2

RL-TR-92-315
Final Technical Report
December 1992

AN APPROACH TO SOFTWARE QUALITY PREDICTION FROM ADA DESIGNS

The MITRE Corporation

W.W. Agresti, W.M. Evanco, M.C. Smith, and D.R. Clarson

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

Rome Laboratory
Air Force Materiel Command
Griffiss Air Force Base, New York

APPROVED: Andrew J Chruscicki

ANDREW J. CHRUSCICKI
Project Engineer

FOR THE COMMANDER [illegible]

JOHN A. GRANIERO
Chief Scientist for C3

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE December 1992	3. REPORT TYPE AND DATES COVERED Final Oct 89 - Sep 90

4. TITLE AND SUBTITLE AN APPROACH TO SOFTWARE QUALITY PREDICTION FROM ADA DESIGNS	5. FUNDING NUMBERS C - F19628-89-C-0001 PE - 62702F PR - MOIE TA - 71 WU - 30
6. AUTHOR(S) W. W. Agresti, W. M. Evanco, M. C. Smith, D. R. Clarson	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) The MITRE Corporation Washington C3 Center 7525 Colshire Drive McLean VA 22102-3481	8. PERFORMING ORGANIZATION REPORT NUMBER MTR-90W00135
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Rome Laboratory (C3CB) 525 Brooks Rd Griffiss AFB NY 13441-4505	10. SPONSORING/MONITORING AGENCY REPORT NUMBER RL-TR-92-315

11. SUPPLEMENTARY NOTES
Rome Laboratory Project Engineer: Anirew J. Chruscicki/C3CB/(315) 330-4476

12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited.	12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

An ongoing research project on estimating software quality from Ada designs is discussed. The research is motivated by the need for technology to analyze designs, when they are first represented, for their likely effect on quality factors. The objective of this research is to build multivariate models relating design characteristics and environmental factors to reliability and maintainability. Early results of the research are discussed, including alternative definitions of reliability and maintainability, a representation of Ada design structure, characteristics of software project data used for analysis, and preliminary statistical results testing hypotheses concerning the effects of design structure on reliability and maintainability.

14. SUBJECT TERMS Reliability, Maintainability, Software Design, Ada, Software Quality, Metrics, Measurement, Prediction	15. NUMBER OF PAGES 64
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL

ABSTRACT

An ongoing research project on estimating software quality from Ada designs is discussed. The research is motivated by the need for technology to analyze designs, when they are first represented, for their likely effect on quality factors. The objective of this research is to build multivariate models relating design characteristics and environmental factors to reliability and maintainability. Early results of the research are discussed, including alternative definitions of reliability and maintainability, a representation of Ada design structure, characteristics of software project data used for analysis, and preliminary statistical results testing hypotheses concerning the effects of design structure on reliability and maintainability.

KEYWORDS: software engineering, software measurement, software quality, software reliability, software maintainability, Ada

EXECUTIVE SUMMARY

This report describes preliminary results from an ongoing research project on software quality prediction from Ada designs. The research is supported by the Mission Oriented Investigation and Experimentation (MOIE) program of The MITRE Corporation.

The research objective is to establish a capability to project a software system's reliability and maintainability from an analysis of its design. The technical approach is to build multivariate models for estimating reliability and maintainability. Independent, explanatory variables in the models represent architectural design characteristics. Additional explanatory model variables are environmental factors to account for the effect of the organization and its development process.

Ada designs are analyzed because Ada is increasingly being used as a design language, thereby enabling automated design analysis. An investigation of Ada system structure yields a design representation consisting of compilation units and relations among them. Fifteen kinds of Ada compilation units and five relations are defined as building blocks of Ada systems.

Data from four projects have been analyzed thus far to test our hypotheses about various design structures leading to defect-prone or unmaintainable systems. The four projects total 183,000 non-comment, non-blank source lines of Ada, comprising 21 subsystems which are the units of observation in the analysis.

Design characteristic variables highlight the interconnection and resource-sharing among compilation units. Environmental factor variables provide for effects of reusability and software changes.

Reported results are preliminary because additional project data is being obtained and new hypotheses are being developed and tested. Multivariate regression analyses were conducted with different sets of variables. The resulting models explain 60-80 percent of the variation in reliability and 40-50 percent of the variation in maintainability. Reliability was measured as defect density; maintainability, as the percentage of simple defect corrections.

Research is continuing through developing new hypotheses and seeking additional data to enable new variables to be introduced into the models. Additional data will also lead to stronger statistical results which are needed to meet the objective of building useful quality estimation models.

ACKNOWLEDGEMENTS

The authors acknowledge the contributions to this research of Frank McGarry and Jon Valett of the Software Engineering Laboratory (SEL) at the National Aeronautics and Space Administration (NASA)/Goddard Space Flight Center (GSFC) in allowing us to use SEL project data. Review comments of MITRE colleagues, Frank Maginnis, Arlene Wusterbarth, Roger Duncan, Tom Smith, Brad Ulery, and Tom Walsh are appreciated.

TABLE OF CONTENTS

LIST OF TABLES

SECTION 1

INTRODUCTION

This report describes preliminary results from a continuing research project on software quality prediction from Ada designs. The project is supported under the Mission Oriented Investigation and Experimentation (MOIE) program of The MITRE Corporation.

This section discusses the background and objective of the research project, a perspective on software quality research, and the research approach being pursued in this project.

1.1 BACKGROUND AND RESEARCH OBJECTIVE

High quality software is essential to the success of mission-critical systems. Design characteristics are important determinants of this quality. Decisions made during design can affect software reliability, maintainability, flexibility, and other quality factors. A shortcoming of most large-scale software development projects is the lack of information concerning the consequences of these design decisions until much later in the development process. For example, it may become clear a system is not flexible in adapting to changing requirements only after extensive investment of time and effort to implement the design, and to integrate, test, and use the system. Inflexibility of such a system may be traced to design-structuring decisions made much earlier in the development process.

Greater capability is needed during the design phase to assess the design itself for indications that, when implemented, the resulting system will have particular quality characteristics. The government has an especially strong need for such early design assessment capabilities because it expects delivered systems to be reliable and supportable over long operational lifetimes. For example, maintainability, which may be measured by the ease of finding and repairing software defects, is critically important during the operational phase. However, maintainability is principally a design characteristic; that is, the ease of software maintenance is strongly determined by design decisions. Decisions about allocating functions and data to elements of the system affect the ease of repairing defects in the future.

The use of Ada for mission-critical software development presents an opportunity to more effectively evaluate software design. Prior to the development of Ada, design products were typically diagrams and text in documents. Assessing design quality required a labor-intensive examination of these documents to understand design relationships. For Ada projects, key design relationships can be represented in the language itself. The resulting Ada design representation is a new intermediate product, capturing software design relationships in the Ada language. Using Ada during design eases the transition from design to implementation, since the products of both phases are expressed in the same language. Also, the Ada design representation can be checked by an Ada compiler and can be analyzed by other Ada tools, offering the opportunity to automate part of the design evaluation process. This role for Ada as a design language has been recognized as American National Standards Institute (ANSI)/Institute of Electrical and Electronics Engineers (IEEE) Standard 990-1987 [1].

In this report, we discuss a research project on the evaluation of the quality of Ada-based designs. *An effective quality projection capability would enable a development team to begin corrective action immediately.* This research includes establishing relationships between design characteristics and the quality of the software product. We view this research as one thrust in a continuing effort to expand upon and refine software quality models.

1.2 SOFTWARE QUALITY RESEARCH

Software quality is the degree to which software possesses desired attributes. One such set of attributes, developed by Rome Air Development Center (RADC), includes reliability, efficiency, integrity, usability, survivability, correctness, maintainability, verifiability, portability, reusability, interoperability, expandability, and flexibility [2]. This composite nature of software quality encourages an approach to measuring it by addressing its individual attributes. Of these 13 attributes, reliability and maintainability are among the most critical constituents of software quality and the easiest to measure. Also, data for these quality factors is increasingly being collected on software projects. For these reasons, reliability and maintainability are the initial quality factors addressed in this research.

Reliability and maintainability are associated with events (system failures) and activities (correcting code) which occur after the implementation phase. Our perspective is that these events and activities are influenced by design characteristics. The thrust of our research is to investigate relationships of design characteristics to reliability and maintainability. If such relationships exist, then we will have the basis for projecting the potential quality of a software product in terms of characteristics that manifest themselves during design. We want to test the hypothesis that the quality of an Ada system can be predicted from analyzing its design.

Previous software quality measurement studies have proposed measures that relate design or code features to software reliability. Most of this work, however, has been focused on a single characteristic or oriented toward program modules rather than large software systems or subsystems as units of observation.

The cyclomatic complexity measure of McCabe has been correlated with software reliability [3]. Henry and Kafura showed the relationship of their data flow measure to reliability [4]. Extensions of these control flow and data flow metrics are reported in [5] and [6], respectively. More relevant to the current research, Card and Agresti defined a system-level design measure that incorporates control flow and data flow for FORTRAN designs and shows a strong correlation with reliability [7].

Nearest to this research in its objectives is the development of COQUAMO, the COnstructive QUAlity MOdel to estimate software quality [8]. COQUAMO is sponsored by the European Strategic Program for Research in Information Technology (ESPRIT). COQUAMO differs from our research by basing its estimate of software quality on the observed quality of previous projects implemented by the same organization. Our research is basing the estimate on the design itself: essentially projecting the quality of the complete system from the analysis of the design structure of that system.

1.3 RESEARCH APPROACH

Our approach differs from most previous work in two respects. First, we intend to build multivariate models that incorporate more than one design characteristic to explain a quality facto . Just as there are many dimensions to software quality, there is more than one dimension to characterize design complexity [9]. To the extent these design complexity characterizations are relatively uncorrelated, they will each contribute incrementally to the explanation of software quality.

