UMIACS–TR–88–46
CS–TR–2052

June, 1988

Analyzing Error–Prone System Coupling and Cohesion†

Richard W. Selby
Department of Information and Computer Science
University of California
Irvine, CA 92717

Victor R. Basili
Institute for Advanced Computer Studies
Computer Science Department
University of Maryland
College Park, MD 20742

ABSTRACT

One central feature of the structure of a software system is the coupling among its components (e.g., subsystems, modules) and the cohesion within them. The purpose of this study is to quantify ratios of coupling and cohesion and use them to identify error–prone system structure. We use measure of data interaction, called data bindings, as the basis for calculating software coupling and cohesion. We selected a 140,000 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 cluster analysis to determine a hierarchical description of each of the system's 77 subsystems. An analysis of variance model is ussed to characterize subsystem and individual routines that had either many/few errors or high/low error correction effort.

† This work was supported in part by the IBM Shared University Research (S.U.R.) Program: Department of Computer Science, University of Maryland; and University of Maryland Institute for Advanced Computer Studies.

Contents

List of Figures

1 Introduction

Several researchers have proposed methods for relating the structure of a software system to its quality (e.g., [BE82] [HK81] [Eme84]). One pivotal step in assessing the structure of a software system is characterizing its coupling and cohesion. Intuitively, the *cohesion* 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. Cohesion may sometimes be referred to as "strength." Various interpretations for coupling and cohesion have been proposed [SMC74]. In this paper, we present an empirical study that evaluates the effectiveness of cohesion and coupling in identifying error-prone system structure. Our measurement of cohesion and coupling is based on intra-system interaction in terms of *software data bindings* [BT75] [HB85]. 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.

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.

The paper is organized into several sections. Section 2 discusses the software project selected. Section 3 describes the data definition, collection, and analysis methodology used. The software error data collected is characterized in Section 4. The data bindings software analysis and supporting tools are described in Section 5. The data analysis appears in Section 6. Section 7 presents the interpretations and conclusions.

2 Selected Software Project

The software project selected for study is the next release of an internal software library tool. The previous system release contains approximately 100,000 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 140,000 source lines. 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 lines in the system and the vast majority of the system logic and intra-system interactions. Project duration, including system and field test, spanned approximately 16 months and maximum staffing included 23 persons.

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 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 in size, including comments.

3 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. The fourth subsection summarizes the effectiveness of the data collection process in gathering data during the software project. The fifth subsection 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.

3.1 Data Collection and Analysis Methodology

The data collection and analysis methodology employs the goal-question-metric paradigm [BW84] to result in a set of software product and process metrics, a

"metric vector" [BK83], sensitive to the cost and quality goals for the particular environment. There are several steps in the data collection and analysis methodology spanning software metric definition, collection, analysis, and interpretation. The data collection and analysis methodology consists of seven steps [BW84] [BS84] [Sel85] [Bas85]:

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, referred to as the goal-question-metric paradigm, 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.

3.2 Metric Vector

The set of metrics defined was described in terms of a "metric vector," consisting of seven dimensions: { effort, non-error changes, errors, size, data use, execution, environment } [BK83]. These seven dimensions are defined as follows: (1) effort – the time expended in producing the software product; (2) 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:

- data collection forms, which are discussed in the next section, and
- automated data bindings analysis, which is discussed in Section 5.

3.3 Data Collection Forms

The metrics defined were categorized according to their natural collection source. The approach to data collection spanned two steps.

1. 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.

2. We worked within existing, established procedures using existing forms, as far as possible, to collect the new data.

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).

The collection of the data was conducted so as to affect minimal interference on the project personnel.

Inspections Two kinds of formal inspections are held during development: design inspections and engineering inspections [Fag76] [Fag86]. 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.

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).

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

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. Since the fixes are sometimes implemented in the current development release, they reflect change activity.

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.

3.4 Effectiveness of Data Collection Process

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. Project personnel were interviewed in order to document their reactions to the data collection methodology.
4. The interviews with project personnel confirmed that the errors that occurred were getting reported on data collection forms.
5. Clarifications and suggestions were made to project personnel during the development process regarding the data collection effort.
6. The project personnel seemed to experience a learning effect that over time would continue to increase data accuracy and to decrease collection cost.

