Analyzing Error-Prone System Structure

Richard W. Selby, *Member, IEEE,* and Victor R. Basili, *Fellow, IEEE*

Abstract—**One central feature of the structure of a software system is the nature of the interconnections among its components (e.g., subsystems, modules). The concepts of coupling and strength have been used in the past to refer to the degree of interconnection among and within components. The purpose of this study is to quantify ratios of coupling and strength and use them to identify error-prone system structure. We use measures of data interaction, called data bindings, as the basis for calculating software coupling and strength and analyzing system structure. We selected a 148 000 source line system from a production environment for empirical analysis. We collected software error data from high-level system design through system test and from some field operation of the system. We describe the methods used for gathering data during the ongoing project and characterize the software error data collected. We apply a set of five tools to calculate the data bindings automatically and use a clustering technique to determine a hierarchical description of each of the system's 77 subsystems. A nonparametric analysis of variance model is used to characterize subsystems and individual routines that had either many or few errors or high or low error correction effort.**

The empirical results support the effectiveness of the data bindings clustering approach for localizing error-prone system structure. Routines with the highest coupling/strength ratios had 7.0 times more errors per KNCSS (1000 source statements excluding comments) than did routines with the lowest coupling/strength ratios. Subsystems with high coupling/strength ratios had routines with 4.8 times more errors per KNCSS than did subsystems with low coupling/strength ratios. The interpretations span several areas: development methodology, inspection methodology, data collection and analysis, size, coupling/strength, and system structure.

Index Terms—**Empirical measurement and evaluation, error analysis, software inspections, software metrics, system decomposition, system strength and coupling.**

I. Introduction

SEVERAL researchers have proposed methods for relating the structure of a software system to its quality (e.g., [2], [15], [11]). One pivotal step in assessing a system structure is characterization of the nature of its component interconnections using the concepts of coupling and strength. Intuitively, the *strength* in a software system is the amount of interaction *within* pieces (e.g., subsystems, modules) of a system. Correspondingly, *coupling* in a software system is the amount of interaction *across* pieces of a system. Various interpretations for coupling and strength have been proposed [21]. In this paper, we present an empirical study that evaluates the effectiveness of strength and coupling principles in identifying error-prone system structure. Our measurement of strength and coupling is based on intrasystem interaction in terms of *software data bindings* [9], [14]. Our measurement of error-proneness is based on software error data collected from high-level system design through system test; some error data from system operation are also included.

We have three primary goals for this study: 1) to quantify a measure of system structure based on interactions among components; 2) to validate the usefulness of this measure for identifying error-prone system structure; and 3) to use empirical error data in the validation of the measure which were collected and analyzed during an ongoing software project without negatively impacting the developers.

The research approach was based on the application of a data collection and analysis methodology in a large, production software environment. The use of the methodology incorporates definition of the required data, collection of the data, and appropriate data analysis and interpretation. The research project was conducted in three phases, and they roughly corresponded to the activities of data definition, collection, and analysis and interpretation.

A. Selected Software Project

The software project selected for study is the next release of an internal software library tool (see Fig. 1). The previous system release contains approximately 113 000 source lines. The production of the next release requires the development or modification of approximately 40 000 source lines. Hence, the total size of the next system release is approximately 148 000 source lines. The analysis of the error-proneness in the system distinguishes between code that was unchanged in the current release and code that was changed. This is because experience has shown that there are different error rates in base code versus modified code, and that the structure of the interactions among routines is adjusted to accommodate new functionality [4]. Note that the design standards on the current system release and previous releases were consistent in terms of interactions among routines and were uniformly enforced.

The system is written in four languages: a high-level programming language similar to PL/I, a language for operating system executives, a user-interface specification language, and an assembly language. The static source code metrics discussed later, including the data bindings analysis, pertain to only the system portion written in the high-level source language. This portion constitutes approximately 70% of the system and the vast majority of the system logic and intrasystem interactions. Project duration, including system and field test, spanned approximately 16 months and maximum staffing included 23 persons. The error data analyzed were collected during several years of the system's history, as well as during the current project.

B. System Characterization

There are 163 source code files in the system containing a total of 451 source code *routines.* A routine is a main program, procedure, or function. The number of routines per source code file varies from 1 to 21. On the average, there are 2.8 routines per

Manuscript received October 25, 1988; revised September 27, 1990. Recommended by L. A. Belady. This work was supported in part by IBM under the Shared University Research (S.U.R.) program, the National Science Foundation under Grant CCR-8704311 with cooperation from the Defense Advanced Research Projects Agency under Arpa Order 6108, Program Code 7T10, the National Science Foundation under Grant DCR-8521398, and the Air Force Office of Scientific Research under Contract AFOSR-F49620-80-C-001.

R. W. Selby is with the Department of Information and Computer Science, University of California, Irvine, CA 92717.

V. R. Basili is with the Institute for Advanced Computer Studies and the Department of Computer Science, University of Maryland, College Park, MD 20742.

IEEE Log Number 9040980.

0098–5589/91/0200–0141$01.00 © 1991 IEEE

Source code routines	451
Source code files	163
Subsystems	77
System size in lines of code including comments	148,754
System size in source statements excluding comments	82,806

Fig. 1. Characterization of the software system analyzed.

source code file. There are 77 executable features in the system, referred to as *subsystems* in the paper. These subsystems can be thought of as groups of routines collected together to form functional features of the overall system. The number of source files linked together to form a subsystem varies from 1 to 82. On the average, 26.3 source files are linked together into a subsystem. The same source file is bound into 12.4 different subsystems on the average. Subsystems averaged 19 000 source lines including comments and 10 749 source statements excluding comments.

Section II describes the data definition, collection, and analysis methodology used. The software error data collected are summarized in Section III, and the detailed error data are presented in Appendixes B and C. The data bindings software analysis and supporting tools are described in Section IV. The data analysis appears in Section V, and the interpretations and conclusions appear in Section VI. Appendix A contains definitions of the error-related terminology. Appendix D presents some general observations and recommendations on data collection and analysis.

II. Data Collection

The following three subsections give an overview of the data definition, collection, and analysis methodology, an explanation of the metric vector concept, and a description of the underlying data collection forms. Appendix D summarizes the effectiveness of the data collection process in gathering data during the software project. Appendix D also presents some lessons learned and recommendations based on the use of the data collection and analysis methodology. Note that the data was collected and analyzed at the same time the project took place. An important goal was to minimize the impact of the data collection process on the developers.

A. Data Collection and Analysis Methodology

The *goal-question-metric paradigm* [10], [7], [20], [1] defines a methodology for data collection and analysis and results in a set of software product and process metrics, a "metric vector" [3], sensitive to the cost and quality goals for a particular environment. There are several steps in the methodology spanning software metric definition, collection, analysis, and interpretation. The data collection and analysis methodology consists of seven steps:

1) Define the goals of the data collection and analysis.
2) Refine the goals to determine a list of specific questions.
3) Establish appropriate metrics and data categories.
4) Plan the layout of the study and the statistical analysis methods.
5) Design and test the data collection scheme.
6) Perform the investigation concurrently with data collection and validation.
7) Analyze and interpret the data in terms of the goal-question framework.

The first three steps in the methodology express the purpose of an analysis, define the data that needs to be collected, and provide a context in which to interpret the data. The formulation of a set of goals constitutes the first step in a management or research process. The goals outline the purpose of the study in terms of software cost and quality aspects. Refinement of the goals occurs until they are manifested in a set of specific questions. The questions define the goals and provide the basis for pursuing the goals. The information required to answer the questions determines the development process and product metrics needed. The organization of the defined metrics results in a set of software metrics, referred to as a "metric vector."

The following four steps involve analysis planning and data collection, validation, analysis, and interpretation. Before collecting the data, the researchers outline the data analysis techniques. The appropriate analysis methods may require an alternate layout of the investigation or additional pieces of data to be collected. The investigators then design and test the data collection method; they determine the information that can be automatically monitored and customize the data collection scheme to the particular environment. The data collection plan usually includes a mixture of collection forms, automated measurement, and personnel interviews. The investigators then perform the data collection accompanied by suitable data validity checks. After preliminary analysis to screen the data, they apply the appropriate statistical and analytical methods. They organize the statistical results and interpret them with respect to the goal-question framework. The analysis of the collected data can sometimes lead to the expansion of the original sets of questions, possibly resulting in more goal areas. Once all seven methodology steps have been completed, researchers can apply another iteration of the methodology with a new set of goals.