Second, we recognize differences in the way organizations develop software. One organization may rely on software reuse more heavily than another. Organizations differ in the experience levels of their software developers and the effectiveness of their approaches to testing. Such factors, called *environmental factors*, influence software quality. Including environmental factors enables the models to describe a variety of software development organizations and processes, and, hence, have potentially broad applicability.

The general form of the multivariate quality estimation model, which we will reference in this report as the quality equation, is as follows:

$$Y = f\,(DC_1,\ DC_2,\ldots, DC_n, EF_1, EF_2,\ldots, EF_m | a_1, a_2,\ldots,a_p) + e \qquad (1)$$

where:

Y:	the specific quality factor (e.g., reliability or maintainability)
DC_i:	the ith design characteristic variable ($i=1,\ldots,n$)
EF_i:	the ith environmental factor variable ($i=1,\ldots,m$)
a_i:	the ith parameter ($i=1,\ldots,p$)
e:	disturbance term for unexplained variation.

The parameters in the quality equation are estimated from the empirical data by multivariate techniques such as linear regression analysis or maximum likelihood estimation. Not all variations in a quality factor are explainable by the variables DC_i and EF_i included in the equation. The disturbance term, in effect, reveals the extent to which the model variables do not explain software quality.

Four steps are involved in building this software quality model: specification, statistical estimation, verification, and application. The specification step involves a statement of the structural form of the model. A specific functional form of the quality equation is identified, relating the design and environmental variables to the parameters. Typical functional forms for regression analysis are linear and logarithmic. More complicated functional forms may require the use of non-linear methods.

The statistical estimation step determines specific values for the parameters in the quality equation. Most approaches for estimating these parameters are based either on least-squares or maximum likelihood techniques. Empirical data must be collected and processed for statistical analysis. The estimation step involves measures of overall goodness of fit for the equation, as well as measures of the standard errors of the individual parameter estimates.

The verification step is concerned with decision rules to determine the success of the model. Accuracy of the model in forecasting the quality of new systems (prediction) is important. Alternatively, understanding the effect of changes in specific design characteristics on quality (prescription) is another way to use the model. In either case, criteria for acceptance or rejection of the model must be established and applied.

The application step concerns the use of the model in a context that was not used to estimate its parameters. If the parameters were estimated on the basis of an ensemble of projects, then the model should be applied to new projects to validate its predictive or prescriptive capabilities, and to understand its limitations when applied to new environments.

This four-step approach to model building is being pursued in this research. The preliminary results of this approach are presented in section 4.

SECTION 2

RELIABILITY AND MAINTAINABILITY MEASURES

As a first step in building multivariate models in the form of the quality equation (introduced in section 1), we established definitions for measures of reliability and maintainability. We allowed several definitions for each quality factor to account for the diverse availability of data in organizations in which the models may be used.

Several measurement issues were addressed in the process of identifying appropriate definitions. One issue is the trend toward user-oriented rather than developer-oriented measures. An example is the use of failure-related measures of reliability instead of defect-related measures. Failure-oriented measures are preferred because they express reliability in terms of users' satisfaction with the operation of the software [10]. Defects are manifestations of errors made by the developer: still of interest but not directly addressing user experience. A second issue is the time frame during which the measures apply: before the release of software for production (pre-release) or after its release (post-release). Failure data is most often collected post-release, although data may be gathered during testing, in which the post-release environments are simulated. Defect data is more generally available and collected pre-release. A third issue is the recognition that both calendar time and execution time are possible measures of the time period over which reliability is expressed. Definitions must be explicit on the time measure used. Musa et al. [11] discuss the relationship between calendar time and execution time for software.

We recognize that the reliability and maintainability measures in this section are not complete. An abundant literature exists in software measurement (see, for example, [11], [12], and [13]). We regard the collection of measures presented here as a "working set," open to expansion over time. Three measurement trends not represented in the current working set are (1) composite metrics (e.g., [14]), (2) metrics that stratify errors and changes, for example, by severity [15], and (3) relative metrics. To sidestep the issue of size or time being the more appropriate normalization factor, relative metrics compare two quantities with identical units. A relative metric for maintainability is the ratio of maintenance productivity to development productivity. Measures reflecting these three issues are certainly candidates to enter the working set in the future.

Our current research focuses on only a few of the measures cited below.

2.1 RELIABILITY MEASURES

Reliability is understood informally as the confidence that software will perform without failing. More formally, reliability is defined in ANSI/IEEE Standard 982.2-1988 as the probability that software will not cause the failure of the system over a specified time interval under specified conditions [1].

Reliability measures span a spectrum that bases definitions successively on failures, faults, defects and errors. In this sense, the definitions range from user-oriented (based on failures) to developer-oriented (based on errors).

For our research, we identified seven reliability measures, as follows:

- R1: Probability of failure-free operation over a specified time interval in a specific environment--e.g., R(8) = 0.94 means "the probability is 0.94 that the software will operate without failure over the interval 0-8 (hours)".

- R2: Failure rate--number of failures (e.g., software "problems" reported) per unit of time, e.g., 0.01 failures per hour of execution time.

- R3: Time between failures--elapsed time between successive failures, often expressed as the mean or median of such times. This measure is the inverse of R2.

- R4: Fault density--number of faults divided by software size, e.g., 0.5 unique faults per thousand source lines of code (KSLOC).

- R5: Defect density--number of defects divided by software size, e.g., 0.8 unique defects per KSLOC.

- R6: Error density--number of errors divided by software size, e.g., 0.4 errors per KSLOC.

- R7: Reliability rating (from [2])--errors per line of code, subtracted from unity, e.g., 3 errors per KSLOC yields a reliability rating of 0.997 (= 1 - [3/1000]).

We acknowledge taking liberty in identifying R4 through R7 as measures of reliability. If reliability is defined in terms of failures, R4 through R7 are measuring reliability-related quantities.

Measures R1 through R3 depend on accurate failure occurrence data. These three measures are more clearly defined by using computer execution time. If calendar time is used instead, people using the measures may not be aware of the assumed usage level of the system over the calendar time period. Also, failures typically refer to system failures in a hardware/software system. Our focus on software reliability causes us, for a hardware/software system, to concentrate on software-induced failures in the system.

Measures R4 through R6 are distinguished by their reliance on three frequently confused terms: fault, defect, and error. We use the following ANSI/IEEE Standard 982.2-1988 definitions [1]:

- Fault: an accidental condition that causes a functional unit to fail to perform its required function; a manifestation of an error in software.

- Defect: a product anomaly; examples include such things as (1) omissions and imperfections found during early life-cycle phases and (2) faults contained in software sufficiently mature for test or operation.

- Error: human action that results in software containing a fault.

These definitions illustrate that measure R4, relying on faults, cannot be as broadly applied throughout the life cycle as measures R5 and R6. Data for R4 depends on software being developed and available for testing. R5 and R6 can be computed using data on defects and errors associated with pre-implementation activities like design.

Although failure-based, post-release measures (e.g., R1, R2, and R3) may be preferred, necessary data may not be available to support their use. The alternative of applying fault, defect, and error-based measures before release still may be suggestive of post-release experience. Empirical studies (e.g., [12]) have shown positive relationships between defect densities measured pre-release and post-release.

The reliability rating, R7, derives from the software quality framework of Rome Air Development Center [2]. Like R6, measure R7 uses error data; however, R7 produces a rating in the range of zero to one. Also, unlike R2, R4, R5, and R6, higher values of R7 correspond to higher reliability.

The reliability aspects of this study will focus on measure R5.

2.2 MAINTAINABILITY MEASURES

Maintainability may be defined informally as the ease with which software can be maintained. We are viewing maintainability as referring exclusively to the ease of isolating and correcting faults. Our focus is on corrective maintenance rather than adaptive or perfective maintenance [16]. This view is consistent with the software quality framework in [2], in which the separate quality factors of expandability and flexibility refer to the ease of enhancing the software.

We identified eight maintainability measures. Four measures are based on the effort required to perform maintenance. Three other measures use, as a proxy for effort, the number of modules examined as well as changed, and the lines of code changed. The last measure focuses on the degree of success in performing maintenance actions. The eight maintainability measures, for purposes of our research, are as follows:

- M1: Effort to correct--mean number of staff hours required to isolate the cause of a failure, to develop necessary corrections, to conduct any necessary unit and regression tests, to install the corrections, and to update all relevant documentation.

- M2 Time to repair--number of wall-clock hours required to perform a corrective maintenance task, often expressed as the mean or median of such times.

- M3: Maintainability Rating (from [2])--one-tenth of the mean number of staff days to perform a corrective maintenance task, subtracted from unity, e.g., if it requires 2 staff days, on average, to perform corrective maintenance, the maintainability rating is 0.8, if greater than 10 days are required, the maintainability rating is zero.

- M4: Easy fix frequency--when staff effort to isolate and repair a fault is available as ordinal data (e.g., easy, medium, hard), this measure is the percentage of corrections in the desirable (lowest effort) category.

- M5: Module handling impact--mean number of modules examined while performing a corrective maintenance task. A variation of M5 is the fraction (of total modules) examined.

- M6: Module change impact--mean number of modules changed while performing a corrective maintenance task. A variation of M6 is the fraction (of total modules) changed.

- M7: Code change impact--mean number of source lines of code added, deleted, or modified during a corrective maintenance task. A variation of M7 is the fraction (of total source lines of code) added, deleted, or modified.

- M8: Miscorrection frequency--the percentage of corrective maintenance actions that caused new faults to be introduced, or failed to correct the original fault.