3.5 Recommendations and Lessons Learned

Several recommendations and lessons learned resulted from the application of the data collection and analysis methodology in the production environment. They include comments from interviews of the development personnel and observations from the authors.

Benefits Project personnel responsible for data collection coordination felt that:

1. The data collection "benefits everybody."

2. You have better control over what will/may effect 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.

The software project manger 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 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.

He also felt that:

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.

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/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.

4 Characterization of Software Error Data

The following subsections characterize the software error data collected from the trouble reports (TR's) and software inspections. First, however, some terminology is clarified.

4.1 Terminology

Figure 1 summarizes the data collections forms used in the various development phases. See Section 3.3 for an explanation of each of the data collection forms. The forms record a variety of different types of software error data.

Standardization efforts (e.g., [IEE83] [Glo79]) have attempted to distinguish among the terms *error*, *fault*, and *failure*. The proposed definitions are intended to differentiate among the three entities by the following explanation. A "fault" is a specific manifestation in a software document (e.g., design document, source code) of a programmer "error." Due to a misconception or document discrepancy, a programmer commits an "error" that can result in several "faults" in the program. When input data exercises a "fault" in an executable software document (e.g., source code), a "failure" may be observed by a user in the output from the program. Theoretically, there can be a many-to-many mapping from "errors" to "faults" and a many-to-many mapping from "faults" to "failures."

Figure 1 indicates that design and engineering inspections report *faults*, while error summary worksheets, system trouble reports, and trouble reports record

Figure 1: Data collection forms used in the development phases.

Development phase	Data collection form	Data recorded	
		Faults	Failures
High-level design	Design inspection	X	
Low-level design	Design inspection	X	
Coding and unit test	Error summary worksheet (ESW)	X[1]	X
After unit test completion	Engineering inspection	X	
Integration test	Error summary worksheet (ESW)		X
System test	System trouble report (STR)		X
Field test and operation	Trouble report (TR)		X

failures. Throughout this paper, we use the term *error* to refer to a mistake made by a software developer that resulted in either a *fault* or a *failure*. The definitions used in the paper for several error-related concepts are as follows.

- 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.
- 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.

- Error severity (trouble reports):
  - 1 — Program is unusable; it requires immediate attention (bypass, patch, or replacement).
  - 2 — Program is usable, but functionality is severely restricted; prompt action is required.
  - 3 — Program is usable, but has functionality limitation that is not critical; it can usually be avoided, bypassed, or patched.
  - 4 — Problem is minor, e.g., message or documentation error, and is easily avoided, bypassed, or patched.
- 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.
- 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.
- 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.

4.2 Software Errors from Inspections

Figure 2: Distribution of inspection type.

	Inspection type		
	Design	Engineering	All
#inspections	54	116	170

Figure 3: 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

Summary of Results

1. There were 770 errors reported in 170 inspections [see Figure 2 and 3].
2. Inspectors reported more (56%) major (high) severity errors than they did minor (low) severity errors [see Figure 3].
3. Design inspections resulted in the detection of 58% of all inspection-detected errors [see Figure 3].
4. A majority (63%) of the major severity inspection-errors were detected during design inspections [see Figure 3].
5. Four and one half (4.5) errors were detected per inspection on the average [see Figure 4].
6. The number of major severity errors detected per inspection was three and one half times greater in a design inspection than in an engineering inspection [see Figure 4].
7. The overall design inspection error detection rate was three times greater than the overall engineering inspection error detection rate [see Figure 4].

Figure 4: Distribution of average error detection rates (#errors/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

Figure 5: 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

8. Both design and engineering inspections have a profile across error class that is proportional to the overall profile [see Figure 5].

9. Twice as many of the "missing" errors were of major severity than were of minor severity. One and one half times as many of the "extra" errors were of minor severity than were of major severity. The "wrong" errors included a roughly equal number of major and minor severity errors. [See Figure 6.]

