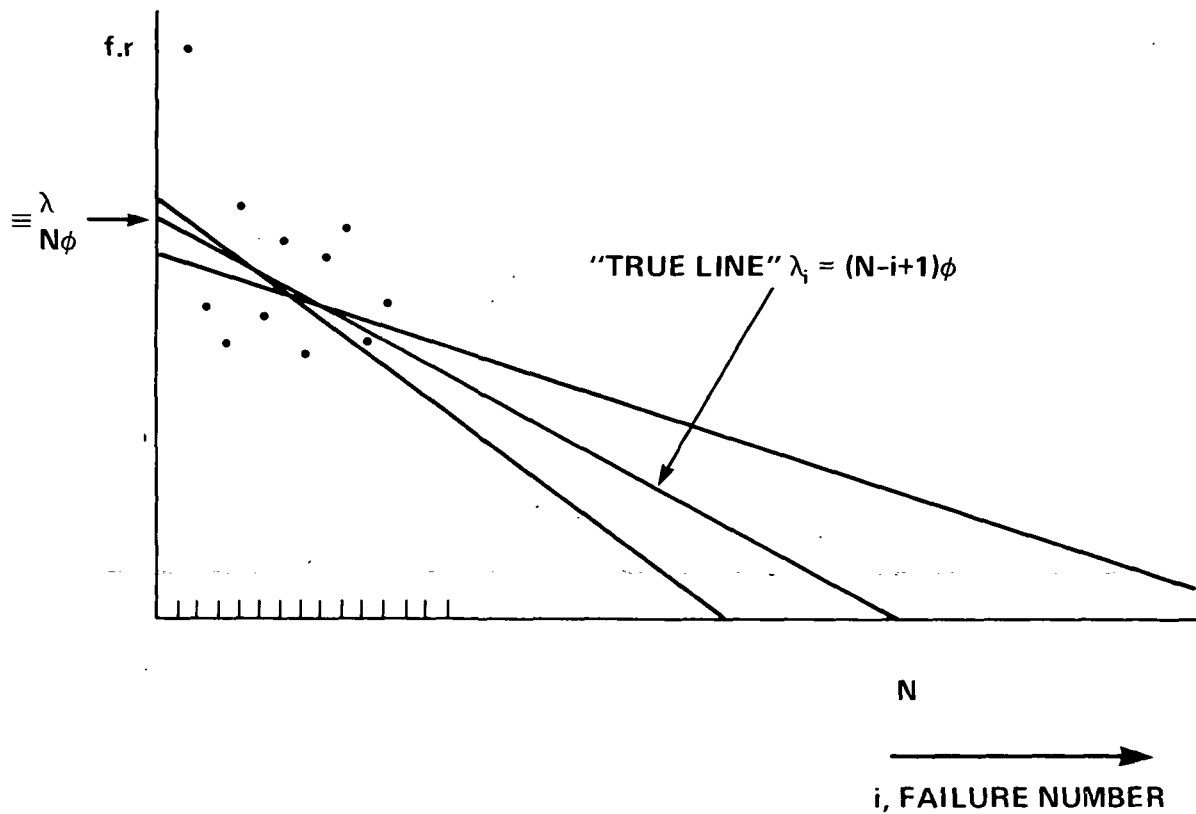
Assume:

$\text{prior}(\lambda, \phi) = \text{prior}(\lambda) \cdot \text{prior}(\phi)$ where $\text{prior}(\lambda)$ and $\text{prior}(\phi)$ are gamma distributed

Then "predictor distribution" is

$$\tilde{F}_1(t) \equiv P(T_i < t) \equiv P(T_i < t \mid t_1, \dots, t_{i-1}) = \int P(T_i < t \mid \lambda, \phi) \text{post}(\lambda, \phi \mid t_1, \dots, t_{i-1}) d\lambda d\phi$$

Reparameterization: Informal Justification



For the case of uniform (improper) priors we get:

$$F_{i+1}(t|t_i, \cdot, t_i) =$$

$$c \left[\sum_{k=0}^i \frac{a_{k,i} k! (i-k)!}{\left(\sum_{j=1}^i (i-j+1) t_j \right)^{k+1}} \left(\frac{1}{\left(\sum_{j=1}^i t_j \right)^{i-k+1}} - \frac{1}{\left(t + \sum_{j=1}^i t_j \right)^{i-k+1}} \right) \right]$$

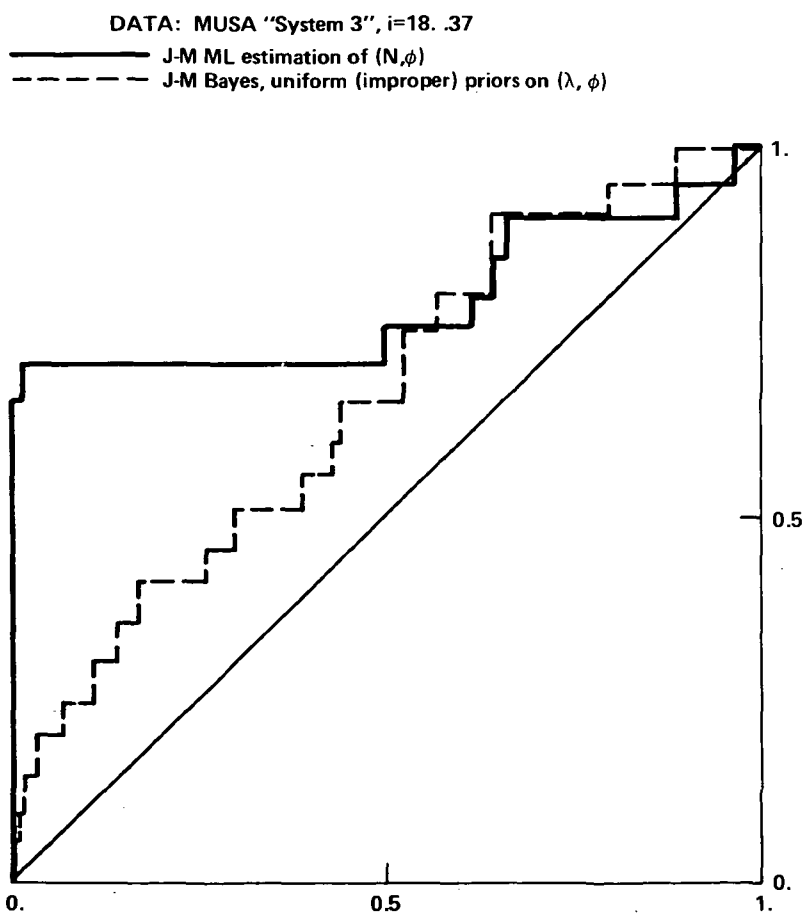
$$\text{where } c^{-1} = \sum_{k=0}^{i-1} \frac{a_{k,i-1} k! (i-k)!}{\left(\sum_{j=1}^i (i-j) t_j \right)^{k+1} \left(\sum_{j=1}^i t_j \right)^{i-k+1}}$$

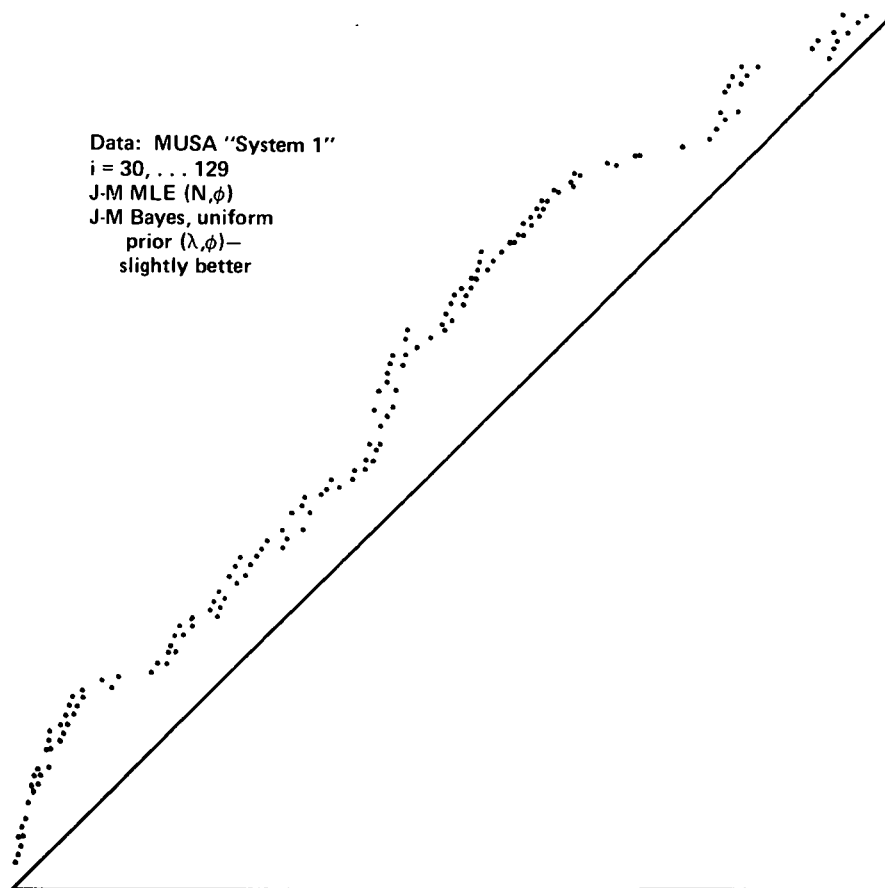
and where $a_{k,i}$ is the coefficient of x^{i-k} in $\prod_{k=1}^i (x+k) = \sum_{k=0}^i a_{k,i} x^{i-k}$