Measure M1 is frequently used because of a high availability of supporting data and its interpretation of ease of maintenance as staff effort [17,18].

Measure M2, especially as mean time to repair (MTTR), is used in hardware maintainability analysis. It requires a history of failures and an accurate log of the time to repair the faults causing the failure. As Sunday [19] confirms, this information is typically not available. MTTR is most relevant for software when the integrated hardware-software system is counted on for very high availability such as in high-volume transaction processing or reservation systems. MTTR is the mean time to restore a system to an operational state after a software failure renders the system non-operational [20].

The maintainability rating M3 parallels R7 in its calculation of a measure in the range zero to one [2]. Also, higher values of M3 (and M4) correspond to higher degrees of maintainability.

While measure M4 (easy fix frequency) is also based on staff effort, it reflects a difference in measurement scale. The collection instrument for maintenance data may be a form completed by a maintainer. To simplify completion, the form may require the maintainer only to indicate effort by checking a box corresponding to easy, medium, and hard or similar ordinal ranking. The maintenance aspects of this study will focus on measure M4.

Measure M5 (module handling impact) is derived from Belady and Lehman's study of the evolution of OS/360 [21]. For successive OS/360 releases, they measured the number of modules handled; that is, examined for possible change and, if necessary, changed. The rationale for M5 is that more difficult maintenance corresponds to handling more of the system. The same rationale underlies measures M6 (module change impact) and M7 (code change impact). Greater effort is expected to be reflected in more modules and code being changed. Measures M6 and M7 are frequently recommended for maintainability [12]. Both measures (and M5 as well) have a benefit of potentially easier data collection. Configuration management systems may provide automated support for calculating these measures, while measures like M1 may require manual collection of effort data. Schaefer observes also that measures M4, M5, and M6 have an additional benefit of not depending as much on staff experience as do maintainability measures based on staff effort [22].

Measure M7, miscorrection frequency, focuses on a different aspect of maintenance: Did the maintenance action introduce new errors or fail to correct the original fault? An indication of maintainable software is a high percentage of maintenance actions successfully repairing reported faults without introducing new ones.

SECTION 3

DESIGN HYPOTHESES AND STRUCTURE

An important aspect of our research is relating design decision-making to design artifacts and, ultimately, to reliability and maintainability. This section discusses these relationships and how they suggest hypotheses about variables to enter our models. Ada design structure is studied to learn how the design artifacts will appear. The analysis leads to an Ada design representation which serves as a basis for defining design characteristics for our model-building activity.

3.1 DESIGN HYPOTHESES

We are developing hypotheses that particular design patterns relate to reliability and maintainability of the implemented system. Our view is that these design patterns are the result of design decisions. In this sense, our hypotheses encompass design decision-making as well as the resulting artifacts. In considering design decisions, we are not investigating the consequences of using particular design methods like structured design or object-oriented design. Instead, we postulate a simple, high-level design process which is sufficiently abstract as to embrace a wide array of design decision-making.

The design process begins with developers who are starting to design software in response to a set of requirements. If the developers are able to meet the requirements for system X with a single word or expression (e.g., "Do X") then our research does not apply to their situation. Obviously, the more realistic case is that the solution is much more extensive. As soon as a solution will require more than a single word or expression, it is possible to speculate about the size, structure, and other attributes of candidate solutions. Developers necessarily must decide how to fashion some arrangement of pieces that, taken together, meet system requirements. Each piece, or design unit, has a role to fulfill in the overall design. Our view of the design process model assumes solely that the software can be viewed as a set of such design units and relations on the set.

We use the term "design unit" to retain the generality we seek. In practice, developers may consider design units to be subsystems, computer software components (CSCs), objects, processes, tasks, modules, packages, or other entities. Similarly, we refer to a design relation to include any relation on a set of design units. Simple relations, using modules as design units, include the following:

- Control coupling--one module can potentially call another module.
- Data coupling--data from one module is made available to another module.

Decisions in our simple design process result in design artifacts which may be characterized by the following attributes:

- Number of design units.
- Kinds of design units (e.g., tasks, modules) and their frequency of use.
- Number and kinds of design relations and their frequency of use.

We hypothesize that complex designs are more likely to correspond to software that is defect-prone or difficult to maintain. We see evidence of complexity in high interconnection (e.g., numerous relations) among the design units.

Other design decisions lead to resources (e.g., subprograms and objects) declared in particular design units. As units need resources from other units, patterns of resource sharing emerge. We view extensive resource accessing among design units as contributing to complexity.

Both notions of complexity--high interconnection and extensive resource accessing are explored in section 4.

To illustrate the linkages we see among design decisions, the design artifact, and resulting quality, consider the following example: A project has reached the stage in which developers are providing a scheme for storing and retrieving data from a database supplied and maintained by a separate organization. Suppose the developers decide all access to the database will be provided by design unit X. Other units requiring data from the database will not access the database directly, but instead will call on unit X. This decision has consequences in the architectural design of the system. Examination of the design artifact will reveal access relations only from unit X to the database. We would expect numerous access relations from other units to unit X. Had the decision instead been made for units to directly access the database, we would detect differences in the artifact, namely, the absence of unit X and the presence of numerous access relations from other units directly to the database.

A single design decision can affect many quality factors, like reliability, maintainability, and flexibility. Continuing the example, a decision not to create unit X to handle database access may lead to more difficult maintenance and more defects. Suppose a task leader misunderstands the command format for accessing the database and communicates the incorrect information to the development team. Several developers may then implement modules with incorrect database access commands. Such a defect may persist through code reading and unit testing. Code readers, relying on incorrect information, may not see the code as defective. Unit testers may not actually access the database, but, instead write test code to emulate the database input and output.

If access to the database is finally provided in integration testing, the defect may be detected. Correcting the defective code would require handling all modules that used the faulty database commands. Also, whenever a module is handled, the potential exists for a new defect to be introduced.

This example traces the relationships we see between a design decision, its observable consequences on the design artifact, and the possible effects on maintainability and reliability.

3.2 REPRESENTATION OF ADA DESIGNS

We developed a representation of Ada designs to serve as the basis for identifying design characteristics. Our interests centered on representing system-level architectures, rather than the control flow and data flow within individual procedures. The approach, following section 3.1, was to view Ada systems as being composed of design units ("parts"), and design relations ("connections"). The representation of software architectures as "parts" and "connections" is similar to representations used by other investigators (see, for example, [23] and [24]). We developed a particular architectural representation for Ada: one that provides a level of granularity appropriate to our consideration of hypotheses about the way design artifacts reflect design decisions. The benefit this provides to our research is having key architectural relations expressible in a machine-processable form.

Our sole assumption was that the Ada design representation be compilable. Assuming compilability, we examined the system structuring rules in the Ada language reference manual [25]. Our objective was to identify Ada language constituents that were candidates to serve as design units and design relations. The first three subsections discuss the results of our examination of Ada program units, Ada compilation units, and the issues related to the dynamic creation of task objects. Our decisions concerning design units and relations are reported in the remaining two subsections.

3.2.1 Ada Program Units

Ada programs are composed of the following program units: generic subprogram, generic package, subprogram, package, and task. Each program unit consists of a separate specification and body, with two exceptions: a non-generic subprogram specification is not always needed, and certain packages and generic packages do not need a package body.

Generic units are templates which are instantiated to produce non-generic subprograms or packages. Generic subprogram declarations, generic package declarations, generic subprogram instantiations, generic package instantiations, subprogram declarations or bodies, or package declarations may be compiled as library units. These may then be imported (via the Ada "with" clause) to provide the context for other compilations.

Program unit bodies nested immediately within library unit bodies may be declared as body stubs with the corresponding proper body compiled as a separate subunit. Subunits, in turn, may contain program unit bodies declared as body stubs. In this way, nesting can extend to multiple levels.

Task units may be declared as single task objects or as task types which are templates for any number of task objects created at run-time. Each task object represents a separate thread of control which is scheduled for execution by the Ada run-time environment.

3.2.2 Ada Compilation Units

In addition to a program unit, an Ada compilation unit emerged as likely candidate to serve as a system building block. According to the Ada standard [25], "A program is a collection of one or more compilation units submitted to a compiler in one or more compilations." Fifteen different Ada compilation units were identified (table 3-1). A compilation unit is described in table 3-1 as consisting of a program unit and a designation as either a specification, body, instantiation, or subunit. Table 3-1 further identifies the compilation unit as a library unit or secondary unit or (in the case of a subprogram body) both.

3.2.3 Dynamic Creation of Task Objects

A task is one of the five Ada program units; a task subunit is one of the fifteen compilation units from table 3-1.

Each task object declared during program execution provides a separate thread of control scheduled by the Ada run-time environment for its activation, synchronization and communication with other task objects, and termination. Limited static analyses could be performed for single task objects declared immediately within the specification or body of library packages or library package instantiations since these declarative regions are elaborated exactly once before execution of the main program begins. It is likely that task objects in the declarative part of the main program also would be elaborated exactly once since it is unlikely that the main program would be recursively called.

Task type declarations visible to the main program or to other task bodies may be used as the designated type in an access type declaration and as component types in a composite type. This allows any number of tasks to be created by the elaboration of object declarations and the execution of allocators at run-time. This requires dynamic analysis of the program units to determine the complexity of the system operation. Any model for the intended interaction of these tasks probably would need to be provided by the system designer together with a method of ensuring that only the required number of task objects were created during system execution. We find it difficult to envision a general analysis method being devised to predict the number of task objects that could be created by a particular set of Ada compilation units. Any alternative method for synchronizing and communicating multiple threads of control probably would be as complex as the corresponding system of Ada tasks and would have the disadvantage of being less well understood. This analysis caused us to defer pursuing techniques for the identification of potential concurrency relations among tasks.