B. Metric Vector

The set of metrics defined was described in terms of a "metric vector" [3], consisting of seven dimensions: {effort, nonerror changes, errors, size, data use, execution, environment}. These seven dimensions are defined as follows: 1) effort—the time expended in producing the software product; 2) nonerror changes—the modifications made to the product; 3) errors—the mistakes made during development or maintenance that require correction; 4) size—the various aspects of the product bulk and complexity; 5) data use—the various aspects of the program's use of data; 6) execution—information about the execution of the program; and 7) environment—a quantitative description of the development and maintenance environment. Each of these dimensions has a variety of metrics associated with it. These metrics depend upon the specific goals and questions articulated for the project. Both a metric vector containing all metrics defined and a vector containing a minimal number of metrics to collect were outlined for this study.

The metric data was collected in two ways: 1) data collection forms, which are discussed in the next section, and 2) automated data bindings analysis, which is discussed in Section IV.

C. Data Collection Forms

The collection of the data was conducted so as to affect minimal interference on the project personnel. We obtained a consensus from project management and development process coordinators on what metrics were already collected, what new metrics could be collected, and how they should be collected. The metrics defined by the methodology were categorized according to their natural collection source. We worked within existing, established procedures using existing forms, as far as possible, to collect the new data. Fig. 12 in Appendix A outlines the data

collections forms used in the various development phases. The collection of the metrics was based on a variety of sources, including the existing set of data collection forms: inspection forms, error summary worksheets (ESW), system trouble reports (STR), and trouble reports (TR). These data sources are summarized below.

1) Inspections: Two kinds of formal inspections are held during development: design inspections and engineering inspections [12], [13]. Design inspections are held during the high-level and low-level design phases. Engineering inspections are code inspections that are held after the completion of unit testing. Inspection forms represent all data recorded during formal inspections and during rework activity following the inspections.

2) Error Summary Worksheets: The Error Summary Worksheet (ESW) form was introduced for the purpose of recording error and change data during the coding, unit testing, and primitive/transaction testing phases (primitive/transaction testing is similar to integration testing).

3) System Trouble Reports: System Trouble Report forms (STR's) are used during system testing. The collection form is the same as the trouble report (TR) form.

4) Trouble Reports: These are problems reported against working, released code. They are typically user-reported. Trouble report forms are also used to report errors found by developers during field testing. The corrections are either implemented immediately or included as part of the development of the next release.

5) Surveys and Interviews: The collection and validation of certain data items are supported by the use of developer surveys. A researcher interviewed developers to acquire the survey information.

III. Characterization of Software Error Data

This section summarizes the software error data collected from the trouble reports (TR's) and software inspections. For further detail, see the tables in Appendixes B and C that contain the inspection error data and the trouble report error data. The error terminology used in this paper is defined in Appendix A. Throughout this paper, the term "error" is used to indicate the root cause of a problem in the software system. Rediscoveries of the same error are not counted as separate errors.

There were 770 software errors reported from the 170 inspections with an average overall detection rate of 4.5 (= 770/170) errors per inspection [see Appendix B, parts 1)–3)]. An analysis of the errors showed that inspections were more effective in finding high severity errors than low severity errors (56% = 433/770 versus 44% = 337/770) [see Appendix B, part 2)] and that design inspections were more effective than engineering inspections. That is, design inspections detected 58% (= 450/770) of all inspection-detected errors and 63% (= 275/433) of the major severity errors [see Appendix B, part 2)]. Design inspections had an overall detection rate three times (= 8.3/2.8) greater than engineering inspections and a major severity detection rate of three and a half times (= 5.1/1.4) greater [see Appendix B, part 3)].

With regard to inspection error type, twice (= 111/55) as many of the "missing" errors were of major severity than of minor severity, and one and a half times (= 31/18) as many of the "extra" errors were of minor severity than of major severity [see Appendix B, part 5)].

There were 54 valid trouble reports, and 70% (= 32/46) of the TR-errors were in the "wrong" category, which was consistent with the results for inspection-reported errors (inspections had 70% = 501/716 "wrong" errors) [see Appendix B, part 5) and Appendix C, parts 1) and 2)]. As might be expected, there were proportionately more "extra" errors and fewer "missing" errors found in inspections than there were in TR's [see Appendix B, part 5) and Appendix C, part 2)]. (In inspections, 7% = 49/716 of the errors were "extra" and 23% = 166/716 were "missing"; while in TR's, 2% = 1/46 of the errors were "extra" and 28% = 13/46 were "missing".)

With regard to TR-error type, isolating a TR-error required almost twice (= 6.3/3.7) as much effort as did fixing it, and correcting (both isolating and fixing) a "wrong" error required more effort than did correcting a "missing" error (10.9 versus 8.1 hours) [see Appendix C, part 3)]. Correspondingly, the isolation of a "wrong" TR-error required the most effort. Note that the isolation costs would have been zero if the errors reported on TR's had been found during inspections.

The effort for fixing a "missing" TR-error could be counted as development effort, as opposed to error correction effort. With this interpretation, the error correction effort for "missing" errors includes only the isolation effort. In this view, the error correction effort (isolation plus fix) for "wrong" errors is 2.6 times (= 10.9/4.2) the correction effort (isolation only) for "missing" errors [see Appendix C, part 3)]. Hence, it was less costly overall to leave out a design or code segment, rather than to include an incorrect one.

IV. Data Bindings Analysis

A. Clustering with Data Bindings

One primary goal for this study was to investigate the relationship of "software data bindings" to software errors. "Data bindings" are measures that capture the data interaction across portions of a software system. The theoretical background for the measures is described in [14]. Earlier studies have revealed insights about the usefulness of data bindings in the characterization of software systems and their errors [9], [14]. In order to describe the data bindings analysis process applied, we first introduce some terminology (see also [14]).

An *actual data binding* is defined as an ordered triple (p, x, q) where p and q are procedures and x is a variable within the static scope of both p and q, where p assigns a value to x and q references x. The calculation of actual data bindings counts those instances where there may be a flow of information from p to q via the variable x. The possible orders of execution for p and q are not considered. That is, there may be other factors (e.g., control flow conditions) which would prevent such communication. Although stronger levels of data bindings have been defined, we calculated *actual data bindings* in this study because they offer an adequate measure of similarity while not requiring complex data flow analysis that stronger levels need. Essentially, we are erring in the direction of safety (as done, for example, by code optimizers) by assuming that procedures may influence one another unless we can show otherwise.

First, we calculated the actual data bindings in the system. Then, we applied an iterative clustering technique to the data bindings information to produce a hierarchical description for the software system (see Fig. 2). The clustering takes place in a bottom-up manner. The process iteratively creates larger and larger clusters, until all the elements have collapsed into a

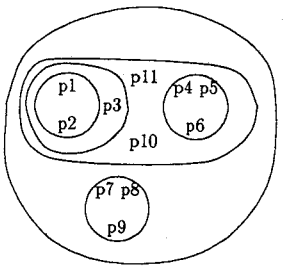


Fig. 2. Example hierarchical cluster based on software data bindings. Routines (e.g., procedures, functions) are denoted by p_i, and clusters are denoted by circles. The smaller clusters are relatively tighter (and form earlier), while the larger clusters are relatively looser (and form later). The clusters define a system hierarchy in the form of a tree: the smaller clusters at the leaf nodes and the largest cluster at the root node.

single cluster. The elements in the clusters are the procedures and functions in the system. The elements with the greatest interaction, in terms of actual data bindings, cluster together first. Clustering techniques have been applied previously to partition a large system into subsystems in [2]. Hierarchical clusters have been formally defined in [17].

B. Data Bindings Analysis Software

A set of five software tools was developed to calculate these hierarchical, data bindings clusters and applied to the 77 subsystems in the selected project. (As defined in Section I, a subsystem in the selected project is a large collection of routines that are linked together to form an executable system feature; subsystems averaged 19 000 source lines.) Four of the five tools are language independent; the other tool—a major one—is language dependent. All of the tools are written in the C programming language [18].