These coefficients are easily computed from the relation

$$a_{k,i} = i a_{k-1,i-1} + a_{k,i-1} \quad k \geq 1$$

$$a_{01} = 1 \quad a_{11} = 1 \quad a_{0i} = 1 \quad \forall i$$

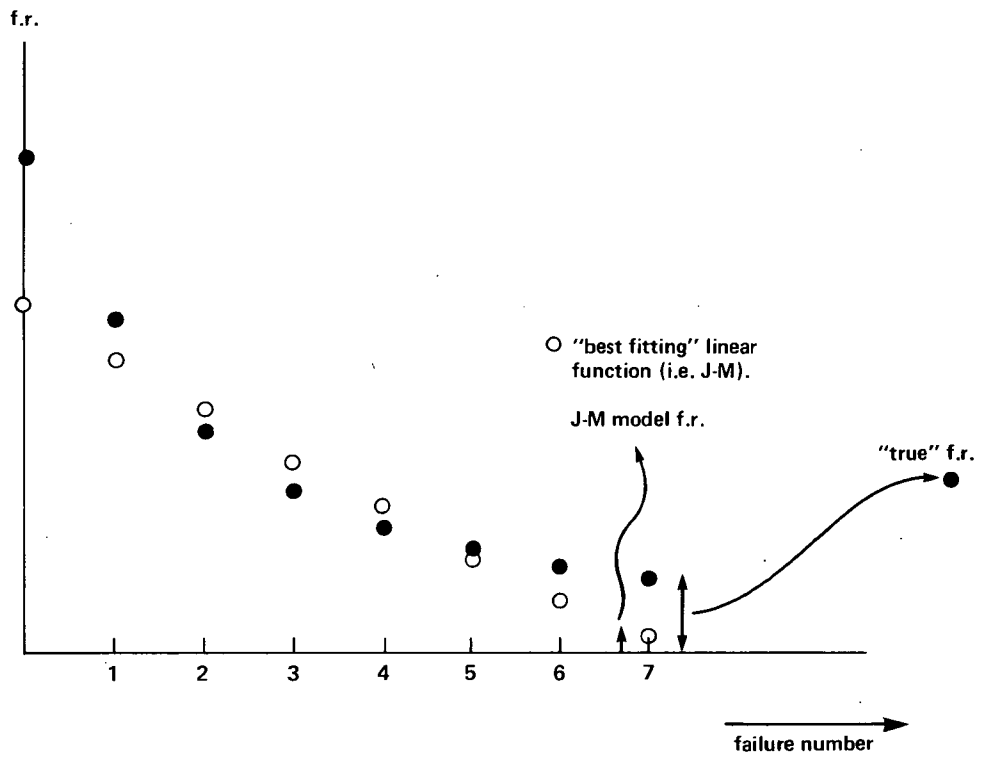




Conclusion!

1. Bayes J-M seems always (?) better than MLE J-M, but sometimes only slightly.
 2. Results on *real* data are always optimistic.
 3. But on SIMULATED data from J-M model, Bayes is very good, ML poor
- ⇒ real data do not follow J-M model?

Hypothesis: Assumption of equal ϕ 's is wrong. In fact ϕ 's different.
Larger ones tend to be eliminated earlier:



The Problem of Resonance
in
Technology Usage

OUTLINE

0. Introduction
1. Composite Case Study
2. Analysis of the Problem
3. Generalization of a Solution to the Problem
4. Conclusion

Hasan H. Sayani, Ph.D.

Cyril P. Svoboda, Ph.D.

Advanced Systems Technology Corporation
9111 Edmonston Road Suite 302
Greenbelt Maryland 20770
(301) 441-9036

ABSTRACT

Developers of information systems are bombarded with publicity releases hawking a plethora of tools and techniques. Although vendors give the impression that their product will lead to developer to the "promised land", they rarely are able to deliver. The result is that information systems developers ride a roller-coaster: rising to a peak of expectation and hope, only to plummet down the track of reality, before beginning to climb up to yet another peak of hope. This paper will analyze this situation from the authors' perspective, formed by using various information system tools/techniques and by consulting with over ten Fortune 500 firms and six government agencies.

A case study will be presented which draws together the issues raised in three distinct cases. Obviously, the names of the organizations will be changed as will any other information that might lead to identification. This case study will show a typical progression from the selection of an analysis methodology (SA) to the adoption of an automated tool for specification and documentation (PSL/PSA) and the difficulty of fitting these into an existing life cycle development methodology.

The problem presented in the case study is similar to the problem of resonance: over a period of time, the morale of system developers reels through a journey over peaks of "hyped" expectations and down into valleys of depressing realizations. In addition, management is weighed down with the pressures of short-term goals and the burdens created by long ignored human factors, both of which entice management to press for "any" product rather than the "right" product. The technology to which both developers and management often turn in desperation is marked by desperate development and by the shallow experience of the developers. Lastly, the mentality of those employing development tools and/or techniques is

very often provincial, relegating various items to a rigidly determined set of categories or hardware-driven.

The general approach to a solution is taken from a procedure for problem-solving developed by Svoboda and Sayani (1980). In this procedure, the system developer is encouraged to take time first to examine the problem before attempting to solve it, defining its major dimensions and determining the evaluative criteria to be used in assessing any proposed solution. Then the problem-solver uses some visualization tactic suited to his/her cognitive style or suggested by an organization's methodology. These visualizations are then elaborated on by translating them into linguistic expressions, at various levels of formality or precision. What is expressed needs to be reflected, so that the composer can grasp the implications of what has been said from various points of view, with a differing focus or scope. Although what has been said seems, on reflection, to be what was intended, it needs next to be analyzed or evaluated against the earlier determined criteria, in light of any constraints, within the scope of resources available. Those specifications which do not "pass" the foregoing evaluation must be modified and this expression-reflection-evaluation-modification process must be repeated until the system has been completely specified and is ready for construction and implementation. Before the development team congratulates itself for a job "well-done", it should project which tool/technique ought next to be selected and employed and what has been learned from the whole process of system development that might give direction to the next effort.

If an organization does not employ such an approach in systems development, it will eventually begin to experience the rollercoaster ride mentioned earlier. If one does employ such an approach, the organization will be in a better position from which to assess the intrinsic quality of its tools/techniques and their

contribution to the successful development of information systems. Such an approach would offer the basis for guiding an organization in the introduction, facilitation and institutionalization of new tools/techniques for the development of future information systems.

**THE VIEWGRAPH MATERIALS
for the
H. SAYANI/C. SVOBODA PRESENTATION FOLLOW**

The Problem
of
Resonance
in
Technology Usage

Presented at Sixth Annual NASA
Software Engineering Workshop
December 2, 1981

Hasan H. Sayani, Ph.D.
Cyril P. Svoboda, Ph.D.

Advanced Systems Technology Corporation
9111 Edmonston Road Suite 302
Greenbelt Maryland 20770
(301) 441-9036

Copyright © 1981 by Advanced Systems Technology Corporation (ASTEC),
Greenbelt, Maryland.

All rights reserved. No part of this material may be reproduced in any form
or by any means, without permission in writing from ASTEC.

CREDENTIALS

Corporate Objectives

- R & D in IS development process
- analysis, design, code generation and life cycle management automation tools
- engineering and human factors background
- application of tools on projects

Corporate Experience

- instruction and application of tools
 - 23 courses, seminars & workshops on PSL/PSA, methodologies, tools (ADL, ADS)
- consultation with organizations using tools (over 10 Fortune 500 & major Government Agencies) on all levels of organization
 - executive
 - management
 - operational
- evaluation of usage of tools

PREVIEW OF PRESENTATION

Composite Case Study

- examination of organization background in software development process
 - recognition of need for formal techniques
- response to problem
- result of piece-wise intro of tools

Analysis of Situation

Generalization Approach

Conclusion

COMPOSITE CASE STUDY

Examination of Background in Software Development Process

- third generation of hardware
- obsolete/poorly documented existing systems
- high turnover/additions to systems people
- dissatisfied users viewing systems as:
 - inadequate and costly
 - in large backlog/overruns
 - unintegrated
- lure of effortless development via tools and techniques
 - "let's get on some bandwagon"

RESPONSE TO PROBLEM

"Small is beautiful"

"Have Money - Will Buy Tools"

- one for each phase of development life cycle
- acquire tools
- train pilot group

RESPONSE TO PROBLEM

Apply the Solution

- result can range from
 - success to disaster

Next Evolutionary Step

- pass on work from one phase to another, or
- have a second group use the same tool
- both of which are usually doomed to disaster

Backlash

- build in-house
- force fit a tool by outspoken advocate
- regress

ANALYSIS OF SITUATION

Problem of Introduction

- reality rarely matches overall expectations
- never possible in isolation
 - distortion between existing and new techniques for each tool
- difficulty of integration across life cycle phases

ANALYSIS OF SITUATION

Management "Baggage"

- short term goals
- due-date versus quality
- ignoring human factors
 - career-path implications
 - E & T budget
 - management styles
 - authoritative
 - democratic
 - laissez-faire

ANALYSIS OF SITUATION

Technology Growing Pains

- first generation of tools/techniques
 - shallow experience
- vendor myopia and user passivity
- disparately developed
 - no overall plan of action
- changing ground rules
 - cost parameters (hardware/software ratios)
 - rapidly changing base technologies

DBMS

A-I

Graphics

ANALYSIS OF SITUATION

Field Immaturity

- failure to recognize commonalities
e.g., different types of systems
 - engineering vs commercial
- financial and legal community's effect
 - capitalization
 - protection (e.g., copyright/trade secrets)
 - inability to keep up with rate of change
- Governmental approach
 - doesn't foster coordinated effort

GENERALIZATION APPROACH

(Problem-Solving)

Problem Recognition

- postpone solution before understanding
- dimensions of problem
- developing criteria of judging solution

Visualization

- cognitive style
- methodology
- merely a basis for further work
- not universal

Expression

- graphics
- linguistic
 - levels of formality

Reflection

- other than mere echo of expression
- other focus, scope, dimension

GENERALIZATION APPROACH

(Problem-Solving)

Analysis/Evaluation

- comparing against criteria
- evaluate against constraints
- realization of resources available

Modification/Iteration

- sensitivity analysis
- impact projection

Solution

- determination } of product
- and }
- presentation }

Iteration

- where should next tool fit?
- what have we learned from experience?