Table 3-1. Ada Compilation Units

Compilation Unit Number	Compilation Unit Name: Program Unit	Compilation Unit Name: Suffix	Library Unit	Secondary Unit
1.	Generic Package	Specification	Yes	
2.	Generic Package	Body		Yes
3.	Generic Package	Subunit		Yes
4.	Generic Package	Instantiation	Yes	
5.	Package	Specification	Yes	
6.	Package	Body		Yes
7.	Package	Subunit		Yes
8.	Generic Subprogram	Specification	Yes	
9.	Generic Subprogram	Body		Yes
10.	Generic Subprogram	Subunit		Yes
11.	Generic Subprogram	Instantiation	Yes	
12.	Subprogram	Specification	Yes	
13.	Subprogram	Body	Yes	Yes
14.	Subprogram	Subunit		Yes
15.	Task	Subunit		Yes

3.2.4 Design Units

Three candidates for design unit emerged from our investigation: program unit, compilation unit, and a library unit aggregation. We define a library unit aggregation to be a library unit, its corresponding body (if any), and all subunits for which it is the ancestor library unit (as used in [25]). Equivalently, library unit aggregation is the library unit's declarative scope.

Compilation unit was selected as the principal design unit for purposes of our research. Our experience with Ada systems indicates that compilation units are frequently used as the entities for both project data collection and static analysis by off-the-shelf source code analysis tools.

3.2.5 Design Relations

Focusing on compilation units as design units, we investigated static relationships among such units. The following observations refer by number to the compilation units in table 3-1 in discussing the legal (in the sense of [25]) units for each relation.

Ada allows any compilation unit to import a library unit (units 1, 4, 5, 8, 11, 12, 13 in table 3-1) through a context clause. A generic instantiation (unit 4 or 11) would reference only the single generic specification (unit 1 or 8 respectively) mentioned in the instantiation since any other generic specification would not provide any entities to be used as actual parameters of the instantiation.

Since Ada allows a body stub to be declared only immediately within the declarative part of a library subprogram body or secondary unit, only these units (units 2, 3, 6, 7, 9, 10, 13, 14, 15) may be the parent unit for a subunit (units 3, 7, 10, 14, 15).

A generic unit specification declared in the visible part of a library package specification or library package instantiation (units 4, 5) may be referenced as the name of the generic unit for a generic instantiation (units 4, 11). If the context clause contained a use clause for the library unit (units 4, 5), any tools used to determine the location of the generic unit would need to inspect the entities declared in the visible part of the library unit or of the generic package specification for the library package instantiation.

The rules defining the order in which units can be compiled and recompiled form dependencies among compilation units of a design representation. A legal order for the original compilation of the compilation units is largely determined by context clauses and body stubs. However an Ada compiler is allowed to introduce additional dependencies based on the actual compilation order used and whether compilation units are submitted as separate compilations or included together as a single compilation. These additional dependencies would affect the legality of subsequent compilations or linking operations and the effectiveness of pragma INLINE for the original or subsequent compilations. Further, an implementation is allowed to retain compilation units if it can deduce that some of the potentially affected units are not actually affected by changes in recompilation. An analysis tool for determining the recompilation complexity of an Ada design could identify the potentially affected units for a proposed recompilation or possibly be tailored to the recompilation rules for a particular compiler.

These observations resulted in the identification of the following *five static design relations* for purposes of this research; where A and B are compilation units defined in table 3-1:

1. Context coupling relation

    A "withs" B,
    where A: any of the 15 compilation units
    B: a library unit (units 1, 4, 5, 8, 11, 12, 13 from table 3-1)

2. Specification/body relation

    A is the specification for body B,
    where A: a library unit specification (units 1, 5, 8 12)
    B: a proper body (units 2, 6, 9, 13)

3. Parent/subunit relation

    A is the parent unit for subunit B,
    where A: a library subprogram body or a secondary unit (units 2, 3, 6, 7, 9, 10, 13, 14, 15)
    B: a subunit (units 3, 7, 10, 14, 15)

4. Generic template/instantiation relation

    A contains a generic body for a generic unit instantiated by B,
    where A: a generic unit body, package body, or subunit (units 2, 3, 6, 7, 9, 10, 14, 15)
    B: any of the 15 compilation units

Note: If unit A is compiled before B and if the body is actually included INLINE at the point of the generic instantiation then a dependency may exist. Subsequent recompilation of A would require recompilation of B. This may add a dependency which originally did not exist. If unit B was compiled before unit A, then no dependency would exist since the generic unit body was not available for inclusion in the compilation of unit A. Some compiler implementations may require that some or all of the bodies for generic units declared in A be included in the same compilation containing the specification for unit A. This forces a dependency for all units that instantiate any generic unit contained in A on all secondary units that contain generic unit bodies since recompilation of any of these forces recompilation of library unit A.

5. Pragma INLINE dependency relation

A contains a subprogram body or instantiation mentioned in pragma INLINE and called by unit B,
where A: a library subprogram body or a subprogram instantiation, body, or subunit (units 11, 13, 14)
B: any of the 15 compilation units

Note: If B is compiled before A, then no dependency exists since the pragma INLINE would have to be ignored. If A is compiled before B and if the body is actually included INLINE at the point of the subprogram call then a dependency exists. Subsequent recompilation of A would require recompilation of B. This may add a dependency which did not exist at the end of the original compilation.

Of the five relations, context coupling has been a particular focus in our research. When a context coupling (e.g., A "withs" B) exists, we know that compilation unit A requires resources in B. We then want to know how many resources in B have been made visible to unit A (and to other units that "with" B). Next we want to profile the resources provided by B as being either program units, types, or objects. Of the resources visible in B, how many are actually accessed or used by unit A? We hypothesize that the number and profile of resources both available and accessed may be significant design characteristics in estimating reliability and maintainability.

Context coupling also provides a medium for control flow and data flow relations among compilation units. These two relations have been the most extensively researched for their effects on software complexity (see, for example, [5] and [6]). Control flow potentially occurs when A "withs" B, a subprogram is one of the visible resources of B, and A includes code to call the subprogram. Data flow occurs within the context coupling relation via subprogram parameters or visible objects. The preliminary analysis results in section 4 reflect the research attention on context coupling to characterize the interconnection and resource sharing in Ada designs.

SECTION 4

PRELIMINARY ANALYSIS RESULTS

This section presents preliminary results of analyzing project data to determine significant factors affecting reliability and maintainability. The results are preliminary because the research project is continuing to obtain project data and develop new hypotheses about the relationships between design characteristics and quality factors. The results in this section are based on 21 observations (subsystems) from four projects. The project data is described in section 4.1.

The statistical analysis and model building methodologies are discussed in section 4.2. Section 4.3 presents the specific reliability and maintainability measures used in the analysis. Section 4.4 discusses the independent (explanatory) variables in this analysis. These variables reflect the design hypotheses introduced in section 3. In section 4.5, the statistical results for both reliability and maintainability are discussed.

The project data elements used in statistical analyses are defined in table 4-1. Names of data elements are consistently written in lower-case. The data elements are used to compute statistics for regression analysis. The computed statistics are written in upper-case.

4.1 PROJECT CHARACTERISTICS

Data used in preliminary analysis of factors affecting reliability and maintainability were obtained through the cooperation of the Software Engineering Laboratory (SEL) of NASA/GSFC. There are two sources of data. First, Ada code for each project is analyzed using the Ada Static Source Code Analyzer Program (ASAP) [26]. The data generated by ASAP is used to compute design characteristics in table 4-1 for the subsystems of each project. We are analyzing Ada code from systems which have completed acceptance testing. We are extracting structural information whose availability is reasonable to expect, based on our comparison of earlier design documents to design features of the completed systems. The general issue of availability of design-related information at various points in the development process is currently being investigated.

A second source of data is the SEL database which characterizes the development process for each Ada project. This database includes information on the origin of each compilation unit (e.g., extent of reuse, subsystem in which it is contained), defect and non-defect modifications, processor utilization, and staff effort expended on various development activities [27].

Data has been obtained on four Ada projects, consisting of 21 subsystems. The projects involve the development of interactive, ground-based, scientific applications.

Table 4-1. Project Data Element Definitions

Software Composition

ss	number of subsystems
lu	number of library units
cu	number of compilation units
ksctot	thousands of source lines of code* - total
kscv	thousands of source lines of code* - verbatim (i.e., reused without modification)
kscs	thousands of source lines of code* - slightly modified (i.e., $\leq$ 25 percent of source lines modified)
kscx	thousands of source lines of code* - extensively modified (i.e., > 25 percent of source lines modified)
kscn	thousands of source lines of code* - newly developed

Design Characteristics

cc	number of context couples
dexp	number of declarations exported
dimp	number of declarations imported
dimpc	number of declarations imported - cascaded
dimpint	number of declarations imported - internal (from within the subsystem)
dimpext	number of declarations imported - external (from other subsystems)

Software Modifications and Errors

modtot	number of non-defect modifications - total
deftot	number of defects - total
defsa	number of defects - reported during system testing and acceptance testing

*Non-comment, non-blank source lines of code.