The tools determine the data bindings that occur among the procedures and functions in the source code and then use them in cluster analysis as a measure of similarity. The tools therefore conduct source code analysis and cluster analysis. Due to the fact that almost all large systems consist of many separately compiled units, there must also be a program to gather the information from several compilation units and combine it, somewhat similar to a linker.

The five tools convert the source code into a hierarchical system description. The first two programs correspond roughly to the standard compile and link paradigm. The first program, *source_bind,* reads the source code and produces a file containing information about the variable usage of the procedures and functions, from which the data bindings will be determined. This first program is built specifically for the source language, and hence, is language dependent. The second program, *link_bind,* takes the outputs from one or more runs of *source_bind* and combines the information together in much the way that a link editor does, giving each data object a unique name. The third program, *matrix_bind,* takes the output of *link_bind* and builds a matrix that contains a row and a column for each procedure and function in the source code. The matrix entries are the number of actual data bindings between a pair of procedures or functions. The fourth program, *fold_bind,* reads the output of *matrix_bind* and creates a dissimilarity matrix (a nonnegative, real, symmetric matrix with zeros on the diagonal [17], [14]) that contains the binding information in a format that cluster programs require. The fifth program, *cluster_program,* reads the output of *fold_bind* and produces a description of the system as a tree. The tree gives a view of the hierarchy of the system with respect to data usage.

V. Data Analysis

The data collection and analysis methodology was successful in producing a wide range of statistically significant results.

1) Terminology: Throughout the analysis and interpretation, we use the terms *subsystems* and *routines.* A *routine* is a main program, procedure, or function. There are a total of 451 source code routines in the system. A *subsystem* is a large set of routines that are linked together to form an executable system feature. Any routine may be a part of several features. There are 77 executable features in the system, and they average 19 000 source lines including comments and 10 749 source statements excluding comments. A routine is linked into 12.4 subsystems on the average.

Rather than analyzing the system as a whole, we used the analysis tools described in Section IV to produce hierarchical descriptions for each of the 77 subsystems (see Fig. 2). The hierarchical descriptions are rooted, connected trees that reflect the internal subsystem structure. Each routine in a subsystem occurs as a leaf node in the tree exactly once. Subtrees indicate groupings of routines that form natural *clusters* based on the data bindings criteria. There is a one-to-one correspondence between subtrees and clusters. A cluster can contain either routines or other clusters. In other words, the root node of a subtree can have as its children either leaf nodes (i.e., routines) or the root node of another subtree (i.e., a subset of its own routines that form a smaller cluster).

In the software system being analyzed, a routine may be linked into more than one subsystem. Each of the 77 subsystems has a separate hierarchical description. Therefore, a routine appears in the hierarchical description of each subsystem into which it is linked. A routine may cluster with different sets of routines in different subsystems.

Each cluster in a subsystem has a numerical value associated with it which reflects the binding nature of its routines. This numerical value is the following ratio:

$$\frac{\text{number of data bindings between routines within the cluster and those outside of it}}{\text{number of data bindings between routines within the cluster}}.$$

In other words, the numerical value measures the binding between routines in a cluster and those outside of it (i.e., "external binding") in relation to the binding between routines within a cluster (i.e., "internal binding"). This number is interpreted as the following ratio:

$$\frac{\text{the coupling of the cluster with other clusters in the subsystem}}{\text{the internal strength of the cluster}}.$$

That is, the number captures the *coupling/strength ratio* for a cluster of routines within a subsystem. The clusters are formed in a bottom-up manner in the analysis process. The clusters with the lowest coupling/strength ratios form in the first iteration, the clusters with the next lowest ratios form in the second iteration, and so forth. In earlier work, alternate measures having less straightforward interpretations were used to capture the binding nature among routines [14].

The lower a cluster's coupling/strength ratio is, the lower the relative coupling with other clusters and the higher the relative strength of binding within the cluster. The higher a cluster's coupling/strength ratio is, the higher the relative coupling with

other clusters and the lower the relative strength of binding within the cluster. Software engineering principles generally suggest that it is desirable to have low coupling and high strength, which in this context means a low coupling/strength ratio [21].

The data bindings analysis produced 77 trees corresponding to the subsystems which included a total of 4211 clusters containing 5045 routine occurrences. Recall that there were a total of 451 routines in the system—each routine was bound into 12.4 subsystems on the average (see Section I). We calculated three different measures based on the clusters resulting from the data bindings analysis. For each routine occurrence, we calculated the following two measures. *Routine coupling/strength ratio*—The coupling/strength ratio of the first cluster to form that included the routine as a member. This metric is intended to capture the relationship of a routine to other routines in a subsystem in terms of coupling and strength. *Routine location in subsystem's data binding tree*—The depth in the tree of the first subtree (i.e., cluster) to form that included the routine as a member. More precisely, it is the depth in the tree of the root of that subtree. This metric is intended to characterize the location of a routine in a data binding tree. This location information is useful to know when data binding trees are used as an alternative form of system documentation. For each subsystem, we calculated the following measure. *Subsystem coupling/strength ratio*—The median of the coupling/strength ratios for the clusters within the subsystem. We use a nonparametric statistic here, i.e., a median, because the coupling/strength ratios are relative measures. This metric is intended to characterize the overall coupling and strength within a subsystem.

A. Data Analysis Method

A nonparametric analysis of variance model was used to characterize subsystems and routines that had either many/few errors or high/low development effort spent in error correction.

Primary Factors and Interactions: The analysis of variance model considered numerous factors simultaneously [19]. When defining the levels for some of the factors, we used nonparametric statistics (e.g., quartiles) since the coupling/strength ratios are relative measures and the data bindings trees have different overall depths. Some thresholds between factor levels (e.g., routine size of 142 statements) were selected to create groups approximately equal in size. Subsystem size and routine size are included as factors in the analysis because earlier analyses have indicated a relationship between size and software effort and error data (e.g., [5], [8], [22]). Also included as a factor is whether or not a routine was changed as part of the current system release, because earlier studies have indicated that system structure varies between base code and modified code [4].

The primary factors and their respective levels in the model were as follows. 1) *Subsystem size:* Small—Subsystems with less than or equal to 6090 source statements, excluding comments. Large—Subsystems with greater than 6090 source statements, excluding comments. 2) *Subsystem coupling/strength ratio:* Low—Subsystems with a coupling/strength ratio below the median. High—Subsystems with a coupling/strength ratio above the median. 3) *Individual subsystem's attributes:* One level for each of the 77 subsystems. 4) *Routine size:* Small—Routines with less than or equal to 142 source statements, excluding comments. Large—Routines with greater than 142 source statements, excluding comments. 5) *Routine coupling/strength ratio:* a_Highest—The uppermost quartile of the coupling/strength ratios for the clusters in a subsystem. b_High—The next lower quartile of the coupling/strength ratios for the clusters in a subsystem. c_Low—The next lower quartile of the coupling/strength ratios for the clusters in a subsystem. d_Lowest—The lowest quartile of the coupling/strength ratios for the clusters in a subsystem. 6) *Routine location in subsystem's data binding tree:* a_Root—The uppermost quartile (nearest the root of the tree) of the clusters in a subsystem's data binding tree (quartiles are based on the number of nodes in a tree). b_Shallow—The next lower quartile of the clusters in a subsystem's data binding tree. c_Deep—The next lower quartile of the clusters in a subsystem's data binding tree. d_Deepest—The lowest quartile (furthest from the root of the tree) of the clusters in a subsystem's data binding tree. 7) *Whether or not routine was changed as part of the current system release:* Yes—Routine was changed; code was added or modified. No—Routine was unchanged from previous release.

Four two-way interactions were also included in the model. 8) Interaction of subsystem size with subsystem coupling/strength ratio. 9) Interaction of routine size with routine coupling/strength ratio. 10) Interaction of routine size with routine location in subsystem's data binding tree. 11) Interaction of routine coupling/strength ratio with routine location in subsystem's data binding tree.