4.3 Software Errors from Trouble Reports

Summary of Results

1. There were a total of 54 valid trouble reports (TR's) [see Figure 7].

Figure 6: 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

Figure 7: 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. TR-errors reported by users tended to be of higher severity than those reported by system developers [see Figure 7]. Note that system users may perceive errors to be of higher relative severity than do developers.

3. The majority (70%) of the TR-errors were a design or code segment being "wrong", as opposed to being "missing" or "extra" [see Figure 8].

4. Isolating a TR-error required almost twice 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 [see Figure 9]. Correspondingly, the isolation of a "wrong" TR-error required the most effort — almost twice as much effort as isolating a missing TR-error. Note that the isolation costs would have been zero if the errors reported on TR's had been found during inspections.

5. The effort for fixing a "missing" error could be counted as development effort, as opposed to error correction effort. With this interpretation, the error correction effort for "missing" errors is only the isolation effort. In this view, the error correction effort (isolation plus fix) for "wrong" errors is 2.8 times the correction effort (isolation only) for "missing" errors [see Figure

Figure 8: 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

Figure 9: TR error correction effort (isolation effort plus fix effort) in hours by error class.

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

9]. Hence, it was less costly overall to leave out a design or code segment, rather than to include an incorrect one.

6. The severity distribution of the TR-errors appears to be reasonably proportional across the "wrong" and "missing" error classes [see Figure 10].

7. A majority (70%) of the inspection-reported errors were in the "wrong" class, which is reasonably consistent with the percentage (70%) of "wrong" TR-reported errors [see Figures 6 and 8].

8. In inspections there were proportionately more "extra" errors and fewer "missing" errors than there were from TR's [see Figures 6 and 8]. Seven percent of the inspection-reported errors were "extra," as opposed to two percent of TR-reported errors [see Figures 6 and 8]. Finding "extra" errors is certainly worthwhile since it helps reduce development costs; e.g., developers do not have to test the "extra" portion. It is desirable to detect the "extra" errors as early in development as possible, such as in design inspections [see Figure 5].

Figure 10: 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

5 Data Bindings Analysis

5.1 Clustering with Data Bindings

One primary goal for this study was to investigate the relationship of "software data bindings" to software errors [HB85]. "Data bindings" are measures that capture the data interaction across portions of a software system. The theoretical background for the measures are described in [HB85]. Earlier studies have revealed insights about the usefulness of data bindings in the characterization of software systems and their errors [BT75] [HB85]. In order to describe the data bindings analysis process applied, we first introduce some terminology (see also [HB85]).

Potential Data Binding A potential 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. Potential data bindings reflect the possibility of a data interaction between two components, based upon the locations of p, q, and x. That is, there is a possibility that p and q can communicate via the variable x without changing or moving the definition of x. Whether x is actually mentioned inside of p or q is irrelevant in the computation of potential data bindings.

Used Data Binding A used data binding is a potential data binding where p and q use x for either reference or assignment. The used data binding requires more work to calculate than the potential data binding as it is necessary to look inside the components p and q. It reflects a similarity between p and q (they both use the variable x).

Actual Data Binding An actual data binding is defined as a used data binding where p assigns a value to x and q references x. The actual data binding is slightly more difficult to calculate as a distinction between reference and assignment must be maintained. Thus more memory is required but there is little difference in computation time. The actual data binding only counts those used data bindings 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.

There are stronger levels of data bindings. However, in this study we calculated *actual data bindings*. This level of data bindings seems to offer 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 prove otherwise.

First, we calculated the actual data bindings in the system. Then, we applied the statistical technique of clustering [Eve80] to the data bindings information to produce a hierarchical description for the software system. 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 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. The technique of clustering has been applied previously to partition a large system into subsystems in [BE82]. Hierarchical clusters have been formally defined in [JS71].

5.2 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 2, 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 [KR78].

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 will 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 [Hut87]. 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_bin and creates a dissimilarity matrix (a non-negative, real, symmetric matrix with zeros on the diagonal [HB85] [JS71]) 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.