Table 4-1. (Concluded)

isovs	number of times isolating the cause of the defect was very simple (requiring ≤ 1 hour)
isos	number of times isolating the cause of the defect was simple (requiring > 1 hour and ≤ 1 business day)
isod	number of times isolating the cause of the defect was difficult (requiring > 1 day and ≤ 3 days)
isovd	number of times isolating the cause of the defect was very difficult (requiring > 3 days)
fixvs	number of times fixing the defect was very simple (requiring ≤ 1 hour)
fixs	number of times fixing the defect was simple (requiring > 1 hour and ≤ 1 business day)
fixd	number of times fixing the defect was difficult (requiring > 1 day and ≤ 3 days)
fixvd	number of times fixing the defect was very difficult (requiring > 3 days)

Selected project characteristics are shown in table 4-2. Across all projects, approximately 183 thousand source lines of code (KSLOC) in 1,984 compilation units have been analyzed. The projects range between 33 and 73 KSLOC. Reuse ratios (fraction reused verbatim or with slight modifications) lie between nine and thirty percent. The reliability varies between 3.0 and 9.3 total defects per KSLOC. Total defects include those reported during unit testing, system testing, and acceptance testing. Maintainability is indicated by two different measures. The fraction of defects taking less than or equal to one hour to isolate varies between 28 percent and 73 percent, while the fraction of defects taking less than or equal to one hour to correct varies between 39 percent and 72 percent.

In table 4-3, the equivalent data is given for each of the subsystems of the projects. The subsystems for project i are denoted i-1, i-2, etc. Note, in particular, the increased ranges over which the reusability, reliability and maintainability measures vary when data is reported at the level of subsystem instead of project.

4.2 STATISTICAL ANALYSIS AND MODEL BUILDING

Our research is built around the identification of hypotheses regarding the determinants of software reliability and maintainability, and the empirical testing of these hypotheses.

Table 4-2. Characteristics of Projects

Project	Software Size[a]	Library Units	Compilation Units	Reuse[b]	Reliability[c]	Maintainability Isolation[d]	Maintainability Correction[e]
1	73.1	167	658	.28	5.5	.61	.70
2	39.0	117	476	.27	3.0	.28	.39
3	33.3	58	418	.02	9.3	.73	.72
4	37.0	166	432	.26	4.4	.57	.55
All Projects	182.4	508	1,984	.22	5.4	.60	.65

a Size defined as thousands of non-comment, non-blank source lines of code (ksctot in Table 4-1).
b Reuse defined as (kscv+kscs)ksctot: the fraction of reused verbatim and slightly modified code.
c Reliability defined as (deftot/ksctot): defects per thousand source lines of code.
d Isolation defined as (isovs/deftot): the fraction of defects requiring less than or equal to one hour to isolate.
e Correction defined as (fixvs/deftot): the fraction of defects requiring less than or equal to one hour to correct.

Table 4-3. Characteristics of Subsystems

Subsystem	Software Size[a]	Library Units	Compilation Units	Reuse[b]	Reliability[c]	Maintainability Isolation[d]	Maintainability Correction[e]
1-1	7.3	13	63	.00	6.4	.45	.62
1-2	2.7	7	20	.50	8.0	.77	.77
1-3	26.1	84	286	.07	4.5	.69	.78
1-4	4.0	7	31	.02	8.6	.26	.79
1-5	27.3	38	185	.44	6.2	.64	.65
1-6	5.7	18	73	.94	1.6	.66	.89
2-1	2.9	10	30	.00	4.8	.57	.79
2-2	15.4	68	227	.35	2.7	.32	.41
2-3	13.8	16	159	.02	3.8	.21	.26
2-4	6.9	23	60	.74	1.4	.20	.40
3-1	2.6	3	25	.00	12.9	.81	.85
3-2	11.6	19	128	.02	8.6	.78	.69
3-3	7.5	14	107	.00	2.9	.64	.59
3-4	5.0	6	63	.09	16.6	.70	.75
3-5	6.6	16	95	.00	10.7	.68	.72
4-1	6.6	30	101	.38	3.8	.60	.64
4-2	12.9	67	105	.00	5.2	.58	.58
4-3	3.5	12	66	.09	8.0	.46	.46
4-4	2.2	9	35	.05	4.9	.36	.55
4-5	2.2	5	13	.61	3.7	1.00	.75
4-6	9.6	43	112	.54	2.6	.56	.40

a Size defined as thousands of non-comment, non-blank source lines of code (ksctot in Table 4-1).
b Reuse defined as (kscv+kscs)/ksctot: the fraction of reused verbatim and slightly modified code.
c Reliability defined as defects per thousand source lines of code.
d Isolation defined as (isovs/deftot): the fraction of defects requiring less than or equal to one hour to isolate.
e Correction defined as (fixvs/deftot): the fraction of defects requiring less than or equal to one hour to correct.

The overarching perspective is that design complexity is a major determinant of software system defects generated during post-design implementation. Design complexity is multidimensional, and may be characterized in terms of measurable variables. If these design measures exhibit independent variation within an ensemble of empirical observations, their impacts on software defects can be estimated.

We also recognize the environment within which software is developed may influence defect occurrences. Examples of environmental factors are the experience of the software development team, and the extent to which the software development organization reuses software.

Design characteristics and environmental factors are introduced into multivariate models to explain reliability and maintainability as depicted in the quality equation from section 1.3.

The following two subsections discuss the dependent variables and independent variables of the models.

4.3 SELECTION OF DEPENDENT VARIABLES

This section defines the dependent variables, corresponding to reliability and maintainability, used in the analysis. The selection of measures for reliability and maintainability was guided by the set of candidate definitions in section 2 and the availability of project data as described in section 4.1.

4.3.1 Reliability

Project data, discussed in section 4.1, supports a reliability measure based on defects. We used measure R5 (from section 2), defect density. The project data provides pre-release defect data collected during unit, system, and acceptance testing.

We regard this complete project defect record as total defects (deftot from table 4-1), leading to the reliability measure TOTDEFSL defined as follows:

$$\text{TOTDEFSL} = \text{deftot}/\text{ksctot} \tag{2}$$

Our research focus on architectural design, rather than detailed, intra-procedural design, led us to investigate a second reliability measure. We subtracted the unit testing defects from the total, leaving the defects reported during system and acceptance testing (defsa). We wanted to explore the matching of architectural design decisions with defects likely to reflect those decisions. Defects during unit testing are more likely to reflect implementation or detailed design decisions. When unit testing is completed, collections of units are tested during system and acceptance testing in the SEL environment. These testing activities are more likely to expose defects reflecting higher-level design decisions concerning inter-unit operation and relationships. Note, however, that defects were associated with either unit, system, or acceptance testing by matching the date of the defect report with phase dates for each testing activity. Defects reported during system and acceptance cannot be expected exclusively to correspond to inter-unit relationships. Original defect reports were not examined to attempt to make a finer determination by, for example, analyzing the description of the defect. We maintain only that defsa is more likely to exclude implementation-related defects corrected during unit testing. The resulting reliability measure, SYACDEFSL, is defined as follows:

$$\text{SYACDEFSL} = \text{defsa}/\text{ksctot} \quad (3)$$

The analysis results in this section indicate when either TOTDEFSL or SYACDEFSL is used as the dependent variable for reliability.

4.3.2 Maintainability

Ideally, we want to know the time (in hours or days) required to isolate the defects and make the necessary changes to repair the defects (measure M1 from Section 2). Our project data instead provides this time data in four ordinal categories:

- Less than or equal to one hour
- Greater than one hour but less than or equal to one business day
- Greater than one day but less than or equal to three days
- Greater than three days

This maintenance data is more informative than a simple ordinal ranking (e.g., from 1 to 4) because the ranks are associated with ranges of staff effort. In this sense, the data clearly supports measure M4, easy fix frequency, and provides information concerning M1, effort to correct.

An additional feature of the project data is the presence of two rankings: one for effort to isolate the defect and the other for effort to correct the defect. Table 4-1 defines the eight data elements corresponding to the four ordinal categories for isolation time and correction time, respectively.

Our approach was to use M4, defining an easy fix as the first category requiring less than or equal to one hour. The two maintainability measures are defined by the EFFISO and EFFFIX, as follows:

$$\text{EFFISO} = \text{isovs}/\text{deftot} \quad (4)$$
$$\text{EFFFIX} = \text{fixvs}/\text{deftot} \quad (5)$$

4.4 SELECTION OF INDEPENDENT VARIABLES

Our research identified independent measures for design characteristics and environmental factors. The number of independent (explanatory) variables which can be introduced is restricted by the number of observations collected thus far. This section discusses the independent variables used in the preliminary statistical analyses. Four design characteristics (context coupling, visibility, import origin, and internal complexity) and two environmental factors (volatility and reuse) are discussed. Table 4-4 summarizes the independent variables defined in this section. Table 4-5 provides descriptive statistics for the independent variables using project data (21 subsystems).

4.4.1 Context Coupling

Context coupling refers to the use of "with" clauses allowing the exporting of declarations from a library unit to another compilation unit. The compilation unit is said to import the declarations of the library unit.

Context coupling measures the interconnection of compilation units as an indication of design complexity. Higher values of context coupling measures are hypothesized to be associated with higher defect densities and lower maintainability.

Several context coupling measures have been identified. The most obvious measure is the number of context couples per library unit aggregation (defined in section 3) denoted by CCPLU and defined as follows:

$$\mathrm{CCPLU} = cc/lu \qquad (6)$$

If the identical library unit is "withed" into a package specification and its corresponding body, then the contribution to CCPLU is one rather than two. Importing declarations to the specification makes the declarations available to the corresponding body, so for our purposes the context clause for the body is redundant.

Another context coupling measure is the import/export ratio, IMPEXP, defined by:

$$\mathrm{IMPEXP} = dimp/dexp \qquad (7)$$

For a closed system (i.e., no exports or imports across the system boundary), all of whose library units export the same number of declarations, the value of IMPEXP equals the value of CCPLU. When the library units do not export the same number of declarations, the measure is interpreted as a weighted average of the context couples. The weights are chosen in proportion to the number of declarations exported across a context couple. So that the weights sum to unity, the proportionality constant is set equal to the total number of exports. Thus, context clauses with relatively high numbers of exported declarations are more heavily weighted.