Dependent Variables: There were four dependent variables examined with the analysis of variance model. 1) *Total errors*—The total number of inspection, trouble report (TR), system trouble report (STR), and error summary worksheet (ESW) errors in a routine. 2) *Total errors per KNCSS*—The total number of inspection, TR, STR, and ESW errors in a routine per 1000 source statements excluding comments (i.e., "noncommentary source statements"). 3) *Error correction effort*—The total amount of effort (in hours) spent correcting TR and ESW errors in a routine. 4) *Error correction effort per KNCSS*—The total amount of effort (in hours) spent correcting TR and ESW errors in a routine per 1000 noncommentary source statements.

In general, the discussion will focus on the errors per KNCSS and the error correction effort per KNCSS measures of the routines as opposed to the absolute numbers. This factors out possible underlying correlations between size and number of errors or amount of error correction effort. The statistics for all four measures are reported, however. The measure of size used in the primary factors and dependent variables is noncommentary source statements, as opposed to source lines including comments. This is because there are wide variations in the proportion of comments across the routines and there tend to be relatively few major errors in comments compared to their rate in executable code. The discussion will tend to highlight results that demonstrated a statistically significant difference, as opposed to those where there was no statistical difference. A primary result in each subsection will be italicized for emphasis.

B. Subsystem Characterization

In the source code portions of the system (see Section I), there was a total of 299 distinct errors recorded from inspections, error summary worksheets (ESW's), system trouble reports (STR's), and trouble reports (TR's). Data on the effort required for error correction were available for 204 distinct errors recorded on ESW's and TR's. As mentioned in Section III, these errors are unique root causes of problems in the software system. Rediscoveries of the same error are not counted as separate errors. In the subsequent figures, all inspection, ESW, STR, and TR errors are counted equally.

Subsystem coupling/ strength	Errors				Error correction hours			
	per KNCSS		Total		per KNCSS		Total	
	Mean	Std	Mean	Std	Mean	Std	Mean	Std
High	2.89	7.87	0.44	0.99	4.45	11.10	0.88	2.69
Low	0.60	2.20	0.15	0.52	1.55	7.62	0.42	2.39
Overall	2.40	7.10	0.38	0.92	3.82	10.52	0.78	2.63

Fig. 3. Distribution of errors and error correction effort by subsystem coupling/strength ratios.

Subsystem size	Errors				Error correction hours			
	per KNCSS		Total		per KNCSS		Total	
	Mean	Std	Mean	Std	Mean	Std	Mean	Std
Large	2.82	7.30	0.44	0.99	4.45	11.05	0.88	2.66
Small	0.86	6.09	0.14	0.49	1.57	7.91	0.42	2.49
Overall	2.40	7.10	0.38	0.92	3.82	10.52	0.78	2.63

Fig. 4. Distribution of errors and error correction effort by subsystem size.

In the following sections we analyze the number of errors and the error correction effort in the subsystems. The characterization of the subsystems is based on subsystem coupling/strength ratio, subsystem size, and interactions across these two factors.

Subsystem Coupling/Strength Ratio: Fig. 3 presents the errors and error correction effort in the routines in subsystems with different coupling/strength ratios. This figure and the following analogous figures give the means and standard deviations for 1) the number of errors per 1000 noncommentary source statements (KNCSS), 2) the number of errors, 3) the error correction effort per KNCSS, and 4) the error correction effort in the routines. Subsystem coupling/strength ratio was a statistically significant factor with respect to errors per KNCSS and error correction hours per KNCSS ($\alpha < 0.0002$ and $\alpha < 0.002$, respectively).[1] The subsystems with high coupling/strength ratios had routines that averaged 2.89 errors per KNCSS and 4.45 error correction hours per KNCSS, which were greater than the averages of 0.60 errors per KNCSS and 1.55 error correction hours per KNCSS for the other subsystems. *Subsystems with high coupling/strength ratios had routines with 4.8 (= 2.89/0.60) times as many errors per KNCSS than did subsystems with low coupling/strength ratios.*

Subsystem Size: Fig. 4 presents the errors and error correction effort in the routines in subsystems with different sizes. Subsystem size was a statistically significant factor with respect to errors per KNCSS and error correction hours per KNCSS ($\alpha < 0.0001$ for both). *The subsystems with large size had routines that averaged 2.82 errors per KNCSS and 4.45 error correction hours per KNCSS, which were greater than the small subsystem averages of 0.86 errors per KNCSS and 1.57 error correction hours per KNCSS.* A plot of errors per KNCSS versus subsystem size appears in Fig. 5.

Interactions Across Subsystem Coupling/Strength Ratio and Size: Fig. 6 presents the errors and error correction effort in the routines in subsystems with different coupling/strength ratios and different sizes. Combining different subsystem coupling/strength ratios and different sizes did not result in a statistically significant interaction for either errors per KNCSS or error correction hours per KNCSS ($\alpha > 0.05$ for both). However, both individual factors are statistically significant for the two measures of error-proneness (see the previous two subsections), resulting in the error-pronenesses averages to be notably higher for large subsystems with high coupling/strength ratios since the individual effects are combined. These subsystems had routines that averaged 3.04 errors per KNCSS, as compared to the combined average of 0.92 (= (1.50 + 0.74 + 0.53)/3) errors per KNCSS for the other subsystems. The large subsystems with high coupling/strength ratios had routines that averaged 4.73 error correction hours per KNCSS, as compared to the combined average of 1.65 (= (1.79 + 1.72 + 1.45)/3) error correction hours per KNCSS for the other subsystems. *Large subsystems with high coupling/strength ratios had routines with 5.7 (= 3.04/0.53) times as many errors per KNCSS than did small subsystems with low coupling/strength ratios.*

C. Routine Characterization

In the following sections we analyze the number of errors and the error correction effort in the routines. The characterization of the routines is based on routine coupling/strength ratio, routine size, routine location in the data binding tree, and interactions across these three factors. As mentioned in Section V-B there were 299 distinct errors, counting all inspection, ESW, STR, and TR errors equally; 204 of them had data on error correction effort.

Routine Coupling/Strength Ratio: Fig. 7 presents the errors and error correction effort in the routines with different coupling/strength ratios. As before, this figure and the following analogous figures give the means and standard deviations for 1) the number of errors per 1000 noncommentary source statements (KNCSS), 2) the number of errors, 3) the error correction effort per KNCSS, and 4) the error correction effort in the routines.

The routine coupling/strength ratio statistically affected the errors per KNCSS and error correction hours per KNCSS in the routines ($\alpha < 0.0001$ for both). The routines in coupling/strength region a_HIGHEST had the most errors per KNCSS (an average of 4.14) and the highest error correction hours per KNCSS (an average of 9.15). In terms of errors per KNCSS, the routines with coupling/strength ratios in region b_HIGH had the second most and those in region c_LOW had the third most.[2]

In terms of error correction hours per KNCSS, the routines with coupling/strength ratios in either region b_HIGH or c_LOW had the second most, and these two regions were not statistically different. Those routines in region d_LOWEST had the fewest errors per KNCSS (an average of 0.59) and the least error correction hours per KNCSS (an average of 0.42).[3] *The routines with the highest coupling/strength ratios had 7.0 (= 4.14/0.59) times as many errors per KNCSS and 21.7 (= 9.15/0.42) times as many error correction hours per KNCSS than did routines with the lowest coupling/strength ratios.* These results empirically support the software engineering principle of desiring low coupling and high strength.

Routine Size: Fig. 8 presents the errors and error correction effort in the routines with different sizes. The routine size statistically affected the errors per KNCSS and error correction hours per KNCSS for the routines ($\alpha < 0.0001$ for both). *Routines of large size had an average of 5.08 error correction*

[1] The F-test significance levels reported in this and later sections are based on the use of Type IV partial sums of squares [19]. Any statistical difference discussed will at least be significant at the $\alpha < 0.05$ level, unless otherwise noted.

[2] Even though the sample mean—a parametric statistic—in Fig. 7 seems to show that c_LOW is greater than b_HIGH in terms of errors per KNCSS, the nonparametric test—a "ranking" statistic—indicates that b_HIGH is actually greater.

[3] All multiple comparison results, such as the one in the previous four sentences, were conducted with Tukey's multiple comparison statistic [19], [16]. All of the pairwise statistical comparisons of these four categories are statistically significant at the $\alpha < 0.05$ level simultaneously.