6 Data Analysis

The data collection and analysis methodology was successful in producing a wide range of statistically significant results. Several analysis techniques, including analysis of variance and cluster analysis, were employed in the study.

6.1 Terminology

Throughout the analysis and interpretation, we use the terms *subsystems* and *routines* as follows:

- Routine — A routine is a main program, procedure, or function. There are a total of 451 source code routines in the system.

- Subsystem — A subsystem is a large set of routines that are linked together to form an executable system feature. There are 77 executable features in the system. They average 19,000 source lines in size.

A routine is linked into 12.4 subsystems on the average. Therefore, the total size of the whole system is not 77 x 19,000 = 1,463,000 source lines; the total size is 140,000 source lines. See Section 2 for further description of the subsystems and routines in the software system.

We used the analysis tools described in Section 5 to produce hierarchical descriptions for each of the 77 subsystems. The hierarchical descriptions are rooted, connected trees that indicate 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.

Associated with each cluster in a subsystem is a number ranging from 0 to 100. This number reflects the nature of the binding of the routines in the cluster. This number is interpreted as the following ratio:

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

That is, the number captures the coupling/strength ratio for a cluster of routines within a subsystem. The coupling/strength ratios range from 0 to 100 since they are calculated on a relative scale. The use of the word "relative" here means relative to the coupling/strength ratios that could result from the range of all possible occurrences of data bindings. In the data bindings analysis process, the clusters are formed in a bottom-up manner. 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.

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 [SMC74].

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 2). We calculated three different measures based on the clusters resulting from the data bindings analysis. For each routine occurrence, we calculated:

- 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 alternate form of system documentation.

For each subsystem, we calculated:

- Subsystem coupling/strength ratio — The median of the coupling/strength ratios for the clusters within the subsystem. We use a non-parametric 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.

6.2 Data Analysis Method

An 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.

6.2.1 Independent Variables

The analysis of variance model considered numerous factors simultaneously [Sch59]. When defining the levels for some of the factors, we used non-parametric 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., 86 for subsystem coupling/strength ratio) 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., [Boe81] [BSP83]). The primary factors and their respective levels in the model were:

1. Subsystem size
    - Small — Subsystems with less than or equal to 12,148 source lines, including comments
    - Large — Subsystems with greater than 12,148 source lines, including comments
2. Subsystem coupling/strength ratio
    - Low — Subsystems with a coupling/strength ratio of less than 86
    - High — Subsystems with a coupling/strength ratio of greater than or equal to 86
3. Individual subsystem's attributes
    - One level for each of the 77 subsystems
4. Routine size
    - Small — Routines with less than or equal to 255 source lines, including comments
    - Large — Routines with greater than 255 source lines, including comments
5. Routine coupling/strength ratio
    - 4_Highest — The uppermost quartile of the coupling/strength ratios for the clusters in a subsystem

- 3_Higher — The next lower quartile of the coupling/strength ratios for the clusters in a subsystem
- 2_Lower — The next lower quartile of the coupling/strength ratios for the clusters in a subsystem
- 1_Lowest — The lowest quartile of the coupling/strength ratios for the clusters in a subsystem

6. Routine location in subsystem's data binding tree

   - 4_Root — The uppermost quartile (nearest the root of the tree) of the clusters in a subsystem's data binding tree
   - 3_Shallower — The next lower quartile of the clusters in a subsystem's data binding tree
   - 2_Deeper — The next lower quartile of the clusters in a subsystem's data binding tree
   - 1_Deepest — The lowest quartile (furthest from the root of the tree) of the clusters in a subsystem's data binding tree

Four two-way interactions were also included in the model:

7. Interaction of subsystem size with subsystem coupling/strength ratio
8. Interaction of routine size with routine coupling/strength ratio
9. Interaction of routine size with routine location in subsystem's data binding tree
10. Interaction of routine coupling/strength ratio with routine location in subsystem's data binding tree

6.2.2 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 KLOC — The total number of inspection, TR, STR, and ESW errors in a routine per 1000 lines of source code

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 KLOC — The total amount of effort (in hours) spent correcting TR and ESW errors in a routine per 1000 lines of source code

In general, the discussion will focus on the errors per KLOC and the error correction effort per KLOC measures of the routines as opposed to the absolute numbers. This factors out possible underlying correlations between source lines and number of errors or amount of error correction effort. The statistics for all four measures are reported, however. The discussion will tend to highlight results that demonstrated a statistically significant difference, as opposed to those where there was no statistical difference.