Table 4-4. Independent Variables Used in Statistical Analysis

Design Characteristic	Independent Variable Name		Independent Variable Definition[1]
Context Coupling	CCPLU:	Context Couples Per Library Unit	CCPLU = cc/lu
	IMPEXP:	Import/Export Ratio	IMPEXP = dimp/dexp
	CIMPEXP:	Cascaded Import/Export Ratio	CIMPEXP = dimpc/dexp
Visibility	CIMPIMP:	Cascaded Imports	CIMPIMP = dimpc/dimp
Import Origin	FINTIMP:	Fraction of Internal Imports	FINTIMP = dimpint/dimp
Internal Complexity	EXPPLU:	Exports Per Library Units	EXPPLU = dexp/lu

Environmental Factor	Independent Variable Name		Independent Variable Definition
Volatility	MODPLU:	Modifications Per Library Unit	MODPLU = modtot/lu
Reuse	FNEMSL:	Fraction of New or Extensively Modified Code[2]	FNEMSL = (kscn+kscx)/ksctot

1 Data elements cc, lu, and so on are defined in Table 4-1.

2 FNEMSL measures the fraction of code not reused (see Section 4.4.6).

Table 4-5. Descriptive Statistics for Model Variables*

Variable	Mean	Standard Deviation	Minimum	Maximum
TOTDEFSL	6.1	3.8	1.4	16.6
SYACDEFSL	3.5	2.2	.18	7.8
EFFISO	.57	.21	.20	1.0
EFFFIX	.64	.17	.26	.89
CCPLU	13.1	.5	.67	35.3
IMPEXP	24.6	22.6	.87	102.2
CIMPEXP	60.4	42.7	2.3	167.6
CIMPIMP	2.9	1.6	1.2	7.4
FINTIMP	.31	.31	.01	1.0
EXPPLU	39.1	20.3	15.0	91.2
MODPLU	3.7	4.6	.13	19.3
FNEMSL	.77	.29	.06	1.0

* Sample size = 21

An alternative interpretation is that the import/export ratio represents the average number of times an export in a closed system is imported, which is an indicator of coupling.

Still a third perspective, applicable to both open and closed systems, interprets the import/export ratio as the average number of declarations imported to "support" an exported declaration. Imports can be viewed as providing services to the importing unit. The unit depends on these imports, for example, to provide a data type or a procedure. Of course, some imported declarations will not be used, but they nevertheless contribute to complexity from the perspective of a programmer responsible for implementing the design or isolating and correcting defects. For example, a programmer must consider the possibility that a declaration which has not been referenced may be the source of a defect.

A final measure of context coupling can be derived by observing that the declarations exported by a library unit to a specification, for example, will cascade through any corresponding body and subunits, effectively importing to these compilation units as well. Thus, the import count is magnified by this cascading effect. A context coupling measure accounting for this effect is defined as follows:

CIMPEXP = dimpc/dexp (8)

The value of this measure may be controlled somewhat by placing context clauses at the lowest possible level in the library unit aggregation.

4.4.2 Visibility

Visibility has been investigated as a measure of interest in Ada development [28]. A library unit aggregation may have extensive structure in terms of a corresponding body and perhaps multiple levels of subunits. Context clauses for such a library unit aggregation may be all at the specification level, or may appear at the body or subunit levels providing only the needed visibility. Information hiding is served by the appropriate placement of context clauses within a library unit aggregation. If all context clauses are at the highest level (the specification), then the programmer may be working with an excessive number of imported declarations which cascade through the entire aggregation.

A measure which accounts for this effect is given by CIMPIMP, defined as follows:

CIMPIMP = dimpc/dimp (9)

This measure effectively allows us to split the measure in (8) into the direct effect given in (7) and the cascade effect in (9).

4.4.3 Import Origin

Exports and imports may occur among compilation units within a subsystem, or compilation units may import declarations from other subsystems. If software development teams are organized by subsystem, then a team may be less familiar with the imports coming from other subsystems. We might expect defects to decrease as the ratio of internal imports to total imports increases. This measure, FINTIMP, is defined as follows:

$$\text{FINTIMP} = \text{dimpint}/\text{dimp} \tag{10}$$

4.4.4 Internal Complexity

Context coupling measures are basically measures of the "external" or architectural complexity of the design. The visibility measure exhibits characteristics of both an external and an internal complexity measure. It reflects both importing of declarations among units (an architectural feature) and positioning of context clauses within a library unit aggregation (an internal complexity feature). However, we believe a more complete explanation of reliability and maintainability requires greater consideration of internal complexity within a library unit aggregation. Marginally, we might expect some negative correlation between internal and external complexity. Thus, ignoring internal complexity might result in an upward bias in the estimate of the coefficients associated with external complexity [29, p. 291-298].

Internal complexity may be characterized by the partitioning of the calling tree within the program library. ASAP does not have the capability of providing this information. Therefore, the incorporation of a measure based on calling tree fragmentation into models for reliability and maintainability has not been possible.

Accordingly, a less satisfactory measure of internal complexity was devised. The number of declarations a library unit exports was regarded as a crude proxy of its internal complexity. These declarations are used and implemented in the secondary units associated with the library unit. Thus, the average internal complexity of a subsystem is the number of exported declarations per library unit denoted by EXPPLU and defined as follows:

$$\text{EXPPLU} = \text{dexp}/\text{lu} \tag{11}$$

4.4.5 Volatility

The project data also reflects non-defect modifications. These changes are interpreted as an indicator of software volatility which may be expected to increase the defect rate. The volatility measure MODPLU is defined as follows:

$$\text{MODPLU} = \text{modtot}/\text{lu} \tag{12}$$

4.4.6 Reuse

Reuse has been shown to improve reliability [30]. The project data provides information on software reuse for each compilation unit. The origin of a compilation unit is identified according to four categories:

- Reused verbatim (without change)
- Reused with slight modification ($\leq$ 25 percent of the source lines)
- Reused with extensive modification (> 25 percent of the source lines)
- Newly developed code

FNEMSL is the fraction of source code that was new or extensively modified. The reuse measure was defined in this way to eliminate zero values from entering into the logarithmic transforms. FNEMSL is defined as follows:

$$\text{FNEMSL} = (\text{kscn} + \text{kscx}) / \text{ksctot} \qquad (13)$$

4.5 PRELIMINARY STATISTICAL RESULTS

Preliminary results of statistical analyses are presented in this section. The results are based on 21 observations (subsystems) from the project data described in section 4.1. Analysis results are presented first for reliability and then for maintainability.

4.5.1 Reliability

Results are presented for the two measures of reliability discussed in section 4.3: total defects per thousand source lines of code (TOTDEFSL), and system and acceptance test defects per thousand source lines of code (SYACDEFSL). Multivariate regression techniques are used to regress the two forms of defect density against various combinations of the explanatory variables discussed in section 4.4. After initial experiments with linear models, we decided to focus on log-linear models because of the inherent curvature in the relationships between the dependent and independent variables. These models take the form of:

$$\log(Y) = a_0 + a_1 \cdot \log(X_1) + a_2 \cdot \log(X_2) + \ldots \qquad (14)$$

where:

Y	=	dependent variable (e.g. TOTDEFSL or SYACDEFSL)
X_i	=	ith independent (explanatory) variable, i=1,2, ...

Initial univariate regression analyses were conducted for the three different context coupling measures in table 4-4. The results are shown in the first three equations of table 4-6 (for TOTDEFSL), and table 4-7 (for SYACDEFSL). The variables shown in the tables are logarithmic transforms of the dependent and independent variables. Thus equation (1) of table 4-6 can be written explicitly as:

$$\log(\text{TOTDEFSL}) = .58 + .47 \cdot \log(\text{CCPLU}) \qquad (15)$$

The numbers in parentheses in tables 4-6 and 4-7 represent the standard errors of the parameter estimates. R^2, the coefficient of determination, is the fraction of variation of the dependent variable explained by the independent variables.

In all cases, the coefficient estimates have the expected signs: defect densities increase as design complexity, measured by context coupling, increases. Of the three context coupling measures, CCPLU performs poorly in tables 4-6 and 4-7. In both tables, the cascaded import/export ratio, CIMPEXP, performs about the same as direct import/export ratio, IMPEXP.

Results from equations 2 and 3 in both tables suggest that the context coupling effect and the cascade effect might each enter the regression analysis as independent variables. The process by which context coupling is established may be viewed in two steps. First, declarations are imported into a library unit aggregation without reference to the specific compilation units into which the declarations are imported; the complexity associated with this step is characterized by the direct import-export ratio IMPEXP. Second, decisions are made to attach the context clauses to particular compilation units. CIMPIMP measures the effect of this second step. This two-step process uses IMPEXP and CIMPIMP rather than the single variable CIMPEXP.

Results of incorporating IMPEXP and CIMPIMP in a regression are shown in equation 4 of tables 4-6 and 4-7. In both cases, the coefficient of CIMPIMP enters with the appropriate sign, and the equations have a bit more explanatory power (as indicated by R^2) than equation 3.

Tables 4-8 and 4-9 present the results of three-variable regression analyses for the two different measures of defect density. These analyses are more exploratory in nature, because they are based on only 21 observations. Spurious correlations among the explanatory variables (multicollinearity) and the lower number of degrees of freedom associated with the coefficient estimates may contribute to larger standard errors for these estimates.