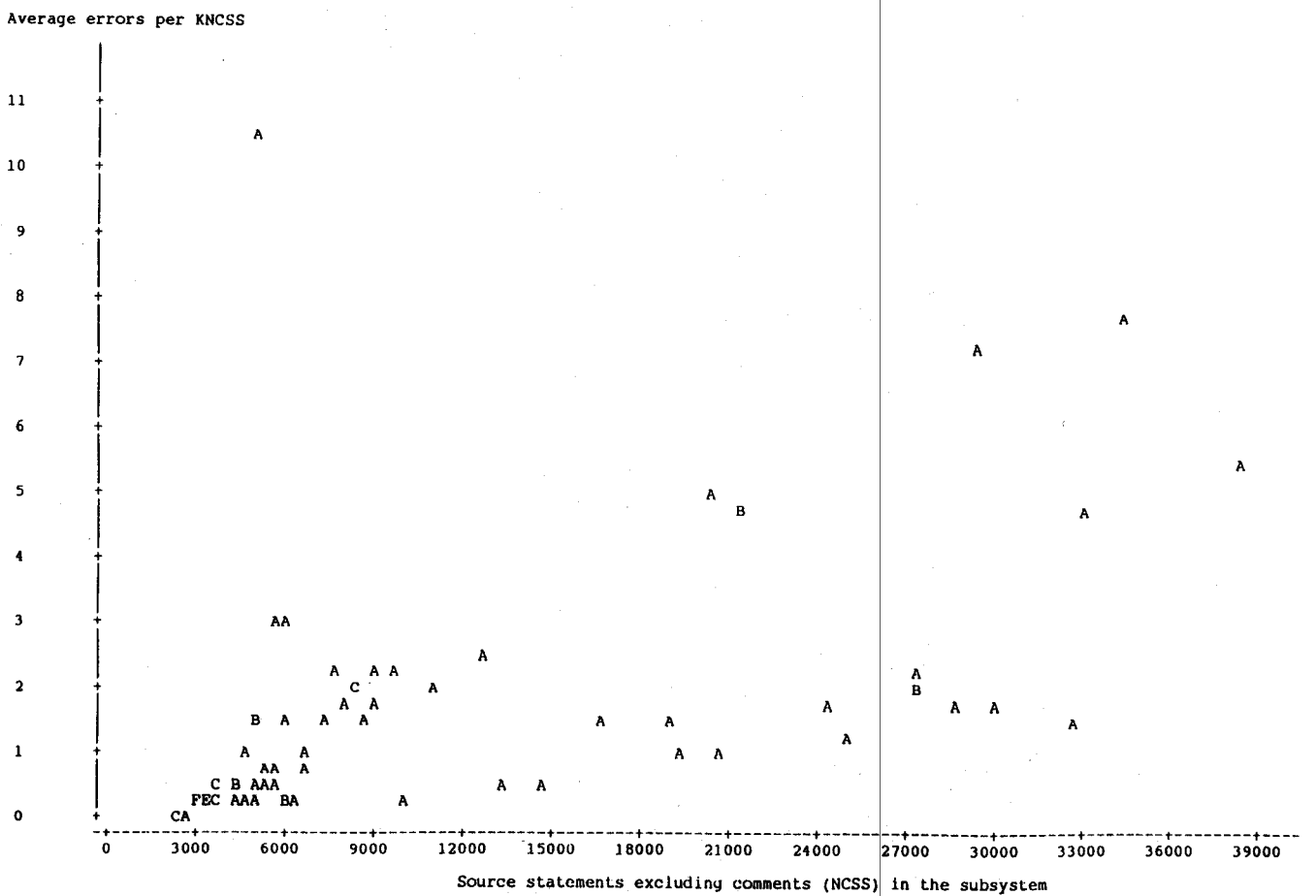


Fig. 5. Relationship between errors per KNCSS (1000 noncommentary source statements) in the routines and subsystem size. *Legend:* A = 1 observation, B = 2 observations, etc.

Subsystem coupling/ strength	Subsystem size	Errors				Error correction hours			
		per KNCSS		Total		per KNCSS		Total	
		Mean	Std	Mean	Std	Mean	Std	Mean	Std
High	Large	3.04	7.60	0.47	1.02	4.73	11.33	0.93	2.70
	Small	1.50	10.00	0.13	0.46	1.79	8.10	0.43	2.51
Low	Large	0.74	2.53	0.17	0.56	1.72	7.26	0.44	2.21
	Small	0.53	2.00	0.14	0.51	1.45	7.80	0.42	2.48
Overall		2.40	7.10	0.38	0.92	3.82	10.52	0.78	2.63

Fig. 6. Distribution of errors and error correction effort across subsystem coupling/strength ratios and subsystem size.

Routine coupling/ strength	Errors				Error correction hours			
	per KNCSS		Total		per KNCSS		Total	
	Mean	Std	Mean	Std	Mean	Std	Mean	Std
a_Highest	4.14	8.89	0.59	1.04	9.15	16.09	1.94	4.20
b_High	2.14	6.00	0.34	0.74	3.45	10.10	0.72	2.54
c_Low	2.81	8.69	0.44	1.18	2.65	7.03	0.49	1.61
d_Lowest	0.59	2.36	0.15	0.49	0.42	2.00	0.06	0.29
Overall	2.40	7.10	0.38	0.92	3.82	10.52	0.78	2.63

Fig. 7. Distribution of errors and error correction effort by routine coupling/strength ratios.

hours per KNCSS which was more than did those of small size (an average of 2.15 error correction hours per KNCSS). In terms of errors per KNCSS, routines of large size had more errors per KNCSS than did those of small size. However, note that if a parametric statistical test is applied instead of the nonparametric "ranking" statistical test (which is used throughout the paper), then the routines of small size had more errors per KNCSS than

Routine size	Errors				Error correction hours			
	per KNCSS		Total		per KNCSS		Total	
	Mean	Std	Mean	Std	Mean	Std	Mean	Std
Large	2.18	4.74	0.48	0.98	5.08	12.72	1.22	3.38
Small	2.69	9.35	0.24	0.80	2.15	6.14	0.20	0.61
Overall	2.40	7.10	0.38	0.92	3.82	10.52	0.78	2.63

Fig. 8. Distribution of errors and error correction effort by routine size.

did those of large size. The interpretation of this result is that the median errors per KNCSS of large routines is greater than the median of small routines, but the mean errors per KNCSS of small routines is greater than the mean of large routines. *This basically says that a relatively minor portion of the small routines were much more error-prone than the large routines, while the majority of them were not (which is supported by the large standard deviation of 9.35).* A separate study of a different environment has indicated, however, that smaller routines may be more error-prone than larger routines [6]. A third study of another environment has also indicated that smaller routines may be more error-prone than larger routines [22]. A plot of errors per KNCSS versus routine size appears in Fig. 9.

Routine Location in Data Binding Tree: Fig. 10 presents the errors and error correction effort in the routines with different data binding tree locations. The routine location in the data binding tree statistically affected the errors per KNCSS and error correction hours per KNCSS in the routines ($\alpha < 0.015$ and $\alpha < 0.0001$, respectively). *Routines in tree location*