6.3 Characterization of High-Error and Low-Error Subsystems

In the source code portions of the system (see Section 2), 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. In the subsequent figures, all inspection, ESW, STR, and TR errors are counted equally.

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. The results are summarized in a following section.

6.3.1 Subsystem Coupling/Strength Ratio

Figure 11 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 (i) the number of

Figure 11: Distribution of errors and error correction effort by subsystem coupling/strength ratios.

Subsystem coupling/ strength	Errors				Error correction hours			
	per KLOC		Total		per KLOC		Total	
	Mean	Std	Mean	Std	Mean	Std	Mean	Std
High	1.54	3.95	0.44	0.99	2.80	7.53	0.88	2.69
Low	0.31	1.16	0.15	0.52	0.91	4.51	0.42	2.39
Overall	1.28	3.58	0.38	0.92	2.39	7.03	0.78	2.63

errors per 1000 lines of source code (KLOC), (ii) the number of errors, (iii) the error correction effort per KLOC, and (iv) the error correction effort in the routines. Subsystem coupling/strength ratio was not a statistically significant factor with respect to either errors per KLOC or error correction effort per KLOC ($\alpha > .05$)[2].

6.3.2 Subsystem Size

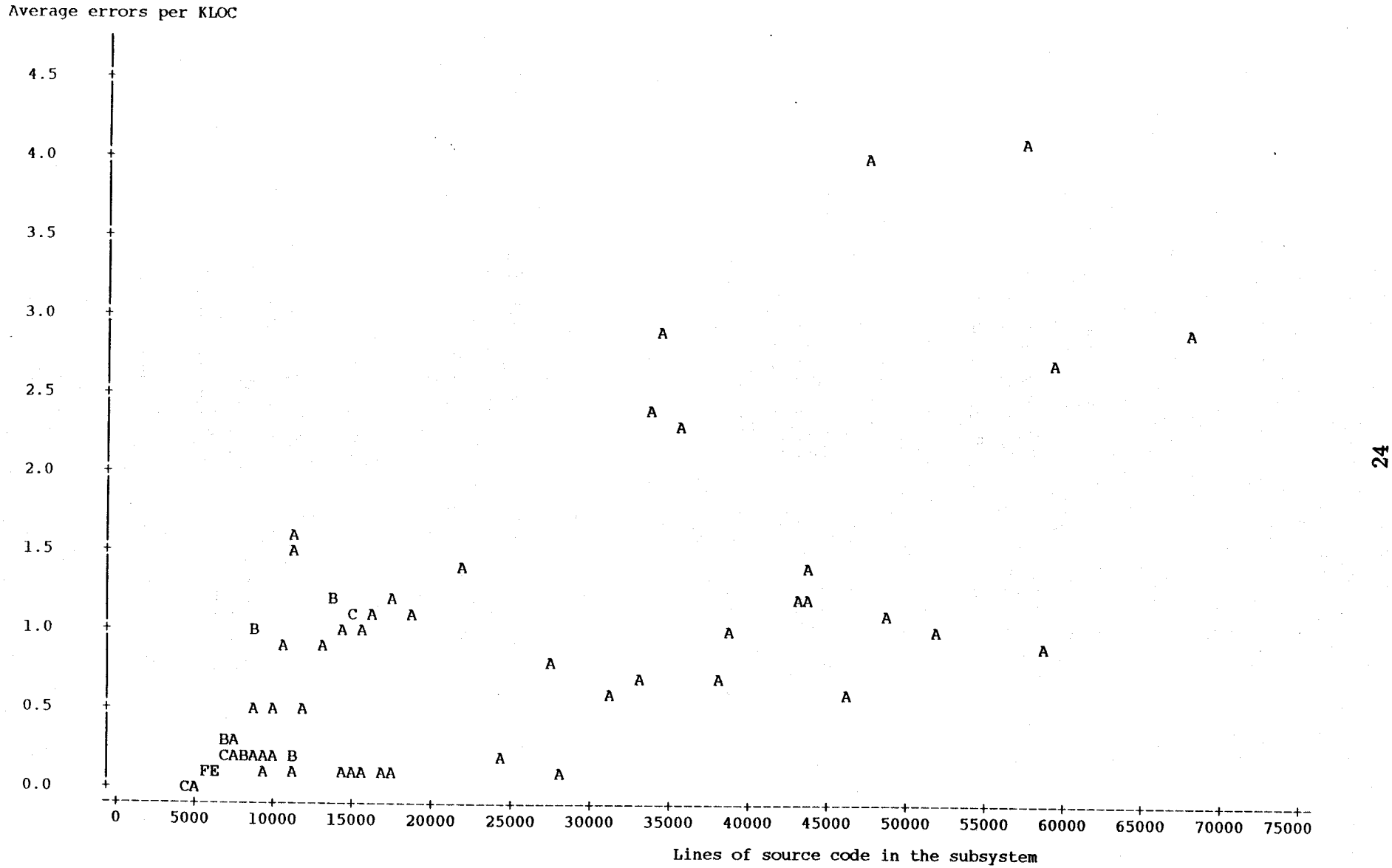
Figure 12: Distribution of errors and error correction effort by subsystem size.