Table 4-6. Log-Linear Regression Results for TOTDEFSL

Variable	Equation 1	Equation 2	Equation 3	Equation 4
Intercept	.58[a] (.24)[b]	.27 (.28)	-.14 (.35)	-.04 (.35)
CCPLU	.47 (.10)			
IMPEXP		.48 (.09)		.51 (.09)
CIMPEXP			.47 (.09)	
CIMPIMP				.26 (.18)
R^2 [c]	.52	.58	.59	.62

a Parameter estimate.

b Standard error of the parameter estimate.

c Coefficient of determination.

Table 4-7. Log-Linear Regression Results for SYACDEFSL

Variable	Equation 1	Equation 2	Equation 3	Equation 4
Intercept	-.24[a]	-.87	-.15	-1.42
	(.24)[b]	(.39)	(.47)	(.48)
CCPLU	.54			
	(.16)			
IMPEXP		.65		.70
		(.13)		(.13)
CIMPEXP			.66	
			(.12)	
CIMPIMP				.46
				(.25)
R^2 [c]	.37	.56	.61	.63

a Parameter estimate.

b Standard error of the parameter estimate.

c Coefficient of determination.

Table 4-8. Three-Variable Log-Linear Regression Results for TOTDEFSL

Variable	Equation 1	Equation 2	Equation 3	Equation 4
Intercept	.02[a]	-1.5	.01	.65
	(.36)[b]	(1.04)	(.66)	(.36)
IMPEXP	.41	.61	.49	.27
	(.15)	(.11)	(.16)	(.11)
CIMPIMP	.28	.27	.25	.05
	(.18)	(.18)	(.21)	(.16)
FINTIMP	-.11			
	(.13)			
EXPPLU		.34		
		(.22)		
FNEMSL			.03	
			(.24)	
MODPLU				.27
				(.08)
R^2 [c]	.64	.67	.62	.76

a Parameter estimate

b Standard error of the parameter estimate.

c Coefficient of determination.

Table 4-9. Three-Variable Log-Linear Regression Results for SYACDEFSL

Variable	Equation 1	Equation 2	Equation 3	Equation 4
Intercept	-1.4[a] (.51)[b]	-1.7 (1.5)	.77 (.65)	-1.5 (.62)
IMPEXP	.69 (.21)	.72 (.16)	.19 (.16)	.74 (.18)
CIMPIMP	.46 (.26)	.46 (.26)	.07 (.21)	.49 (.28)
FINTIMP	-.01 (.19)			
EXPPLU		.06 (.32)		
FNEMSL			.97 (.24)	
MODPLU				-.04 (.14)
R^2 [c]	.63	.63	.81	.63

a Parameter estimate.

b Standard error of the parameter estimate.

c Coefficient of determination.

For all equations in the tables, estimated coefficients are of the expected signs. Equation 4 in table 4-8, which involves the non-defect modifications per library unit, MODPLU, is the strongest predictor of defect density. On the other hand, equation 3 in table 4-9, which incorporates the fraction of new and extensively modified code, is the strongest predictor of system and acceptance test defects per thousand lines of code.

The three-variable regression estimates suffer from the fact that the four new variables (FINTIMP, EXPPLU, FNEMSL, and MODPLU) introduced in tables 4-8 and 4-9, are all highly correlated with the import-export measure IMPEXP, as follows (using the Pearson correlation coefficient): FINTIMP (-.79), EXPPLU (-.59), FNEMSL (.75), and MODPLU (.65).

While all of these correlations may be spurious, resulting from the relatively small number of observations, the correlations for the last three variables deserve special discussion. The negative correlation of the internal complexity variable, EXPPLU, with the external complexity variable, IMPEXP, might be expected on the grounds that tradeoffs may exist between these two types of complexity. Decisions to incorporate exported declarations into larger library units may lead to lower context coupling. Therefore, this correlation may not be spurious.

The correlation of the fraction of new and extensively modified code, FNEMSL, with the external complexity measure is intriguing. The correlation possibly suggests that the decision to extensively reuse software components may result in lower complexity. The exact mechanism through which this occurs is open to discussion, but reuse may possibly encourage a more careful analysis of interfaces and a looser coupling of the compilation units.

The last correlation of the non-defect modifications per library unit, MODPLU, with context coupling may result from modifications causing new information to be accessed from other library units, leading to greater context coupling. On the other hand, more complex coupling may lead to more modifications to improve clarity, maintainability, or documentation as a consequence, for example, of code inspections. A resolution of this question requires further analysis.

4.5.2 Maintainability

Section 4.3 discusses two measures for maintainability, effort to isolate defects and effort to correct defects. To estimate these measures, we used ordered response models as discussed by Gurland et al. [31]. Assume that there is an underlying response variable, denoted by Y^*, which is either effort to isolate defects or effort to correct defects, defined by the following regression relationship for observation i:

$$\log(Y_i^*) = -a_0 - a_1 \cdot \log(X_{1i}) - a_2 \cdot \log(X_{2i}) - \ldots + u_i \qquad (16)$$

where:

a_j	=	jth regression parameter to be estimated, $j=1,...,n$
X_{ji}	=	value of jth explanatory variable for the ith observation, $i=1,...,m$
u_i	=	disturbance term for the ith observation.

The minus signs in front of the coefficients are chosen for convenience. The disturbance term, u, represents the unexplained variation, and is characterized by some as yet unspecified probability distribution.

Y_i^* is a continuous variable measuring effort to isolate or correct defects and is not directly observable in our categorical data. Instead, what is observed in the categorical data is a discrete dependent variable, Y_i, defined by:

$$\begin{aligned} Y_i &= 1 && \text{if} \quad Y_i^* \leq 1 \text{ hour} \\ &= 2 && \text{if} \quad 1 \text{ hour} < Y_i^* \leq 1 \text{ day} \\ &= 3 && \text{if} \quad 1 \text{ day} < Y_i^* \leq 3 \text{ days} \\ &= 4 && \text{if} \quad Y_i^* > 3 \text{ days} \end{aligned}$$

On the basis of this categorical data, a polychotomous ordinal analysis including all four categories can be conducted [31]. However, such a model would require the estimation of three constants, as well as any additional parameters associated with the explanatory variables. Since only 21 observations are currently available, we decided to conduct a dichotomous ordinal analysis, involving only two categories as defined below:

$$\begin{aligned} Y_i &= 1 && \text{if} \quad O < Y_i^* \leq 1 \text{ hour} \\ &= 2 && \text{if} \quad Y_i^* > 1 \text{ hour} \end{aligned}$$

The "$\leq$ 1 hour" cutoff was chosen because over 90 percent of the isolate and correct data was in the "less than one day" category. This definition corresponds to EFFISO and EFFFIX from section 4.3. Expressing the inequalities above in terms of logarithms yields the comparable relationships:

$$\begin{aligned} Y_i &= 1 && \text{if} \quad \log(Y_i^*) \leq C_O \\ &= 2 && \text{if} \quad \log(Y_i^*) > C_O \end{aligned}$$

where the logarithm of the time constraint is expressed in terms of the more general parameter C.

Substituting equation (16) into the above inequalities, and rearranging terms gives:

$$\begin{aligned} Y_i &= 1 && \text{if} \quad u_i \leq a_O + a_1 \cdot \log(X_{1i}) + a_2 \cdot \log(X_{2i}) + ... + C_O \\ &= 2 && \text{if} \quad u_i > a_O + a_1 \cdot \log(X_{1i}) + a_2 \cdot \log(X_{2i}) + ... + C_O \end{aligned}$$

With the u normally distributed, the probabilities of the occurrences of the dependent variable Y are given by:

$$\text{Prob}(Y_i = 1) = F\,(a_0 + a_1 \cdot \log(X_{1i}) + a_2 \cdot \log(X_{2i}) + \ldots + C_0)$$

$$\text{Prob}(Y_i = 2) = 1 - F\,(a_0 + a_1 \cdot \log(X_{1i}) + a_2 \cdot \log(X_{2i}) + \ldots + C_0)$$

where F(.) is the cumulative normal distribution function centered at the origin and with unit standard deviation.

The parameters of these equations were estimated using the logistics procedure of the Statistical Analysis System (SAS) which provides for the estimation of the parameters of models of the kind cited above. These preliminary estimates for EFFISO and EFFFIX are shown in table 4-10. The design characteristic variables identified in the reliability analyses were also used here with one notable exception. It was noted from the empirical data that subsystems with larger defect densities tended to have greater fractions of defects in the simple-to-isolate and correct categories. This observation is consistent with that of Jones [32] who pointed out that low-defect software may have a larger proportion of "difficult to fix" defects. On this basis, we decided to incorporate the defect density as an explanatory variable for maintainability.

In the case of the effort to correct defects, EFFFIX, shown in table 4-10, the estimated coefficients all appear with the expected signs. Fifty percent of the variation of the effort to correct defects is explained by the independent variables. Thus, the positive sign associated with the defect density, TOTDEFSL, was expected on the basis of the observation above. The negative signs associated with the context coupling variable, IMPEXP, the ratio of cascaded to direct imports, CIMPIMP, and the exports per library unit, EXPPLU, all indicate that as complexity increases, there is a shift toward more difficult defects (e.g., greater than one hour to correct). Finally, the coefficient associated with the internal import ratio, FINTIMP, is positive, indicating that as the fraction of imports which are internal to a subsystem rises, the fraction of defects taking less than one hour to correct increases.

The first column in table 4-10 shows the results for the effort to isolate. All estimated coefficients are of the expected signs except for the import/export ratio. However, because of the large associated standard error, this coefficient estimate is not inconsistent with a negative value. Forty-one percent of the variation of the effort to isolate defects is explained by the independent variables.

Environmental factor variables were not introduced into the analyses for two reasons. First, the limited number of observations cannot support additional variables. Second, an environmental factor characterizing staff experience with particular subsystems, which might be expected to play a role in maintainability, has not been available.