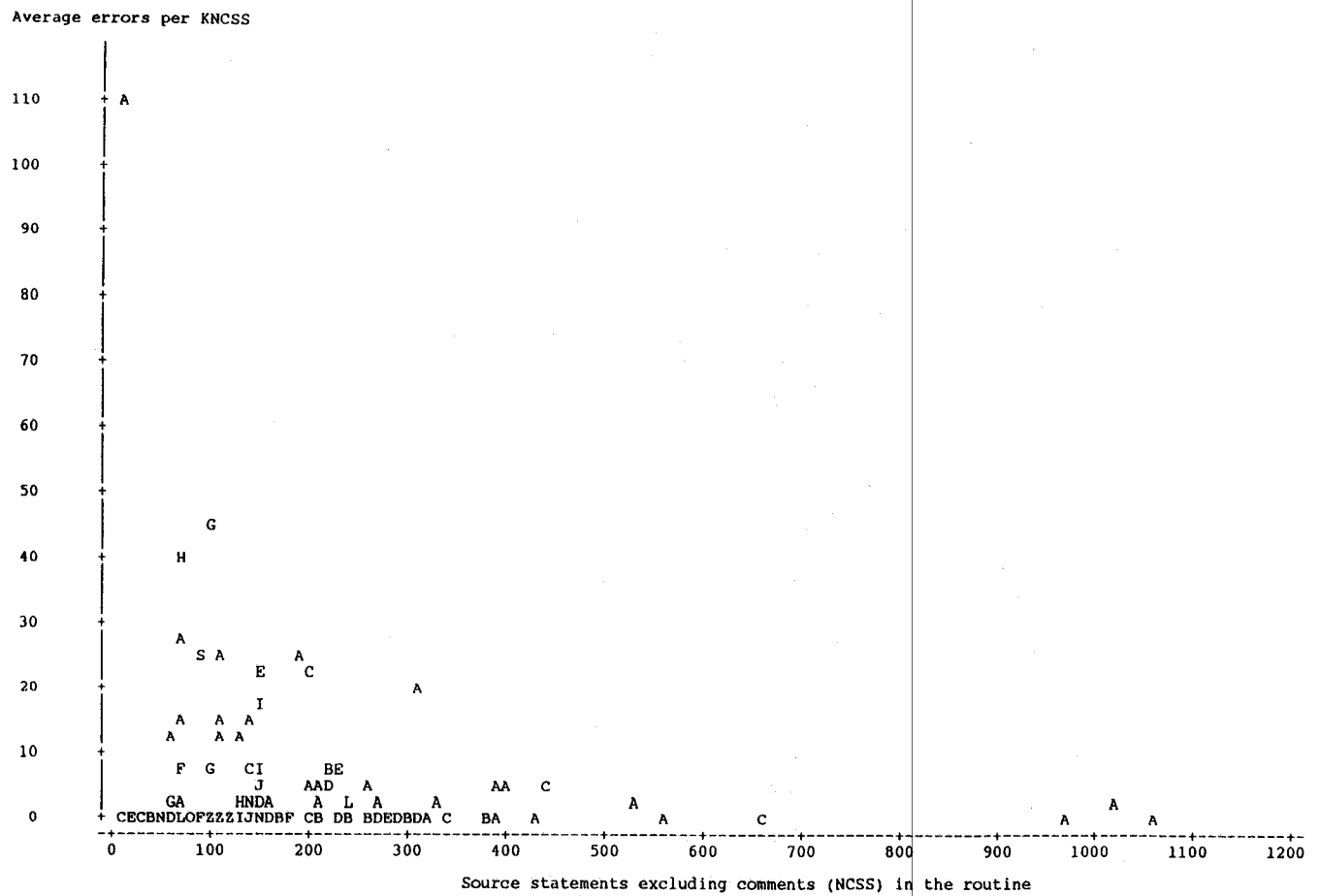


Fig. 9. Relationship between errors per KNCSS (1000 noncommentary source statements) in the routines and routine size. *Legend:* A = 1 observation, B = 2 observations, etc. *Note:* 50 observations hidden behind Z's.

Routine tree location	Errors				Error correction hours			
	per KNCSS		Total		per KNCSS		Total	
	Mean	Std	Mean	Std	Mean	Std	Mean	Std
a_Root	1.72	5.37	0.30	0.77	2.21	7.32	0.37	1.59
b_Shallow	3.30	8.41	0.51	1.12	5.64	13.16	1.19	3.36
c_Deep	1.68	4.77	0.27	0.63	3.88	10.92	0.83	2.82
d_Deepest	2.53	8.38	0.38	0.96	2.84	7.93	0.57	1.95
Overall	2.40	7.10	0.38	0.92	3.82	10.52	0.78	2.63

Fig. 10. Distribution of errors and error correction effort by routine location in data binding tree.

region b_SHALLOW had the most errors per KNCSS (an average of 3.30) and the most error correction hours per KNCSS (an average of 5.64). Routines in tree location region c_DEEP had the second most errors per KNCSS and the second most error correction hours per KNCSS. Routines in region d_DEEPEST had the third most in each measure, and those in region a_ROOT had the fewest errors per KNCSS (an average of 1.72) and the fewest error correction hours per KNCSS (an average of 2.21). One interpretation for there being fewer error correction hours per KNCSS in regions a_ROOT and d_DEEPEST may be the following. The structure of the system at the highest level (i.e., initial stages of problem decomposition) and the lowest level (e.g., formulation of abstract data types) may be better understood than the intermediate levels of system development. The effect of the less understood intermediate levels is compounded in larger subsystems, as was seen in Section V-B.

Interactions Across Routine Coupling/Strength Ratio, Size, and Location in Data Binding Tree: Fig. 11 presents the errors and error correction effort in the routines with different coupling/strength ratios and different data binding tree locations. All of the three two-way interactions (routine coupling/strength ratio with routine size, routine coupling/strength ratio with routine tree location, routine size with routine tree location) statistically affected the errors per KNCSS and error correction hours per KNCSS for the routines (all at $\alpha < 0.0001$, except $\alpha < 0.005$ for routine size with tree location for errors per KNCSS). *Routines with the highest coupling/strength ratios (a_HIGHEST) and a location in the "central portion" of the data binding tree (b_SHALLOW or c_DEEP) had the most error correction hours per KNCSS (a combined average of 9.82 = (10.56 + 9.09)/2).*

D. Data Bindings for System Documentation and Evaluation

Dialog with project personnel regarding the data binding trees resulted in the following observations. The data binding clusterings were able to detect major system data structures. The data binding clusterings seemed to provide a different view of the system than that provided by the system documentation, which included textual documents and a calling hierarchy. Analyzing the clusters of data bindings provided insights to the development and maintenance team.

VI. Interpretations and Conclusions

In this study, we have merged three goals: 1) to apply data

Routine coupling/ strength	Routine tree location	Errors per KNCSS Mean	Errors per KNCSS Std	Errors Total Mean	Errors Total Std	Error correction hours per KNCSS Mean	Error correction hours per KNCSS Std	Error correction hours Total Mean	Error correction hours Total Std
a_Highest	a_Root	4.28	8.96	0.55	1.07	6.36	12.05	1.10	2.74
	b_Shallow	4.27	9.24	0.63	1.06	10.56	17.59	2.33	4.68
	c_Deep	2.66	5.19	0.45	0.63	9.09	16.44	2.10	4.49
	d_Deepest	1.91	2.69	0.83	1.18	0.38	0.54	0.17	0.24
b_High	a_Root	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	b_Shallow	2.57	7.73	0.36	0.95	1.80	6.32	0.27	1.29
	c_Deep	2.18	5.35	0.36	0.68	5.44	13.23	1.21	3.41
	d_Deepest	1.95	5.34	0.32	0.64	2.77	8.24	0.58	2.08
c_Low	a_Root	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	b_Shallow	3.84	9.10	0.68	1.53	2.49	5.74	0.44	0.90
	c_Deep	1.66	5.00	0.25	0.69	2.39	6.66	0.42	1.63
	d_Deepest	3.24	10.35	0.47	1.19	3.35	8.32	0.65	2.00
d_Lowest	a_Root	0.75	1.82	0.23	0.60	0.52	2.16	0.07	0.32
	b_Shallow	0.20	1.86	0.03	0.21	0.21	1.53	0.03	0.21
	c_Deep	0.28	1.86	0.05	0.23	0.32	1.95	0.04	0.24
	d_Deepest	0.94	5.37	0.11	0.43	0.35	1.75	0.06	0.30
Overall		2.40	7.10	0.38	0.92	3.82	10.52	0.78	2.63

Fig. 11. Distribution of errors and error correction effort across routine coupling/strength ratios and routine tree location.

bindings analysis to characterize error-prone system structure; 2) to investigate the software engineering principles of coupling and strength and their relationship to software errors and error correction effort; and 3) to collect and analyze data from an ongoing software project without negatively impacting the software developers. This study highlights and empirically supports several software engineering principles. Some of them are widely recognized and some are not. The interpretations span several areas: development methodology, inspection methodology, data collection and analysis, size, coupling/strength, and system structure.