CONCLUSION

- User organization: "get your house in order"
- Articulate needs of tools/techniques
- Set quality standards
- Evaluate existing tools/techniques
- Walk through whole development cycle scenario
- Introduce in a studied fashion
 - deliverables
 - career paths
 - feedback
 - support usage
 - training
- Study the process as well as the problem

PANEL #4

SOFTWARE METHODOLOGIES

H. Mills/M. Dyer, IBM
B. Jones, Hughes Aircraft Corporation
R. Hamilton, Bell Labs

Sixth Annual Software Engineering Workshop

Goddard Space Flight Center

December 2, 1981

Cleanroom Software Development

M. Dyer and H. D. Mills

The 'cleanroom' software development process is a new IBM technical and organizational approach to developing software with certifiable reliability. Key ideas behind the process are well structured software specifications, randomized testing methods and the introduction of statistical controls; but the main point is to deny entry for defects during the development of software. This latter point suggests the use of the term 'cleanroom' in analogy to the defect prevention controls used in the manufacture of high technology hardware.

The present state of the art in software development is to conceive and design a system in response to perceived requirements, then test the system with cases perceived to be typical to those requirements. The result is frequently a system which works well against inputs similar to those tested for, but one which is unreliable in unexpected circumstances. In fact, the evidence obtained by such testing is entirely anecdotal rather than statistical.

In the 'cleanroom', we embed the entire software development process within a formal statistical design, in contrast to executing selected tests and appealing to the randomness of operational settings for drawing statistical inferences. Instead, we introduce random testing as a part of the statistical design itself so that when development and testing is completed, statistical inferences can be made about the future operation of the system.

We believe there are several major benefits to such a procedure. One benefit is derived from standard statistical procedures in which a formal statistical design permits objective statements about properties of the system. But it is believed that an even more important benefit will arise from effects on the developers through the discipline of the statistical design on their activities. In fact, we believe that developing systems under stringent statistical controls will induce significant behaviour modifications on software developers.

Presently, when developers conceive early tests to check the correct operation of a system, they are able to identify just those parts of the system that will have to function correctly to pass those tests. Therefore, they can develop systems in phases, and control the testing such that the system under development is protected from unwanted testing. As a consequence, system parts may be omitted or done perfunctorily since the choice of tests is under the control of the developers.

We have in mind a different circumstance in testing under statistical control, namely, that from the outset tests are selected at random out of an expanding (top down) hierarchy of operational test cases. Therefore, the system designer must be prepared to deal with a growing, but always coherent, set of eventualities. It is believed that this circumstance, which may seem unfair or impossible at first glance, will dramatically change the way software development is done, by forcing a system approach top down rather than permitting bottom up pieces to be conceived and built under the protection of developer-selected testing.

THE VIEWGRAPH MATERIALS
for the
H. MILLS/M. DYER PRESENTATION FOLLOW

CLEANROOM SOFTWARE DEVELOPMENT PROCESS

DEFINITION

- TECHNICAL AND ORGANIZATIONAL APPROACH TO DEVELOPING SOFTWARE PRODUCTS WITH CERTIFIABLE RELIABILITY

LOGICAL EXTENSION OF

- SOFTWARE RELIABILITY THEORY
- MODERN SOFTWARE ENGINEERING PRACTICES
- FUNCTIONAL ORGANIZATIONAL STRUCTURE

GOALS

- PRODUCT RELIABILITY
 - INITIALLY ADDRESS PRODUCTS IN THE RANGE OF 10-25K SLOCS
 - RELIABILITY TARGETS OF MTBF'S MEASURED IN MONTHS AND YEARS
- STATISTICAL DESIGN
 - EXPECTATION OF CORRECT SOFTWARE DESIGNS
 - "BLACKBOX" TESTING OF SOFTWARE
 - TESTING FOR THE OPERATIONAL ENVIRONMENT
- PROCESS CONTROLS
 - SOFTWARE PRODUCT ENGINEERING FUNCTION
 - MANAGEMENT TO RELIABILITY COMMITMENTS

CLEANROOM SOFTWARE DEVELOPMENT PROCESS

RELIABILITY MODEL

- BASED ON SOFTWARE OPERATING FAILURES, NOT ERRORS IN THE CODE
- DIFFERS FROM HARDWARE MODELS, LOGICAL NOT PHYSICAL FAILURES
- REASONABLENESS DEMONSTRATED USING PUBLISHED SOFTWARE FAILURE DATA

STATISTICAL APPROACH

- INPUT/OUTPUT SPECIFICATIONS
- INPUT PROBABILITY DISTRIBUTIONS
- STOCHASTIC PROCESS INTRODUCED THROUGH RANDOMLY SELECTED RUNS
- MTBF STATISTICS DEVELOPED FROM CYCLE/FAILURE RATIO
- CERTIFICATION BASED ON FAILURE FREE EXECUTION INTERVALS, NOT ERROR FREE CODE

CLEANROOM SOFTWARE DEVELOPMENT PROCESS

CLEANROOM DEVELOPMENT METHOD

- STARTS WITH STRUCTURED SPECIFICATION
 - STATE MACHINE MODEL
- SOFTWARE DESIGN ENGINEERING PROCESS
 - MODERN DESIGN METHODS
 - FIRST TIME CORRECT PROGRAMS
- SOFTWARE PRODUCT ENGINEERING PROCESS
 - IDENTIFICATION OF PRODUCT INPUTS AND PROBABILITY DISTRIBUTIONS
 - SOFTWARE INTEGRATION INTO PRODUCT FORM
 - COLLECTION/CORRELATION OF FAILURE STATISTICS (MTBF)
 - CERTIFICATION TO CUSTOMER
- SOFTWARE MANAGEMENT
 - RELIABILITY COMMITMENTS
 - PRODUCT VISIBILITY THROUGH MTBF MEASUREMENTS

CLEANROOM SOFTWARE DEVELOPMENT PROCESS

DESIGN FUNDAMENTALS

- MODERN DESIGN METHODS
 - STATE MACHINES AND FUNCTIONS
 - STEPWISE REFINEMENT AND CORRECTNESS PROOFS
 - DATA TYPING AND ABSTRACTION
 - PROCESS DESIGN LANGUAGE (PDL) DOCUMENTATION
- MODERN IMPLEMENTATION METHODS
 - PROGRAM SUPPORT LIBRARIES
 - HIGH-ORDER PROGRAMMING LANGUAGES
 - STRUCTURED PROGRAMMING
 - REVIEWS AND INSPECTIONS

DESIGN INNOVATIONS

- STATISTICAL DESIGN APPROACH
 - DESIGN ALWAYS EXPOSED TO RANDOMIZED OPERATING INPUTS
 - EMPHASIS ON TOP-DOWN IMPLEMENTATION STRATEGY
- ELIMINATION OF SOFTWARE DEBUGGING
 - FOCUS TESTING ON OPERATING ENVIRONMENT
 - FOCUS DESIGN ON CORRECTNESS

CLEANROOM SOFTWARE DEVELOPMENT PROCESS

PRODUCT ENGINEERING STRATEGY

- CERTIFICATION BY INDEPENDENT GROUP
 - TESTING FROM SOFTWARE SPECIFICATION WITH DESIGN DETAILS HIDDEN
 - SEPARATION OF RESPONSIBILITIES AND INTERACTIONS
- TEST DEVELOPMENT
 - ANALYSIS OF INPUT PROBABILITY DISTRIBUTIONS
 - STATISTICAL/DISCRETE INPUT VALUES
 - INITIALIZATION AND OUTPUT VALUES
 - CONCURRENCY CONSIDERATIONS FOR PERFORMANCE TESTS
- TEST EXECUTION
 - SELECTION OF RANDOM INPUT SAMPLES
 - RECORDING OF FAILURE FREE EXECUTION MATERIALS
 - GENERATION OF MTBF STATISTICS
- FAILURE DIAGNOSTIC SUPPORT
 - FAULT LOCALIZATION
 - REGRESSION TESTING

CLEANROOM SOFTWARE DEVELOPMENT PROCESS

SOFTWARE DESIGN ENGINEER

- o CREATES THE PRODUCT
- o RESPONSIBILITY
 - IMPLEMENTATION OF AN APPROVED SPECIFICATION
 - DELIVERY OF CORRECT SOFTWARE TO THE PRODUCT ENGINEER
- o OUTPUTS
 - SOFTWARE PRODUCT DESIGN
 - SOFTWARE PRODUCT CODE
 - SOFTWARE PRODUCT DOCUMENTATION

SOFTWARE PRODUCT ENGINEER

- o CERTIFIES THE PRODUCT
- o RESPONSIBILITY
 - VALIDATION OF THE PRODUCT AGAINST THE SPECIFICATION
 - DELIVERY OF A CERTIFIED SOFTWARE TO THE CUSTOMER
- o OUTPUTS
 - SOFTWARE PRODUCT TEST PLANS/PROCEDURES
 - SOFTWARE PRODUCT INTEGRATION PLANS/PROCEDURES
 - SOFTWARE PRODUCT LIBRARIES
 - SOFTWARE PRODUCT TEST REPORTS

CLEANROOM SOFTWARE DEVELOPMENT PROCESS

TOOL REQUIREMENTS

- LIBRARY SYSTEM
 - DESIGN DOCUMENTATION
 - PRODUCT CODE
 - CERTIFICATION TEST SAMPLES
- STATISTICAL MODEL
 - MTBF CALCULATIONS
 - TREND ANALYSES
- SOFTWARE UTILITIES
 - TEST SAMPLE BUILD
 - TEST EXECUTION CONTROL
 - DATA COLLECTION/REDUCTION

SELECTING A SOFTWARE DEVELOPMENT METHODOLOGY

Robert E. Jones
Hughes Aircraft Company
Fullerton, CA

This paper describes the "Integrated Software Development Methodology (ISDM)" which is being accomplished by Hughes Aircraft Company, Software Engineering Division, in Fullerton, California and is sponsored by the Air Force Wright Aeronautical Laboratories, Flight Dynamics Laboratory at Wright Patterson AFB, Dayton, Ohio under Contract F33615-80-C-3614.