Subsystem size	Errors				Error correction hours			
	per KLOC		Total		per KLOC		Total	
	Mean	Std	Mean	Std	Mean	Std	Mean	Std
Large	1.52	3.94	0.43	0.98	2.77	7.44	0.86	2.61
Small	0.35	1.22	0.17	0.58	0.98	4.96	0.49	2.71
Overall	1.28	3.58	0.38	0.92	2.39	7.03	0.78	2.63

Figure 12 presents the errors and error correction effort in the routines in subsystems with different sizes. The subsystems of large size had routines that averaged 1.52 errors per KLOC, which was greater than the small subsystem average of 0.35 errors per KLOC ($\alpha < .05$). A plot of errors per KLOC versus subsystem size appears in Figure 13.

[2]The F-test significance levels reported in this and later sections are based on the use of Type IV partial sums of squares[Sch59]. Any statistical difference discussed will at least be significant at the $\alpha < .05$ level, unless otherwise noted.

Figure 13: Relationship between errors per KLOC in the routines and subsystem size. Legend: A = 1 observation, B = 2 observations, etc.



6.3.3 Interactions Across Subsystem Coupling/Strength Ratio and Size

Figure 14: Distribution of errors and error correction effort across subsystem coupling/strength ratios and subsystem size.

Subsystem coupling/ strength	Subsystem size	Errors				Error correction hours			
		per KLOC		Total		per KLOC		Total	
		Mean	Std	Mean	Std	Mean	Std	Mean	Std
High	Large	1.66	4.12	0.46	1.02	2.99	7.71	0.92	2.66
	Small	0.45	1.41	0.21	0.66	1.11	5.31	0.56	2.93
Low	Large	0.36	1.27	0.15	0.52	0.91	4.11	0.39	2.07
	Small	0.28	1.09	0.15	0.52	0.90	4.75	0.44	2.57
Overall		1.28	3.58	0.38	0.92	2.39	7.03	0.78	2.63

Figure 14 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 resulted in a statistically significant interaction for errors per KLOC ($\alpha < .011$). Large subsystems with high coupling/strength ratios had routines that averaged 1.66 errors per KLOC, which was substantially more than the other subsystems — their combined average was 0.36 errors per KLOC. In addition, combining subsystem coupling/strength ratio and size resulted in an interaction that was almost statistically significant for error correction effort per KLOC ($\alpha < .066$). Large subsystems with high coupling/strength ratios had routines that averaged 2.99 error correction hours per KLOC — the other subsystems had a combined average of 0.97 error correction hours per KLOC.