A. Development Methodology

It is better to leave it out than do it incorrectly, and it is better to do only what is necessary. In other words, it is less costly to leave out part of the design or code than to include incorrect design or code. It is cost effective to eliminate extraneous design and unexecutable code.

- Errors of omission ("missing") are 74% [= 8.1/10.9 from Appendix C, part 3)] of the cost of errors of commission ("wrong"). Moreover, when you consider that fixing errors of omission is actually postponed development cost, errors of omission are actually 39% [= 4.2/10.9 from Appendix C, part 3)] the cost of errors of commission.
- It is worthwhile finding "extra" design or code during inspections, especially design inspections, since the associated life cycle costs, e.g., development and testing, are eliminated. Seven percent [= 49/716 from Appendix B, part 5)] of the errors found during inspections were "extras."

B. Inspection Methodology

Design and engineering inspections are cost effective vehicles for error detection, and design inspections are more effective than engineering inspections.

- It is less expensive to find errors during inspections than via trouble reports (TR's) since it is 1.7 [= 6.3/3.7 from Appendix C, part 3)] times more expensive to isolate errors than to fix them.
- If you are not using inspections, you are better off starting with design inspections since they were 3.0 [= 8.3/2.8 from Appendix B, part 3)] times more effective in terms of errors found per inspection than engineering inspections. Design inspections were 3.6 [= 5.1/1.4 from Appendix B, part 3)] times more effective in finding major severity errors.

C. Data Collection and Analysis

Data can be gathered and analyzed during an ongoing software project without hindering the developers.

- Software project personnel should be motivated by the purpose of data collection and briefed on the results of data analysis.
- Data collection forms should be simple, few in number, and generalized with clearly indicated required and optional sections. Data should be collected at the appropriate granularity for analysis and kept on-line.
- An in-house data collection coordinator helps catalyze the data collection process and validate data.

D. Size

Subsystem size seems to be at least as important, if not more important, than routine size. Hence, maybe the software community has been worrying about the wrong issue.

- Smaller subsystems had routines with 3.3 (= 2.82/0.86 from Fig. 4) times fewer errors per KNCSS than did larger subsystems.
- Smaller routines had a slightly higher average (2.69 versus 2.18 from Fig. 8) of errors per KNCSS than did larger routines, but their error-proneness was statistically greater only when a parametric statistical test was used (but not otherwise). Overall, errors in smaller routines were 2.4 (= 5.08/2.15 from Fig. 8) times less expensive to fix.

E. Coupling/Strength

Low coupling and high strength are desirable.

- Routines with the lowest coupling/strength ratios had 7.0 (= 4.14/0.59 from Fig. 7) times fewer errors per KNCSS than routines with the highest coupling/strength ratios and errors were 21.7 (= 9.15/0.42 from Fig. 7) times less costly to fix.
- Subsystems with low coupling/strength ratios had routines with 4.8 (= 2.89/0.60 from Fig. 3) times fewer errors per KNCSS than did subsystems with high coupling/strength ratios.

F. System Structure Hierarchy: Data Bindings View

The structure of the system at the highest level, i.e., initial stages of problem decomposition, and lowest level, e.g., formulation of abstract data types, appear to be better understood than the intermediate levels of abstraction and specification.

- The errors were 47% (= $1.0 - (2.21 + 2.84)/(5.64 + 3.88)$ from Fig. 10) less costly to fix in routines at the shallowest and deepest levels of the data bindings view of the system structure hierarchy than at the middle levels, and there were 15% (= $1.0 - (1.72 + 2.53)/(3.30 + 1.68)$ from Fig. 10) fewer errors per KNCSS.

Further analysis and interpretation of the data are underway. The authors are interested in replicating the results on other projects.

Appendix A
Data Collection Forms and Terminology

Fig. 12 outlines the data collections forms used in the project. The definitions used in the paper for several error-related concepts are as follows.

Development phase	Data collection form
High-level design	Design inspection
Low-level design	Design inspection
Coding and unit test	Error summary worksheet (ESW)
After unit test completion	Engineering inspection
Integration test	Error summary worksheet (ESW)
System test	System trouble report (STR)
Field test and operation	Trouble report (TR)

Fig. 12. Data collection forms used in the development phases.

1) Error-related effort: Error isolation effort—How long it takes to understand where the problem is and what must be changed. *Error fix effort*—How long it takes to implement a correction for the error. *Error correction effort*—How long it takes to correct an error, which is the sum of error isolation effort and error fix effort.

2) Error type: Wrong—Implementation requires a change. The existing code or logic needs to be revised; the functionality is present but it is not working properly. *Extra*—Implementation requires a deletion. The error is caused by existing logic that should not be present. *Missing*—Implementation requires an addition. The error is caused by missing logic or function.

3) Error severity (trouble reports): Level 1—Program is unusable; it requires immediate attention (bypass, patch, or replacement). *Level 2:*—Program is usable, but functionality is severely restricted and there is no work-around; prompt action is required. *Level 3:*—Program is usable, but has functionality limitation that is not critical; it can be avoided, bypassed, or patched. *Level 4:*—Problem is minor, e.g., message or documentation error, and is easily avoided, bypassed, or patched.

4) Error severity (inspections): Major—Error could lead to a problem reported in the field on a trouble report. *Minor*—Anything that is less than "major" severity, e.g., minor reorganizations, some typographical mistakes and misspellings.

5) Error reporter type (trouble reports): User—Error is reported by field user or found by a developer while using the product. *Developer*—Error is discovered by a developer during field testing or when looking at the source code or searching for errors in a released system.

6) Inspection type: Design inspections—These are inspections held during the high-level and low-level design phases. *Engineering inspections*—These are code inspections that are held after the completion of unit testing.

Appendix B
Inspection Error Data

1) Distribution of inspection type:

	Inspection type		
	Design	Engineering	All
Number of inspections	54	116	170

2) Distribution of errors (inspections) by severity and inspection type:

Severity	Inspection type		
	Design	Engineering	All
Major	275	158	433
Minor	175	162	337
All	450	320	770

3) Distribution of average error detection rates (errors per inspection) by severity and inspection type:

Severity	Inspection type		
	Design	Engineering	All
Major	5.1	1.4	2.5
Minor	3.2	1.4	2.0
All	8.3	2.8	4.5

4) Distribution of errors (inspections) by error class and inspection type:

Error class	Inspection type		
	Design	Engineering	All
Wrong	269	228	497
Missing	93	68	161
Extra	25	24	49
All	387	320	707

5) Distribution of errors (inspections) by severity and error class:

Severity	Error class			
	Wrong	Missing	Extra	All
Major	246	111	18	375
Minor	255	55	31	341
All	501	166	49	716

APPENDIX C
TROUBLE REPORT ERROR DATA

1) Distribution of errors (TR's) by reporter type and severity:

Reporter type	Severity				
	1	2	3	4	All
User	2	10	12	2	26
Developer	0	7	10	11	28
Total	2	17	22	13	54

2) Distribution of errors (TR's) by reporter type and error class:

Reporter type	Error class			
	Wrong	Missing	Extra	All
User	14	7	0	21
Developer	18	6	1	25
Total	32	13	1	46

3) Distribution of average TR error correction effort (isolation effort plus fix effort) in hours by error class. The "extra" class is omitted from the table because of the small sample size (there was only one such error):

Average effort	Error class		
	Wrong	Missing	All
Isolation effort	7.2	4.2	6.3
Fix effort	3.7	3.9	3.7
Total correction effort	10.9	8.1	10.0

4) Distribution of errors (TR's) by severity and error class.

Severity	Error class			
	Wrong	Missing	Extra	All
1	2	0	0	2
2	12	3	1	16
3	11	8	0	19
4	8	3	0	11
All	33	14	1	48

APPENDIX D
DATA COLLECTION RECOMMENDATIONS AND LESSONS LEARNED

Several retrospective observations help assess the effectiveness of the data collection process. 1) The number of data collections forms submitted by the project personnel was reasonable, based on experience with other projects. 2) Sixty-one (9%) of the 665 data collection forms submitted had some form of incomplete data. 3) Interviews with project personnel confirmed that the errors that occurred were getting reported on data collection forms. 4) The project personnel seemed to experience a learning effect that over time would continue to increase data accuracy and to decrease collection cost.