The ISDM project is currently in progress and its purpose is to study in detail state-of-the-art analytical techniques for the development and verification of digital flight control software and produce a practical designer-oriented development and verification methodology.

SCOPE

The scope of this project is limited to the study of existing tools and analytical techniques and the production of a practical ISDM guidebook. The methodology selected is adapted to flight control software, but is also applicable to most real time software developments.

The problem of evaluating the complete system is called validation, while the problem of checking the software at each stage of the design process is called verification. This project is concerned with verification.

The effectiveness of the analytic techniques chosen for the development and verification methodology will be assessed both technically and financially. Technical assessments analyze the error preventing and detecting capabilities of the chosen technique in all of the pertinent software development phases. Financial assessments describe the cost impact of using the techniques, specifically, the cost of implementing and applying the techniques as well as the realizable cost savings. Both the technical and financial assessment will be quantitative where possible. In the case of techniques which cannot be quantitatively assessed, qualitative judgements will be expressed about the effectiveness and cost of the techniques. The reasons why quantitative assessments are not possible will be documented.

BACKGROUND

The design of digital flight control systems has been the role of the control engineer rather than the computer or software specialist. Research into software design and verification has been the role of very specialized software experts. The results of this research have not always been practical in helping the flight control system designer with his tasks. Many tools and techniques are too complex to adapt to the flight control problem. Other tools are too expensive to maintain and operate for the flight control problem.

SUMMARY OF OBJECTIVES AND RESULTS

The objectives and results being discussed here reflect those individual objectives and accomplishments to date.

Metrics

The development of metrics which can be applied to assess the design quality was one of our first objectives. The effort was to be directed toward predictive metrics with the intention of producing metrics which can be used by a flight control systems engineer to determine the quality of the design produced and the likelihood of a successful implementation.

The metrics are being developed to aid in predicting such things as how many errors are likely, how long it will take to test, how long it will take to correct an error, etc.

One of the results is that a set of concepts which provide the foundation for the ISDM metrics has been developed. The equations which will be used as the basis of procedures to calculate predictors for the testability, reliability and flexibility have also been defined.

Guidebook

The overall objective is to create an integrated set of techniques and tools which are usable by a digital flight control systems engineer for development of a DFCS. Primary emphasis is to be on those activities involved in generating the DFCS software requirements specification, performing the software design, and verifying the software design through software integration.

The guidebook represents the bulk of the output from this project and will be the most visible. Emphasis must be placed on generating a document that is clear, understandable, and usable while fulfilling its intended role of a guidebook.

The results thus far have produced a draft guidebook that is ready to be applied during the experiment. The guidebook goes beyond the explanations of the tool and technique description and use. There are discussions regarding the development environment and major issues of DFCS software development. These are included to provide a backdrop for the actual application of the tools and techniques.

As a result of numerous reviews on various versions of the draft guidebook, there now exists a solid foundation from which to build. This building will occur as a result of the experiment. As different techniques are applied and as data is collected and analyzed, the guidebook will be updated. The guidebook will be maintained in a dynamic fashion, being changed as dictated by the experiment results.

Experiment

Having selected candidate analytic techniques and having organized these techniques into a guidebook, there remains the problem of objectively and quantitatively assessing the value of these techniques in producing a reliable flight control software system. For this reason, an experiment will be conducted in which a small sample flight control system will be developed using the ISDM guidebook.

The experiment will begin with the specification and progress through all software development phases. For each phase, an experiment will be conducted in which the analytic techniques and tools described in the guidebook will be applied. The resources expended in the application will be monitored and errors detected will be monitored and summarized.

In each of the development phases of the experiment, two classes of activity will take place. The first class of activities will be the actual application of the techniques in the ISDM guidebook to produce software. The second class of activities will be collection and analysis of the data pointing out the effectiveness of each technique, the impact of each technique on the overall schedule, the cost to prevent/detect errors, and the impact of errors on the total development effort.

Results thus far include the development of the experiment plan. This document is a detailed description of the activities which will occur. The plan includes the following factors to be considered in evaluating the guidebook:

1. Usability by a flight control engineer,
2. Cost to use,
3. Quality of the result and software.

The plan delineates the following data to be captured:

1. Errors,
2. Cost,
3. System documents,
4. Subject comments.

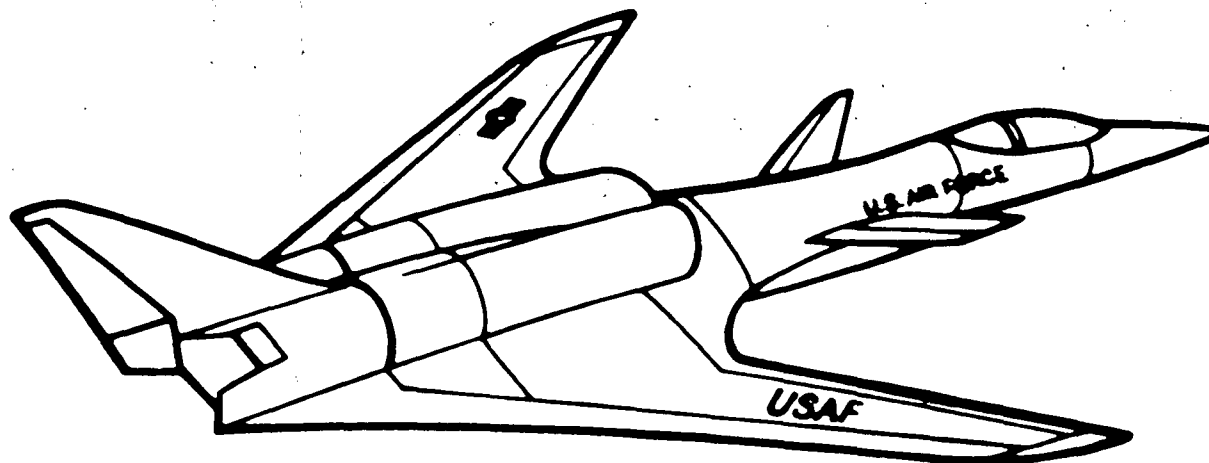
CONCLUSION

The ISDM project has just started in the second phase, the experiment. Although it is too early to provide firm conclusions, we are already starting to see some indications of not only which tools/languages may be useful, but also identify distinct weaknesses. The experiment will help to prove out these preliminary "feelings" and provide quantification, at least when applied to methodologies for specific applications.

**THE VIEWGRAPH MATERIALS
for the
B. JONES PRESENTATION FOLLOW**

HUGHES

INTEGRATED SOFTWARE DEVELOPMENT METHODOLOGY



R. Jones
Hughes
5 of 14

151154-33-(12-9-81)

GROUND SYSTEMS GROUP/FULLERTON, CALIFORNIA

ISDM GOAL

**"EVALUATE ANALYTIC METHODS FOR VERIFICATION OF
DIGITAL FLIGHT CONTROL SOFTWARE"**

HUGHES

RESTATED

**DEVELOP GUIDEBOOK FOR AN ISDM
(INTEGRATED SOFTWARE DEVELOPMENT METHODOLOGY)
AND QUANTITATIVELY EVALUATE THE EFFECTS ON COST
AND RELIABILITY OF USING ISDM FOR DEVELOPING DIGITAL
FLIGHT CONTROL SOFTWARE**