6.3.4 Summary of Results

1. Large subsystems with high coupling/strength ratios had routines with the most errors per KLOC.

2. Large subsystems with high coupling/strength ratios had routines with six times as many errors per KLOC than did small subsystems with low coupling/strength ratios.

3. Large subsystems with high coupling/strength ratios had routines with ten times as many unit and integration test (ESW[3]) errors per KLOC than did small subsystems with low coupling/strength ratios.

4. Large subsystems with high coupling/strength ratios had routines with eight times as much error correction effort per KLOC from unit and integration test (ESW) errors than did small subsystems with low coupling/strength ratios.

6.4 Characterization of High-Error and Low-Error Routines

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. The results are summarized in a following section. As mentioned in Section 6.3 there were 299 distinct errors, counting all inspection, ESW, STR, and TR errors equally; 204 of them had data on error correction effort.

6.4.1 Routine Coupling/Strength Ratio

Figure 15 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 (i) the number of errors per 1000 lines of source code (KLOC), (ii) the number of errors, (iii) the error correction effort per KLOC, and (iv) the error correction effort in the routines.

The routine coupling/strength ratio statistically effected both the number of errors per KLOC and the error correction effort per KLOC in the routines ($\alpha < .0008$ and $\alpha < .002$, respectively). The routines in coupling/strength region 4_HIGHEST had the most errors per KLOC (an average of 2.27) and the highest error correction effort per KLOC (an average of 5.86 hours). The routines with coupling/strength ratios in either region 3_HIGHER or 2_LOWER had the second most errors per KLOC and the second most error correction effort per KLOC. The 3_HIGHER and 2_LOWER regions were not statistically different in either

[3]Errors during unit and integration testing were reported on error summary worksheets (ESW's); see Figure 1.

Figure 15: Distribution of errors and error correction effort by routine coupling/strength ratios.

Routine coupling/ strength	Errors				Error correction hours			
	per KLOC		Total		per KLOC		Total	
	Mean	Std	Mean	Std	Mean	Std	Mean	Std
4_Highest	2.27	4.58	0.59	1.04	5.86	10.98	1.94	4.20
3_Higher	1.15	3.13	0.34	0.74	2.19	6.84	0.72	2.54
2_Lower	1.45	4.19	0.44	1.18	1.57	4.27	0.49	1.61
1_Lowest	0.28	1.11	0.15	0.49	0.21	1.09	0.06	0.29
Overall	1.28	3.58	0.38	0.92	2.39	7.03	0.78	2.63

errors per KLOC or error correction effort per KLOC. Those routines in region 1_LOWEST had the fewest errors per KLOC (an average of 0.28) and the least error correction effort per KLOC (an average of 0.21 hours).[4] Plots of errors per KLOC and error correction effort per KLOC versus routine coupling/strength ratio appear in Figures 16 and 17, respectively. These results empirically support the software engineering principle of desiring low coupling and high strength.

6.4.2 Routine Size

Figure 18 presents the errors and error correction effort in the routines with different sizes. The routine size statistically effected the error correction effort per KLOC for the routines ($\alpha < .0001$). Routines of large size had an average of 3.22 hours error correction effort per KLOC, which was more than did those of small size (an average of 1.36 hours error correction effort per KLOC). Although small routines had slightly more errors per KLOC than did large routines, the difference was not statistically significant ($\alpha > .05$). A separate study has indicated, however, that smaller routines may be more error-prone than larger routines [BP84]. A plot of errors per KLOC versus routine size appears in Figure 19.

6.4.3 Routine Location in Data Binding Tree

[4]All multiple comparison results, such as the one in the previous four sentences, were conducted with Tukey's multiple comparison statistic [Sch59] [Ins82]. All of the pairwise statistical comparisons of these four categories are statistically significant at the $\alpha < .05$ level simultaneously.

Figure 16: Relationship between errors per KLOC in the routines and routine coupling/strength ratio. Legend: A = 1 observation, B = 2 observations, etc.