Several recommendations and lessons learned resulted from the application of the data collection and analysis methodology in the production environment. Project personnel were interviewed during and after the development process in order to document their reactions to the data collection methodology. The interviews, along with other informal discussions, allowed the authors to make clarifications and suggestions regarding the collection methods. Included here are comments from interviews of the development personnel and observations from the authors.

Benefits: The software project manager identified several benefits from the data collection process and coupling/strength analysis: 1) the emphasis on a data collection process; 2) the empirical results from the final analysis; 3) the intermediate presentation of results prior to project completion; 4) the improvement in the development project as a result of the data collection process; and 5) the identification of valuable metrics and analysis methods.

The project manager also made the following observations. 6) A data collection process needs to be a "grass roots effort," including an in-house coordinator to catalyze the process and a sincere interest on the part of the development personnel in the accuracy of the data. 7) The quantification of information greatly facilitates the planning and scheduling of future activities, phases, and projects. 8) Developers need to be able to assess quality without unnecessarily impacting the project.

Project personnel responsible for data collection coordination made these observations. 1) The data collection "benefits everybody." 2) You have better control over what will or may impact your current system and also what is "waiting in the wings" for future releases. 3) The data collection can be used "to assess the defect removal and show quality certification." 4) Without the data collection, the managers have nothing tangible for assessing project quality.

General Recommendations: The application of the methodology resulted in the following set of general recommendations for data collection and analysis. 1) Having software project members motivated by the purposes of the data collection process is a key component of its success. 2) The data collected on paper forms should be put on-line for access and analysis purposes. 3) Fewer, more general data collection forms with clearly indicated required and optional sections advance the simplicity of the collection and help reduce the paper flow. 4) The appropriate granularity of the data collection (e.g., what data to collect at the level of routines, subsystems, or projects) is driven by the goals of the study and the intended analysis methods. 5) The people participating in the data collection effort should be briefed on the results of the work.

ACKNOWLEDGMENT

The authors are very grateful to several persons on the selected software project for their assistance and support in this research.

Their names cannot be mentioned because of a nondisclosure agreement. The authors appreciate the assistance of D. Hutchens in developing the data bindings analysis tools and S. Wilkin in collecting the data.

REFERENCES

[1] V. R. Basili, "Quantitative evaluation of software engineering methodology," in *Proc. First Pan Pacific Computer Conf.*, Melbourne, Australia, Sept. 10–13, 1985; also available as Tech. Rep. TR-1519, Dep. Comput. Sci., Univ. Maryland, College Park, July 1985.

[2] L. A. Belady and C. J. Evangelisti, "System partitioning and its measure," *J. Syst. Software*, vol. 2, no. 1, pp. 23–29, Feb. 1982.

[3] V. R. Basili and E. E. Katz, "Metrics of interest in an ada development," in *Proc. IEEE Workshop Software Engineering Technology Transfer*, Miami, FL, Apr. 1983, pp. 22–29.

[4] L. A. Belady and M. M. Lehman, "A model of large program development," *IBM Syst. J.*, vol. 3, pp. 225–252, 1976.

[5] B. W. Boehm, *Software Engineering Economics.* Englewood Cliffs, NJ: Prentice-Hall, 1981.

[6] V. R. Basili and B. T. Perricone, "Software errors and complexity: An empirical investigation," *Commun. ACM*, vol. 27, no. 1, p. 42–52, Jan. 1984.

[7] V. R. Basili and R. W. Selby, "Data collection and analysis in software research and management," in *Proc. Amer. Statist. Assoc. and Biometric Soc. Joint Statistical Meetings*, Philadelphia, PA, Aug. 13–16, 1984.

[8] V. R. Basili, R. W. Selby, and T. Y. Phillips, "Metric analysis and data validation across Fortran projects," *IEEE Trans. Software Eng.*, vol. SE-9, no. 6, pp. 652–663, Nov. 1983.

[9] V. R. Basili and A. J. Turner, "Iterative enhancement: A practical technique for software development," *IEEE Trans. Software Eng.*, vol. SE-1, no. 4, Dec. 1975.

[10] V. R. Basili and D. M. Weiss, "A methodology for collecting valid software engineering data," *IEEE Trans. Software Eng.*, vol. SE-10, no. 6, pp. 728–738, Nov. 1984.

[11] T. Emerson, "A discriminant metric for module cohesion," in *Proc. Seventh Int. Conf. Software Eng.*, Orlando, FL, 1984, pp. 294–303.

[12] M. E. Fagan, "Design and code inspections to reduce errors in program development," *IBM Syst. J.*, vol. 15, no. 3, pp. 182–211, 1976.

[13] ——, "Advances in software inspections," *IEEE Trans. Software Eng.*, vol. SE-12, no. 5, pp. 744–751, July 1986.

[14] D. H. Hutchens and V. R. Basili, "System structure analysis: Clustering with data bindings," *IEEE Trans. Software Eng.*, vol. SE-11, no. 8, Aug. 1985.

[15] S. Henry and D. Kafura, "Software quality metrics based on interconnectivity," *J. Syst. Software*, vol. 2, no. 2, pp. 121–131, 1981.

[16] *Statistical Analysis System (SAS) User's Guide.* SAS Inst., Cary, NC, Tech. Rep., 1982.

[17] N. Jardine and R. Sibson, *Mathematical Taxonomy.* New York: Wiley, 1971.

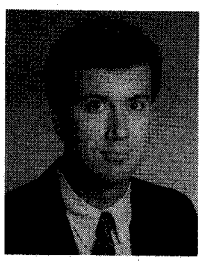
[18] B. W. Kernighan and D. M. Ritchie, *The C Programming Language.* Englewood Cliffs, NJ: Prentice-Hall, 1978.

[19] H. Scheffe, *The Analysis of Variance.* New York: Wiley, 1959.

[20] R. W. Selby, "Evaluations of software technologies: Testing, cleanroom, and metrics," Ph.D. dissertation, Dep. Comput. Sci., Univ. Maryland, College Park, Tech. Rep. TR-1500, 1985.

[21] W. P. Stevens, G. J. Myers, and L. L. Constantine, "Structural design," *IBM Syst. J.*, vol. 13, no. 2, pp. 115–139, 1974.

[22] V. Y. Shen, T. J. Yu, S. M. Thebaut, and L. R. Paulsen, "Identifying error-prone software—An empirical study," *IEEE Trans. Software Eng.*, vol. SE-11, no. 4, pp. 317–324, Apr. 1985.



Richard W. Selby (S'83–M'85) received the B.A. degree in mathematics and computer science from Saint Olaf College, Northfield, MN, in 1981 and the M.S. and Ph.D. degrees in computer science from the University of Maryland, College Park, in 1983 and 1985, respectively.

He is an Assistant Professor of Information and Computer Science at the University of California, Irvine. His research interests include methodologies for developing and testing software, techniques for empirically evaluating software methodologies, and software environments.

Dr. Selby is a member of the Association for Computing Machinery and the IEEE Computer Society.

Victor R. Basili (M'83–SM'84–F'90) is a Professor in the Institute for Advanced Computer Studies and the Department of Computer Science at the University of Maryland, College Park. He was involved in the design and development of several software projects, including the SIMPL family of programming languages. He is currently measuring and evaluating software development in industrial and government settings and has consulted with many agencies and organizations, including IBM, GE, CSC, GTE, MCC, AT&T, Motorola, HP, Boeing, NRL, NSWC, and NASA. He is one of the founders and principals in the Software Engineering Laboratory, a joint venture between NASA Goddard Space Flight Center, the University of Maryland, and Computer Sciences Corporation, established in 1976. He has been working on the development of quantitative approaches for software management, engineering and quality assurance by developing models and metrics for the software development process and product. He has authored over 90 papers. In 1982, he received the Outstanding Paper Award from the IEEE TRANSACTIONS ON SOFTWARE ENGINEERING for his paper on the evaluation of methodologies.

Dr. Basili is currently the Editor-in-Chief of the IEEE TRANSACTIONS ON SOFTWARE ENGINEERING and was Program Chairman for several conferences including the 6th International Conference on Software Engineering. He has served on the editorial board of the *Journal of Systems and Software.* He is a member of the Board of Governors of the IEEE Computer Society.