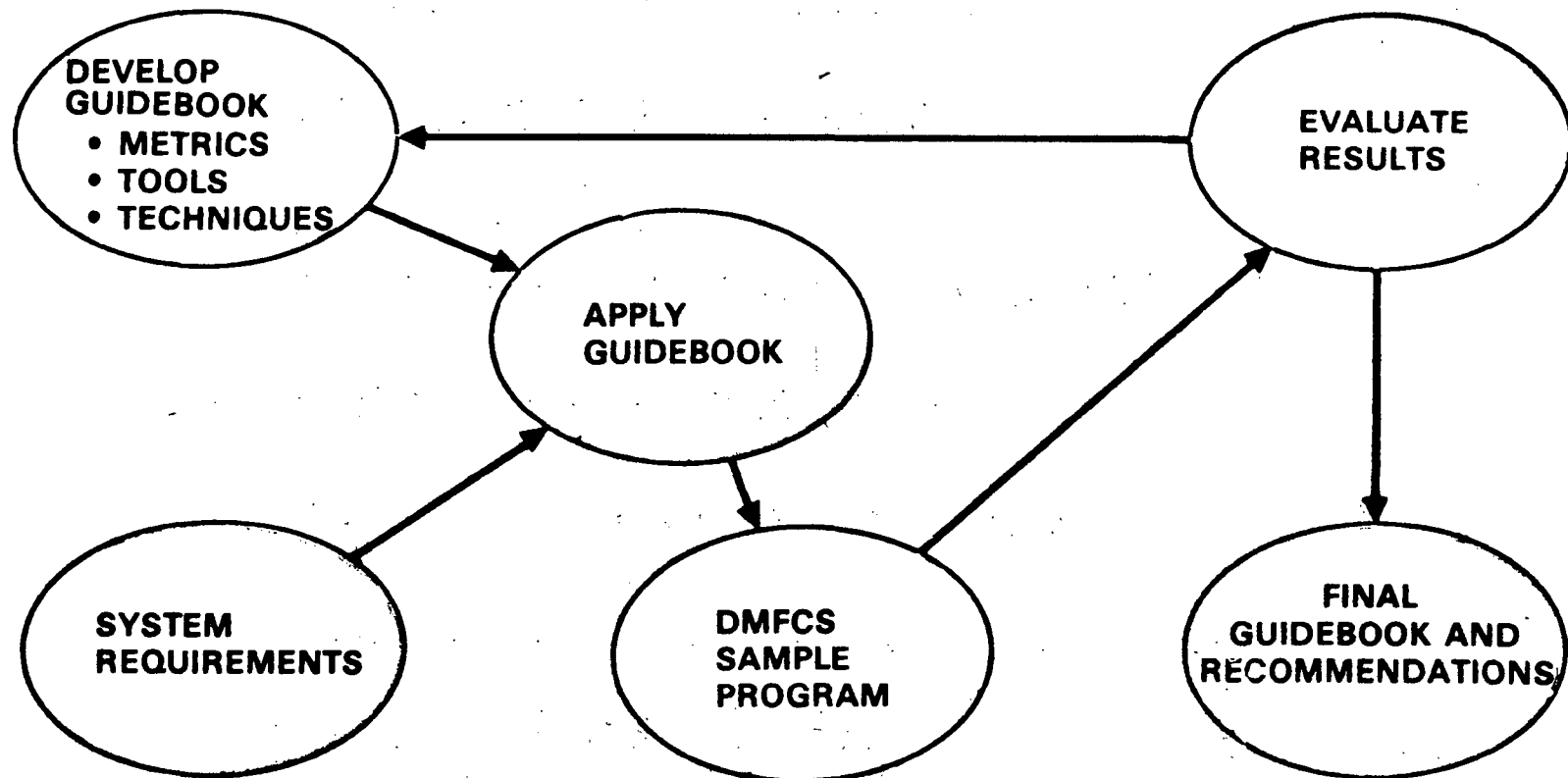
ISDM OBJECTIVES

HUGHES

- DEVELOP GUIDEBOOK FOR ISDM
- CONDUCT DFCS EXPERIMENT
- EVALUATE COST AND ERROR DETECTION EFFECTIVENESS
- DEVELOP DESIGN METRICS
- EVALUATE OVERALL ISDM COST AND RELIABILITY EFFECTIVENESS
- RECOMMEND ISDM USAGE
- RECOMMEND AREAS FOR FURTHER STUDY

PROJECT DESCRIPTION

HUGHES



ISDM GUIDEBOOKS CONTENTS PER PHASE

HUGHES

- **DESCRIPTION OF PROCEDURES**
- **DESCRIPTION OF SUPPORTING TOOLS AND TECHNIQUES**
- **TYPES OF ERRORS DETECTED**
- **REVIEW PROCEDURES**
- **INTERFACE WITH OTHER PHASES**
- **INTERFACE BETWEEN TOOLS AND TECHNIQUES**
- **RESOURCES NEEDED FOR EACH TOOL AND TECHNIQUE**
- **GUIDELINES FOR USING EACH TOOL AND TECHNIQUE**

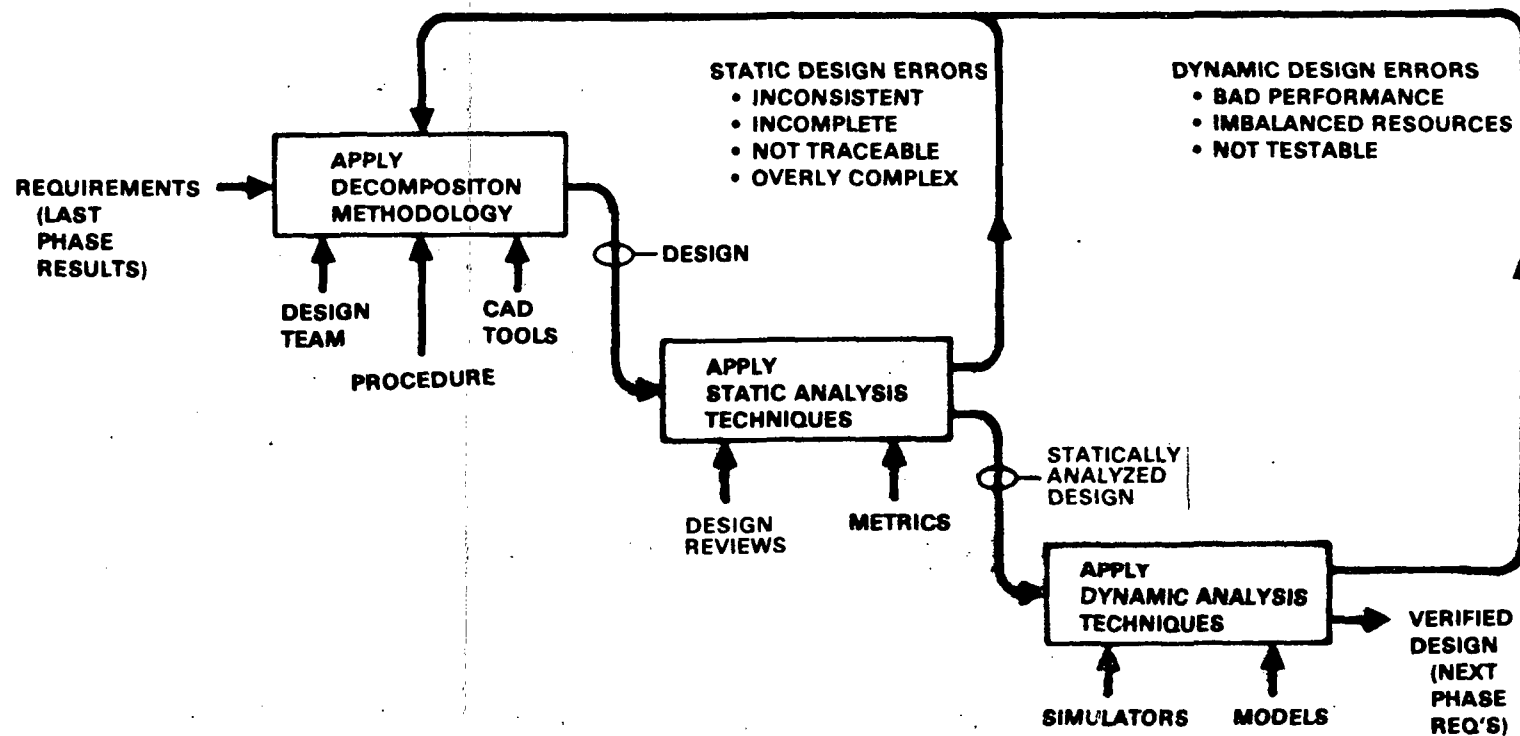
EXPERIMENT PLAN SUMMARY

HUGHES

- IMPLEMENT SAMPLE SYSTEM USING TOOLS/TECHNIQUES
- RECORD DAILY ACTIVITIES FOR COST ANALYSIS
- RECORD GUIDEBOOK COMMENTS FOR FINAL DRAFT
- RECORD ERRORS DETECTED AT EACH PHASE
- ANALYZE TOOL/TECHNIQUE EFFECTIVENESS

EXPERIMENTAL APPROACH

HUGHES



ASSESSMENT REVIEW

HUGHES

- **GUIDEBOOK USABILITY AND ACCURACY**
- **COST OF TOOL/TECHNIQUE APPLICATION**
- **QUALITY OF SOFTWARE/DOCUMENTATION**

RELIABILITY

SAFETY

TESTABILITY

MAINTAINABILITY

FLEXIBILITY

TOOL ANALYSIS

HUGHES

- **TRAINING COSTS**
- **GUIDEBOOK EFFECTIVENESS AND ACCURACY**
- **ERROR DETECTION CAPABILITY**
- **DOCUMENTATION SUPPORT**

PLANNED ACTIVITIES

HUGHES

- **COMPLETE EXPERIMENT**
- **EVALUATE RESULTS**
- **RECOMMEND IMPROVEMENTS**
- **TAILOR GUIDEBOOK**

Development Techniques for Generic Software

Richard L. Hamilton

Bell Laboratories
Holmdel, New Jersey 07733

1. INTRODUCTION

In developing the first version of a generic implementation of X.25, Levels 2 and 3, we examined three development techniques: table-driven finite state machine implementation, an integrated testing environment, and top-down design. While not designed as an experiment, we monitored the project closely and compared the product with other implementations of X.25 at Bell Laboratories to evaluate potential benefits and penalties.