Table 4-10. Estimates for Maintainability

Variable	EFFISO	EFFFIX
Intercept	.02[a] (.90)[b]	2.29 (.90)
TOTDEFSL	.52 (.14)	.72 (.14)
IMPEXP	.13 (.15)	-.27 (.15)
CIMPIMP	-.47 (.10)	-.40 (.10)
FINTIMP	.27 (.06)	.14 (.07)
EXPPLU	-.05 (.18)	-.49 (.18)
R^2 [c]	.41	.50

a Parameter estimate.

b Standard error of the parameter estimate.

c Coefficient of determination.

SECTION 5

SUMMARY

We have discussed the need for technology to analyze software designs for their likely impact on software quality. We outlined a research approach, involving the construction of multivariate models that explain reliability and maintainability in terms of design characteristics and environmental factors. The progress reported thus far is in establishing a working set of definitions of reliability and maintainability, developing a representation of Ada design structure, and conducting preliminary statistical analyses.

We reemphasize that the results for both reliability and maintainability are based on static design analysis and a limited sample of 21 observations. Within these limitations, the results of the analyses are promising. Several design and environmental explanatory variables expected to influence software quality factors have been identified and analyzed.

Context coupling measures consistently show strong correlations with reliability. These measures quantify the strength of coupling of the top-level, manifest architecture (i.e., the association among compilation units) of Ada programs. The context clause visibility measure, and the internal/external import measure provide additional refinements to the top-level context coupling. The justification for the inclusion of these variables is based on our hypothesis concerning the effects on software quality of library unit structure (i.e., the visibility measure), and the work load distribution among project subteams (i.e., internal/external import measure).

A final design variable, the internal complexity measure, is brought into play to account for the impact of library unit complexity on reliability. This measure is viewed as relatively crude one which will be replaced by a more refined measure based on the fan-out structure of the library unit when appropriate analysis tools become available.

The variables introduced in addition to the context coupling variables show relatively high correlations with the context coupling variables. With the limited number of observations at this point, it is not clear if these correlations will persist for larger numbers of observations, or if they are an artifact based on spurious correlations. Theoretical arguments can be made for persistence. For example, we can argue that tradeoffs will exist between external (i.e., architectural) complexity and the internal complexity of library units. We look forward to resolving this question with additional empirical study.

It must be stressed, to an even greater degree than for reliability, that the analysis results for maintainability are preliminary. We intend to develop a polychotomous model whose dependent variable represents four levels of maintainability. But, the model immediately requires that three constant parameters be estimated leaving little room for the addition of variables and their associated parameters. Instead a dichotomous model was developed and estimated.

With the exception of the parameter associated with the import/export ratio for EFFISO, all parameters enter with the expected signs. However, the import/export ratio parameter is statistically insignificant (as evidenced by the large standard error).

Additional thought must be given to the process of maintenance, and the potential variables which may affect it. These variables may not necessarily be the same set as those which influence reliability. For example, an environmental variable influencing maintainability may be the extent to which the maintenance staff also participated in the development of the subsystem. This familiarity should make the maintenance job easier. Regarding variables, the extent to which a library unit exports declarations to compilation units in the system may be more important for maintenance than the extent to which it imports declarations. The argument here is that a person making changes to a library unit may have to trace the impact of those changes on any compilation unit to which the library unit exports declarations. This activity may take more time for library units having more complex export couplings.

In summary, the results of the analyses thus far are heartening. They have provided initial evidence in support of our hypotheses, and have opened new perspectives to be explored with additional project data during the continuation of this research.

LIST OF REFERENCES

1. Software Engineering Standards, Third Edition, October 1989, New York: The Institute of Electrical and Electronics Engineers.

2. Bowen, T. P., G. B. Wigle, and J. T. Tsai, 1985, Specification of Software Quality Attributes, RADC-TR-85-37, Vol. 1, Rome Air Development Center.

3. McCabe, T. J., December 1976, "A Complexity Measure," IEEE Transactions on Software Engineering, Vol. SE-2, No. 4, pp. 308-320.

4. Henry, S. M., and D. G. Kafura, September 1981, "Software Structure Metrics Based on Information Flow", IEEE Transactions on Software Engineering, Vol. SE-7, No. 5, pp. 510-518.

5. McCabe, T. J., and C. W. Butler, December 1989, "Design Complexity Measurement and Testing," Communications of the ACM, Vol. 32, No. 12, pp. 1415-1425.

6. Henry, S. M., and C. A. Selig, March 1990, "Predicting Source-Code Complexity at the Design Stage," IEEE Software, Vol. 7, No. 2, pp. 36-44.

7. Card, D. N., and W. W. Agresti, March 1988, "Measuring Software Design Complexity," Journal of Systems and Software, Vol. 8, No. 3, pp. 185-197.

8. Kitchenham, B., August 1989, "Measuring Software Quality," Proceedings of the First Annual Software Quality Workshop, Rochester, NY, 25 pp.

9. Munson, J. C., and T. M. Khoshgoftaar, May 1988, "The Dimensionality of Program Complexity," Proceedings of the Eleventh International Conference on Software Engineering, pp. 245-253.

10. Musa, J. D., March 1989, "Faults, Failures, and a Metrics Revolution," IEEE Software, vol. 6, No. 2, p. 85.

11. Musa, J. D., A. Iannino, and K. Okumoto, 1987, Software Reliability, New York: McGraw-Hill.

12. Grady, R. B., and D. L. Caswell, 1987, Software Metrics: Establishing a Company-Wide Program, Englewood Cliffs, NJ: Prentice-Hall.

13. Conte, S. D., H. E. Dunsmore, and V. Y. Shen, 1986, Software Engineering Metrics and Models, Menlo Park, CA: Benjamin/Cummings.

14. Kafura, D. G., and G. R. Reddy, March 1987, "The Use of Software Complexity Metrics in Software Maintenance," IEEE Transactions on Software Engineering, Vol. SE-13, No. 3, pp. 335-343.

15. Chruscicki, A., June 1989, "Software Maintainability Prediction and Assessment," Proceedings of the International Association of Science and Technology for Development (IASTED) Conference on Reliability and Quality Control, Lugano, Switzerland.

16. Agresti, W. W., February 1982. "Managing Program Maintenance,"Journal of Systems Management, pp. 34-37.

17. Rombach, H. D., and B. T. Ulery, April 1989, "Improving Software Maintenance Through Measurement," Proceedings of the IEEE, Vol. 77, No. 4., pp. 581-595

18. Gibson, V. R., and J. A. Senn, March 1989, "System Structure and Software Maintenance Performance," Communications of the ACM, Vol. 32, No. 3, pp. 347-358.

19. Sunday, D. A., 1989, "Software Maintainability: A New 'Ility'," Proceedings of the IEEE Annual Reliability and Maintainability Symposium, pp. 50-51.

20. Agresti, W. W., March 1982, "Measuring Program Maintainability," Journal of Systems Management, pp. 26-29.

21. Belady, L. A., and M. M. Lehman, 1976, "A Model of Large Program Development," IBM Systems Journal, Vol. 15, No. 3.

22. Schaefer, H., 1985, "Metrics for Optimal Maintenance Management," Proceedings of the IEEE Conference on Software Maintenance, pp. 114-117.

23. Perry, D. E., 1987, "Software Interconnection Models," Proceedings of the Ninth International Conference on Software Engineering, pp. 61-69.

24. Choi, S., and W. Scacchi, January 1990, "Extracting and Restructuring the Design of Large Software Systems," IEEE Software, Vol. 7, No. 1, pp. 66-71.

25. ANSI/MIL-STD-1815A - 1983, Reference Manual for the Ada Programming Language, American National Standards Institute, Inc.

26. Doubleday, D. L., August 1987, "ASAP: An Ada Static Source Code Analyzer Program," TR-1895, Department of Computer Science, University of Maryland.

27. Valett, J. D., and F. E. McGarry, February 1989, "A Summary of Software Measurement Experiences in the Software Engineering Laboratory," Journal of System and Software. Vol. 9, No. 2, pp. 137-148.

28. Basili, V. R., and E. E. Katz, 1983, "Metrics of Interest in Ada Development," Proceedings of the IEEE Workshop in Software Engineering Technology Transfer, pp. 22-29.

29. Murphy, J., 1973, Introductory Econometrics, Homewood, Illinois: Richard D. Irwin, Inc.

30. Card, D. N., V. E. Church, and W. W. Agresti, February 1986, "An Empirical Study of Software Design Practices," IEEE Transactions on Software Engineering, Vol. SE-12, No. 2, pp. 264-271.

31. Gurland, J., T. Lee, and P. Dahm, 1960, "Polychotomous Quantal Response in Biological Assay," Biometrics, Vol. 16, pp. 382-398.

32. Jones, T. C., 1986, Programming Productivity, New York, McGraw-Hill.

GLOSSARY

ANSI	American National Standards Institute
ASAP	Ada Static Source Code Analyzer Program
COQUAMO	Constructive Quality Model
CSC	Computer Software Component
ESPRIT	European Strategic Program for Research in Information Technology
GSFC	Goddard Space Flight Center
IEEE	Institute of Electrical and Electronics Engineering
KSLOC	Thousand Source Lines of Code
MOIE	Mission Oriented Investigation and Experimentation
MTTR	Mean Time to Repair
NASA	National Aeronautics and Space Administration
OS/360	Operating System/360
RADC	Rome Air Development Center
SAS	Statistical Analysis System
SEL	Software Engineering Laboratory

☆U.S. GOVERNMENT PRINTING OFFICE 1993-710-093-60193