2. TECHNIQUES

2.1 Finite State Machine

A finite state machine (FSM) is a powerful tool for both specifying and implementing protocols. This technique was used in the X.25 specification and has been discussed in the literature[1,2,3,4]. A table-driven implementation of the FSM was chosen to facilitate changes and simplify coding. We were interested in what effect this technique would have on program size, speed of execution, coding time, and debugging time.

2.2 Testing Environment

Contrary to common practice, we made a testing environment before coding. The complexities of a communications protocol, especially X.25, require careful attention to the problems of verifying that an

implementation of that protocol does in fact perform correctly. In addition, we felt that the process of verification should start as early as possible in the development process. The testing environment, which runs under the UNIX* operating system, let us test the FSM and its tables very early in the coding process. We were able to integrate new modules easily and test them thoroughly using this tool.

2.3 Top Down Design

In designing and implementing a solution, we followed a top-down approach. This made it possible to have a "running" version at all times, with unwritten modules replaced by dummy routines. This was not rigorously followed in coding because it was often more sensible to code all of the routines that performed one function even if that meant coding some low-level functions early. Doing this still let us always have a running version, but simplified testing.

3. MEASUREMENTS

Our main method for evaluating these techniques was comparison with existing implementations of X.25 at Bell Laboratories. We measured the size and execution speed of both our implementation and the existing ones and ran some simple complexity metrics.

* UNIX is a Trademark of Bell Laboratories

We used the testing environment to help modify and transport existing implementations of both Level 2 and Level 3 to a new environment, which gave us the opportunity to compare our versions with the existing ones in terms of the ease of making modifications. We kept a log of program bugs found and the effort it took to fix them, for all of the implementations, and tried to characterize the types of problems found.

4. CONCLUSION

A combination of a table-driven finite state machine realization, a comprehensive testing environment, and a top-down approach was used to produce an implementation of X.25, Levels 2 and 3. In comparison with other, ad hoc, X.25 implementations, we found that our solution ran as much as 20% faster, but was about 35 to 40 percent bigger. We were able to explain all but 11% of that difference in terms of added function or added flexibility. A McCabe complexity metric showed little difference between the implementations.

Comparison of time spent debugging showed that our approach was superior to the ad hoc methods, both in terms of number of errors detected and time taken to correct those errors. Even so, the testing environment was shown to be a significant aid in debugging the other implementations when compared to other testing techniques. Although not intended as a controlled experiment, the data collected during development support using these techniques in similar circumstances.

REFERENCES

- [1] Bochmann, Gregor V., "A General Transition Model for Protocols and Communication Services," IEEE Transactions on Communications, vol. COM-28, no. 4, April 1980.
- [2] Bochmann, Gregor V. and Tankoano Joachim, "Development and Structure of an X.25 Implementation," IEEE Transactions on Software Engineering, vol. SE-5, no. 5, September 1979.
- [3] Bochmann, Gregor V. and Carl A. Sunshine, "Formal Methods in Communication Protocol Design," IEEE Transactions on Communications, vol. COM-28, no. 4, April 1980.
- [4] Danthine, Andre A. S., "Protocol Representation with Finite-State Models," IEEE Transactions on Communications, vol. COM-28, no. 4, April 1980.

**THE VIEWGRAPH MATERIALS
for the
R. HAMILTON PRESENTATION FOLLOW**

DEVELOPMENT TECHNIQUES FOR GENERIC SOFTWARE

X.25 DEVELOPMENT

OBJECTIVES

TOOLS⁺

Portable

**C language, minimal
set of primitive functions**

Maintainable

**Testing/
development environment**

Flexible

Table-driven finite state

Modifiable

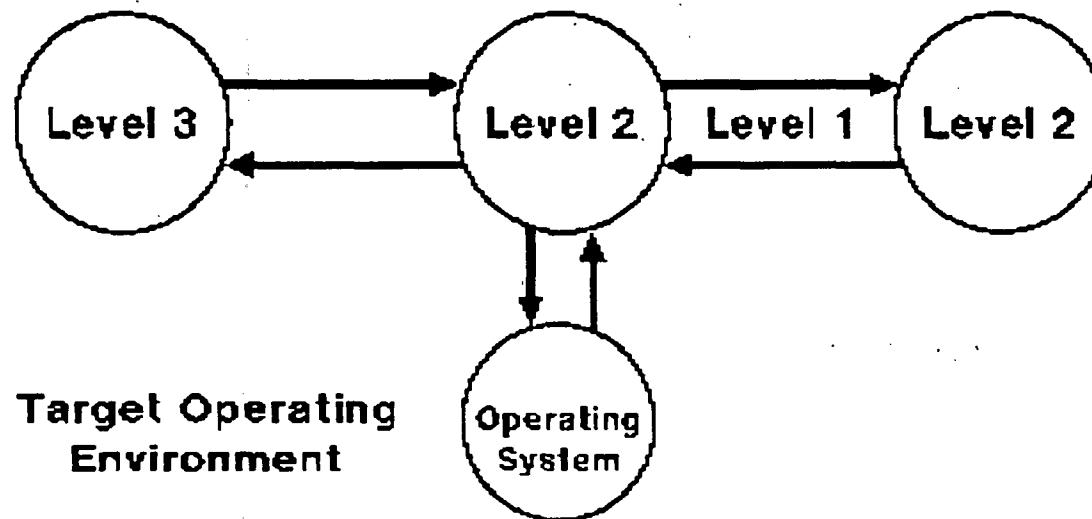
Layered approach

DEVELOPMENT ENVIRONMENT

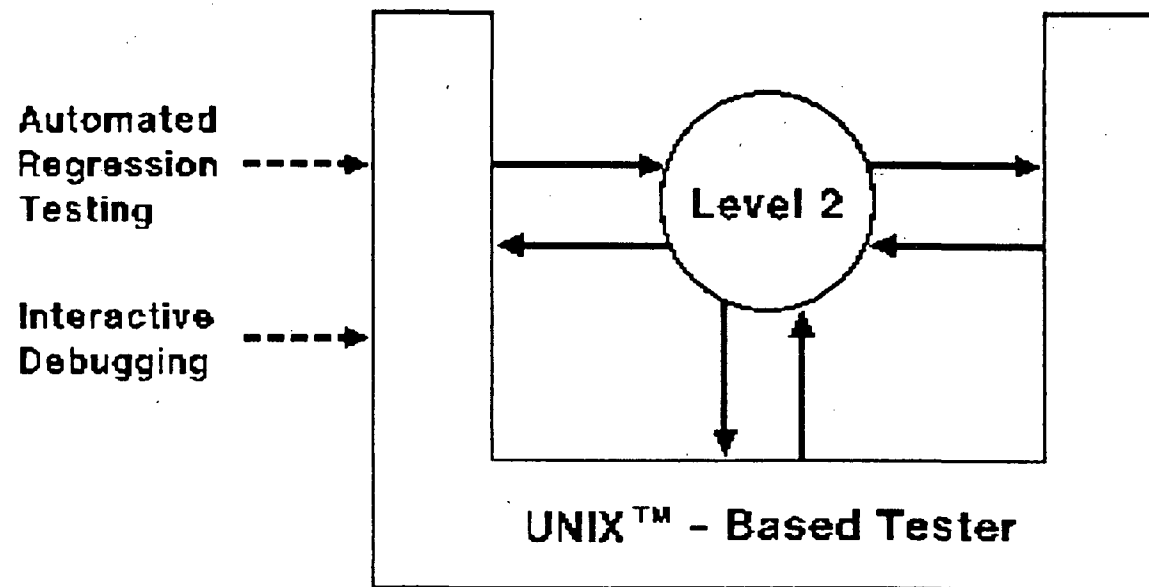
UNIXTM Operating System

- Make
 - AWK
 - SCCS
 - Shell

LEVEL 2 -- NORMAL ENVIRONMENT



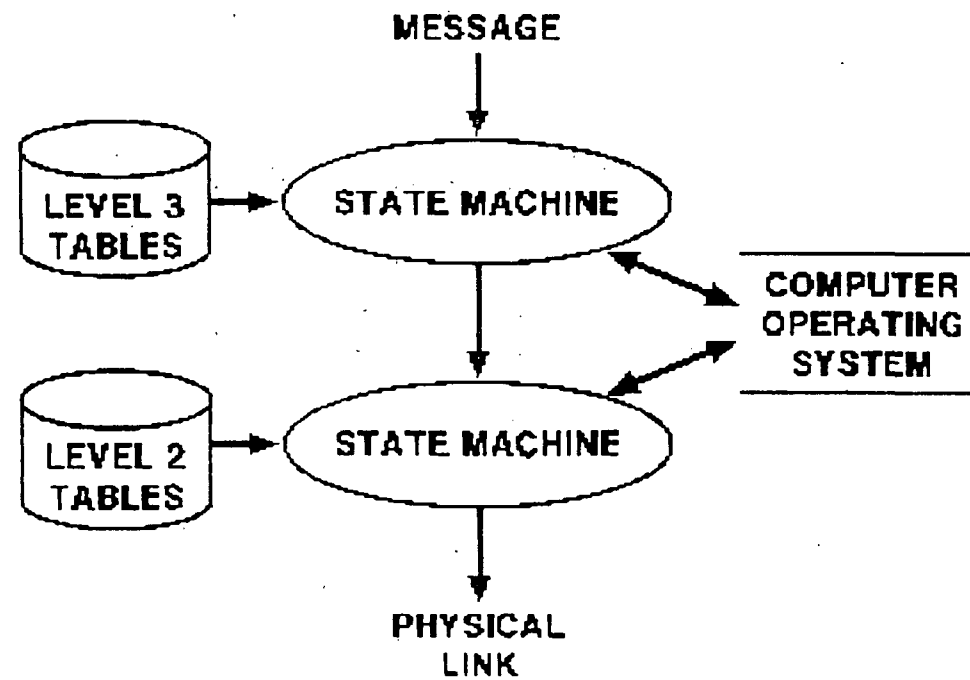
LEVEL 2 -- TESTING ENVIRONMENT



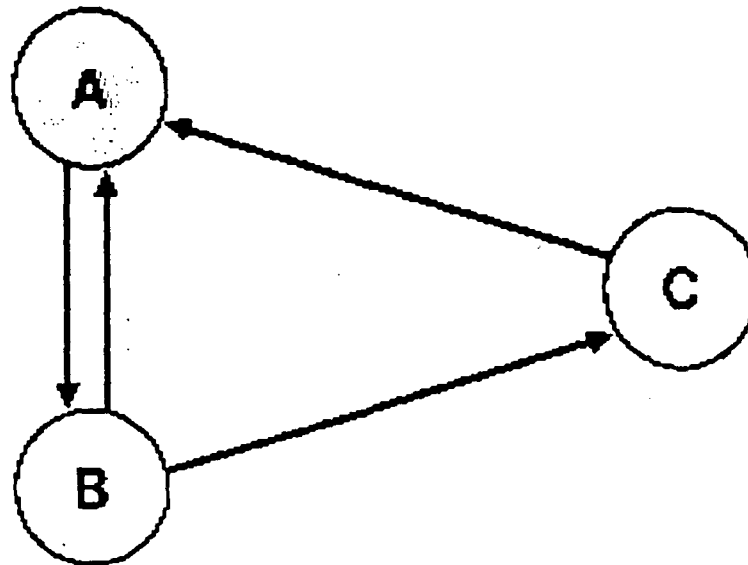
FINITE STATE MACHINE

- **Table-driven**
- **Hierarchical**
- **Parallel**

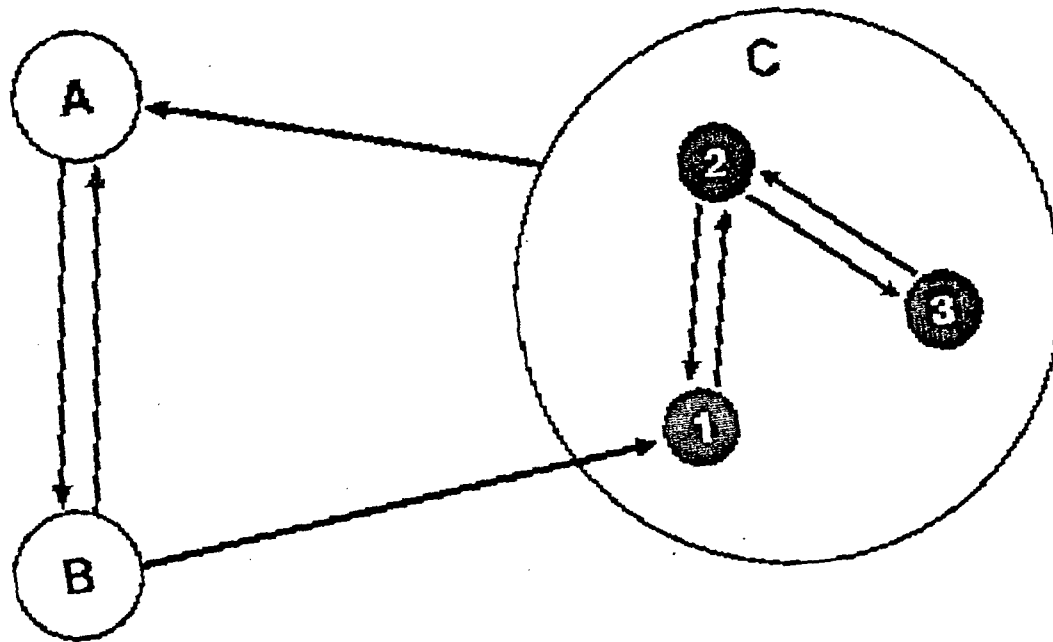
X.25 IMPLEMENTATION



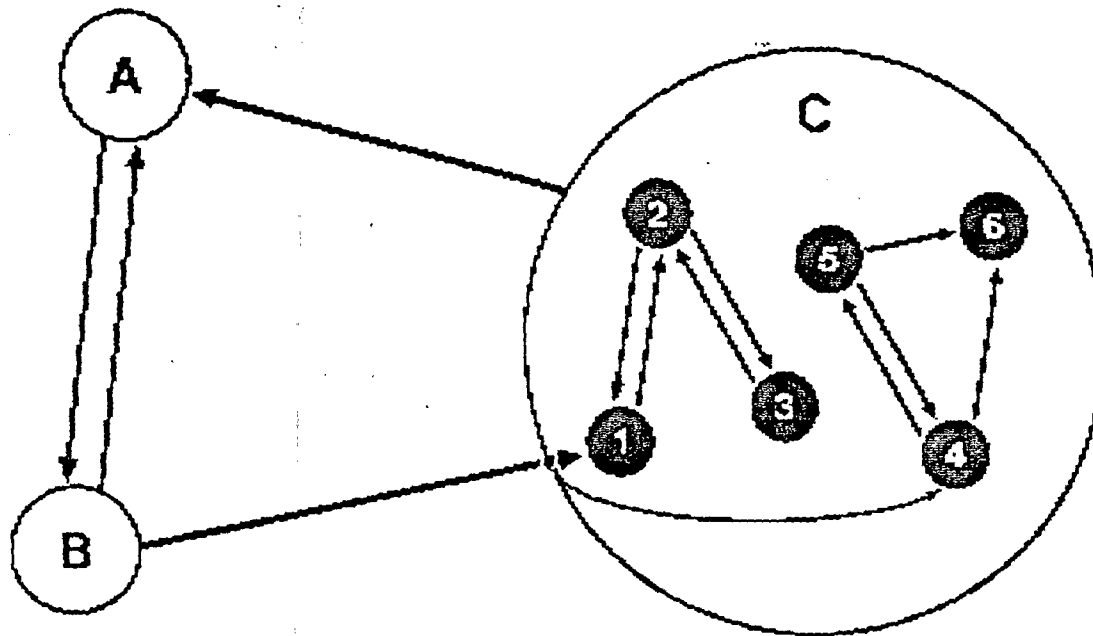
FINITE STATE MACHINE



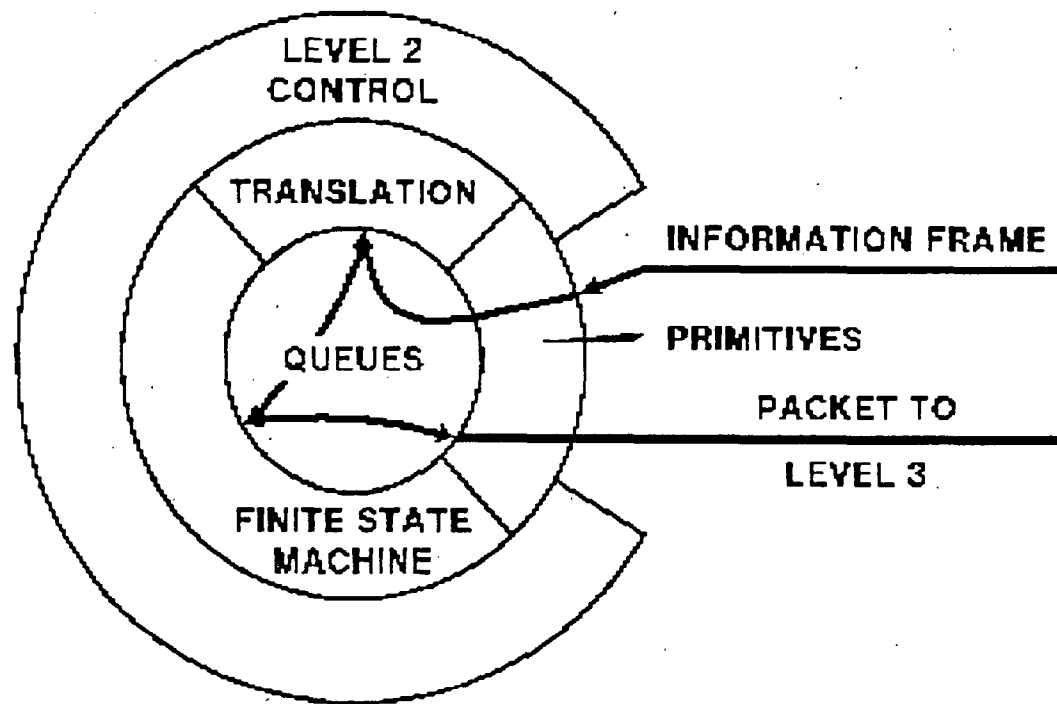
FINITE STATE MACHINE HIERARCHICAL STATES



FINITE STATE MACHINE HIERARCHICAL PARALLEL STATES



LAYERED STRUCTURE



LINES OF CODE

LEVEL 2	LINES OF CODE	% DIFFERENCE
---------	---------------	--------------

- Existing

1039

- Generic

1846

+78%

LEVEL 2	LINES OF CODE	% DIFFERENCE
---------	---------------	--------------

- Existing

1590

- Generic

2252

+42%

SIZE MEASUREMENTS IN BYTES

LEVEL 2	TEXT	DATA	=	TOTAL	% DIFFERENCE
• Existing	5688	56	=	5744	
• Generic	6766	1236	=	8002	+39%
LEVEL 3	TEXT	DATA	=	TOTAL	
• Existing	6818	268	=	7086	
• Generic	8558	926	=	9484	+34%

Note: All programs compiled under the 8086 cross-compiler with the optimize option, without primitives, and without any debugging aids included

LEVEL 2 SIZE DIFFERENCES

Added function

• Channel No.	200
• Timer routines	272
• Disconnect	186

Added flexibility

• Action overhead	248
• Channel select	52
• Multi-table FSM	200
• Table clarity	192
• Optional primes	100

TOTAL 1450

Actual difference 2258

Bytes unaccounted for 808

MEASUREMENTS

Size - 35 to 40% larger

Speed - 0 to 20% faster

Complexity - Equivalent

Alphabetical Listing of Attendees and Their Affiliations

Arnold, Robert	Univ. of Maryland
Bachman, Portia	NASA/GSFC
Bailey, John W.	GE
Barrett, Curtiss C.	NASA/GSFC
Basili, Vic	Univ. of Maryland
Batz, Joe	DOD
Bell, John F.	Action
Boggs, R. B.	NRL
Bond, Jack	NASA
Boone, Dave	CSC
Borochoff, Robert	Nat'l Lib. Med.
Boward, Stephanie	Sachs/Freeman
Bowe, Peggy	Lockheed
Brenneman, Dale	HUD
Card, David N.	CSC
Carpenter, Lloyd	NASA/GSFC
Carson, John H.	G.W.
Cephas, Arnold P.	NASA/GSFC
Cheuvront, S. E.	CSC
Chumura, Louis	NRL
Clarson, John	Stromberg-Carlson
Clements, Paul	NRL
Church, Vic	CSC
Cook, John	NASA/GSFC
Copperthite, Robert	Action
Corrigan, Paul	CTA
Cortez, Romo V.	NASA/HQ
Cruickshank, Robert D.	IBM
Cunningham, H. Conrad	Gen. Dyn.
Daniels, Herman	SASC
Boehm-Davis, Deborah	GE
Decker, William	CSC
Dickenson, Charles	USDA
Dinatale, Vincent	IBM
Diskin, Dave	Census Bureau
Duncan, Ray	CSC
Dyer, Michael	IBM
Eiserike, Howard	NASA/GSFC
Eng, Eunice	NASA/GSFC
Eslinger, Sue Ellen	CSC
Eisenhardt, George H.	Logicon Inc.
Fischer, Kurt	CSC
Forman, Ernest H.	G.W.
Fuchs, Art	NASA/GSFC
Gaertner, Ken	NSA
Gary, J. Patrick	NASA/GSFC
Giammo, Carol A.	DCA/CCTC
Goel, Amrit L.	Syracuse Univ.
Golden, John R.	Rochester Inst. Tech.
Goodson, Al	NASA/GSFC

Green, Art
 Green, Tony
 Grossman, Robert
 Hamilton, Richard
 Hanlin, Richard
 Hannan, Sue K.
 Hansan, Kevin
 Herring, Ellen
 Hilmer, Doug
 Hiller, Donald
 Hocking, Daniel
 Houghton, Raymond C., Jr.
 Howarth, Daniels
 Howell, Carol
 Hull, Larry G.
 Humphrey, William B.
 Hutchens, Dave
 Jamieson, Lillian
 Jones, Antonio L.
 Jones, Robert
 Judge, Robert
 Jun, Linda
 Kallmeyer, Fred W.
 Karl, John K.
 Kartatzke, Owen
 Kell, Veronica
 Kelly, A.
 Knaus, Rodger
 Knight, John C.
 Koschmeder, Lou
 Kruesi, Betsy
 Kurihara, Tom
 Kurzhals, Peter R.
 Kown, Y.R.
 Larson, Robert A.
 Leader, Karen
 Leibowitz, Steve
 Lichtenstein, Arleen
 Lin, Tsu H.
 Laubenthal, Nancy
 Maione, Anthony
 Mark, Marilyn
 Mazzuchi, Thomas
 McGarry, Frank
 McGarry, Mary Ann
 McPhee, John
 Medeiros, Edward J.
 Meick, Douglas
 Miles, Tim
 Mills, Harlan
 Mishoe, Jim
 Modlin, Mark

CSC
 NSA
 HUD
 Bell Labs.
 NASA/GSFC
 GE
 IBM
 NASA/GSFC
 Census Bureau
 Lib. of Congress
 AIRMICS
 NBS
 USDA
 NASA/GSFC
 NASA/GSFC
 DOTY
 Univ. of Maryland
 NASA/GSFC
 CSTA
 Hughes
 IBM
 NASA/GSFC
 NASA/GSFC
 NASA/GSFC
 NASA/GSFC
 NASA/GSFC
 INSCOM
 Nat'l Lib. Med.
 Univ. of Virginia
 NASA/GSFC
 GE
 DOT
 NASA/GSFC
 CSC
 USDA
 IITRI
 Lib. of Congress
 SDC
 USDA
 NASA/GSFC
 NASA/GSFC
 NASA/GSFC
 G.W.
 NASA/GSFC
 IITRI
 Dept. of Commerce
 CSC
 Lib. of Congress
 Dept. of Commerce
 IBM
 IITRI
 Social Security Adm.

Moe, Karen
 Mohanti, Siba
 Motley, Ron W.
 Musa, John
 Nadelman, Matthew
 Napjus, Chris
 Neill, David
 Nelson, Bob
 Neuwann, A. J.
 Oesterricher, Charles
 Oldson, Dennis
 Ondrus, Paul J.
 Ostrand, Tom
 Page, Jerry
 Parker, Donald
 Penny, Leonie
 Peters, Karl
 Phenneger, Milton
 Pietras, John
 Pinsky, Sylvan
 Plett, Michael
 Post, Jonathan
 Postak, John N.
 Posthuma, Bill
 Province, Phillip E.
 Ratte, George
 Redwin, Sam
 Reynold, Paul
 Roeder, John H.
 Rowe, William
 Rupolo, Vince
 Ryland, Jim
 Sandson, Mark
 Savolaine, Cathy
 Sauble, George R., Jr.
 Sayani, Hasan H.
 Scheffer, Paul
 Schlenoff, Marvin
 Schneck, Paul
 Schneider, Richard
 Schwenk, Bob
 Schultheisz, Robert
 Sheppard, Sylvia B.
 Selby, Richard
 Shimer, John
 Shukla, P.
 Siegel, Mark E.
 Singpurwalla, N.D.
 Sloger, Marcia
 Smart, Leslie
 Smith, Gene
 Smith, Luther G.

NASA/GSFC
 Mitre Corp.
 IBM
 Bell Labs.
 CSC
 NSA
 NASA/GSFC
 NASA/GSFC
 NBS
 Mitre Corp.
 Sachs/Freeman
 NASA/GSFC
 Sperry-Univac
 CSC
 NASA/GSFC
 USDA
 CSC
 CSC
 Mitre Corp.
 Social Security Adm.
 CSC
 Boeing Aerospace
 DOTY
 NASA/GSFC
 CSC
 USDA
 Mitre Corp.
 Univ. of Virginia
 NASA/GSFC
 Social Security Adm.
 Bankers Trust Co.
 Social Security Adm.
 CTA
 Bell Labs.
 NASA/GSFC
 ASTEC
 Martin Marietta
 Social Security Adm.
 NASA/GSFC
 NASA/GSFC
 NASA/GSFC
 Nat'l Lib. Med.
 GE
 Univ. of Maryland
 NSA
 CSC
 Dept. Media Lib. Instr'l Systems
 G.W.
 USDA
 Univ. of D.C.
 NASA/GSFC
 Fed. Reserve Bank of Richmond

Snyder, Glen
Sofer, Ariela
Gloss-Soler, Shirley
Sorkowitz, Alfred R.
Sos, John Y.
Soyer, Refik
Stanke, Edward C.
Starbird, Thomas
Stark, Mike
Stevenson, T.Q.
Suddith, Steve
Sokieski, Stanley
Sullivan, William
Svoboda, Cyril
Szulewski, Paul
Tesaki, Keiji
Tippett, James
Truss, Vivian
Truszkowski, Walt
Turner, Chris
Vandegrift, Shia Lu
Voight, Susan
Waligora, Sharon
Walton, Barbara A.
Wamser, Ray
Weaver, Alfred
Weiss, Dave
Will, Ralph
Williams, Clifford
Wong, Alice A.
Youman, Charles
Zelkowitz, Marv

CSC
G.W.
IITRI
HUD
NASA/GSFC
G.W.
Martin Marietta
JPL
NASA/GSFC
USDA
CSTA
NASA/GSFC
Dept. of Commerce
ASTEC
Draper Lab
NASA/GSFC
NSA
IITRI
NASA/GSFC
IITRI
USDA
NASA/Langley
CSC
NASA/GSFC
McDonald Douglas
Univ. of Virginia
NRL
NASA/Langley
DOTY
FAA
Cey Enterprises
Univ. of Maryland

BIBLIOGRAPHY OF SEL LITERATURE

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

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

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

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

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

Basili, V. R., "SEL Relationships for Programming Measurement and Estimation," University of Maryland, Technical Memorandum, October 1980

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

National Aeronautics and Space Administration, Special Publication, NASA Software Research and Technology Workshop, L. B. Holcomb and J. H. Bredekamp, March 1980

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

--, SEL-81-002, Software Engineering Laboratory (SEL) Data Base Organization and User's Guide, D. C. Wyckoff, D. N. Card, V. E. Church et al., September 1981

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

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

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

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

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

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

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

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

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

--, SEL-81-012, Software Engineering Laboratory, G. O. Picasso, December 1981

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

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

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

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

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

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