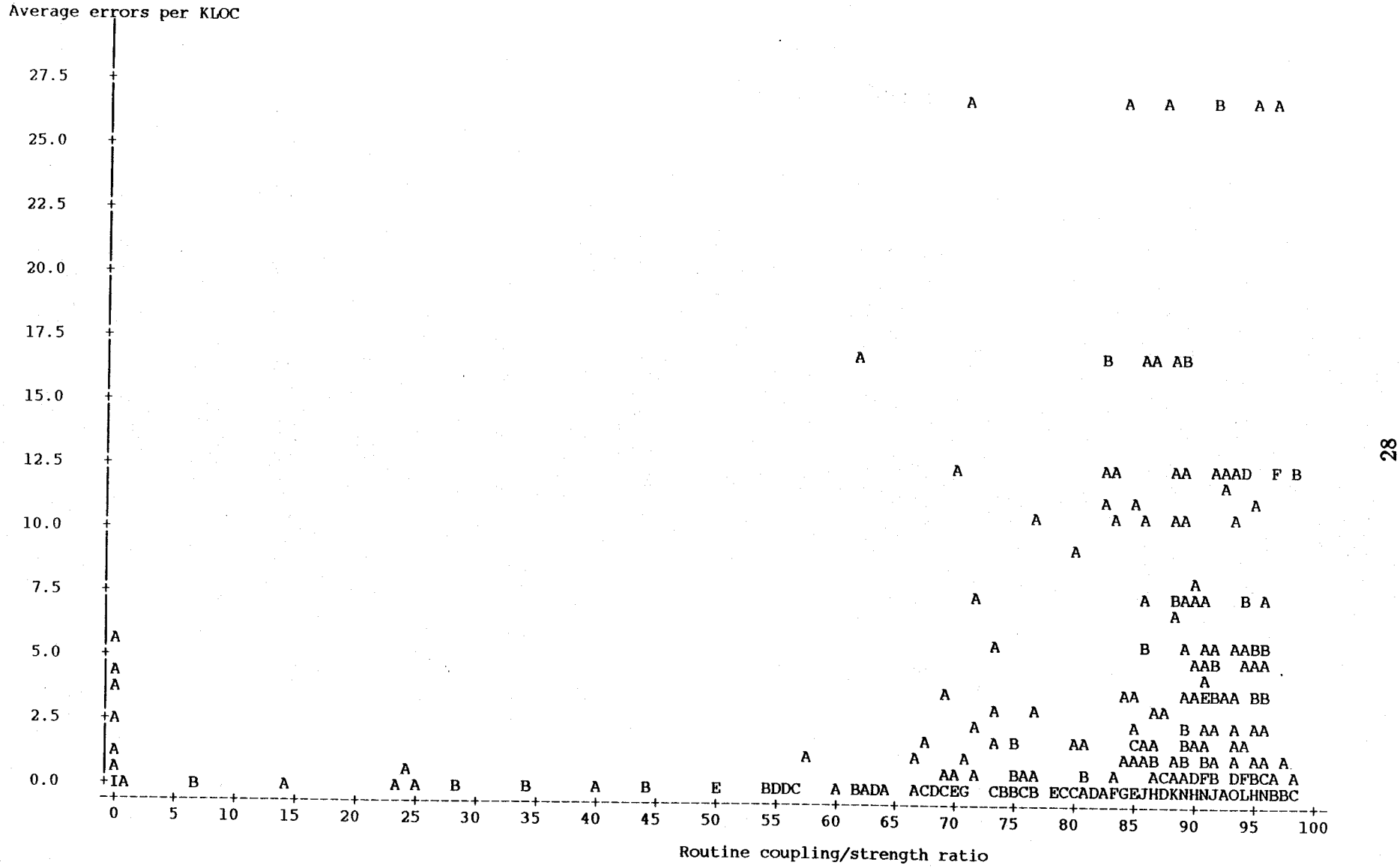


Figure 17: Relationship between error correction effort per KLOC in the routines and routine coupling/strength ratio. Legend: A = 1 observation, B = 2 observations, etc.

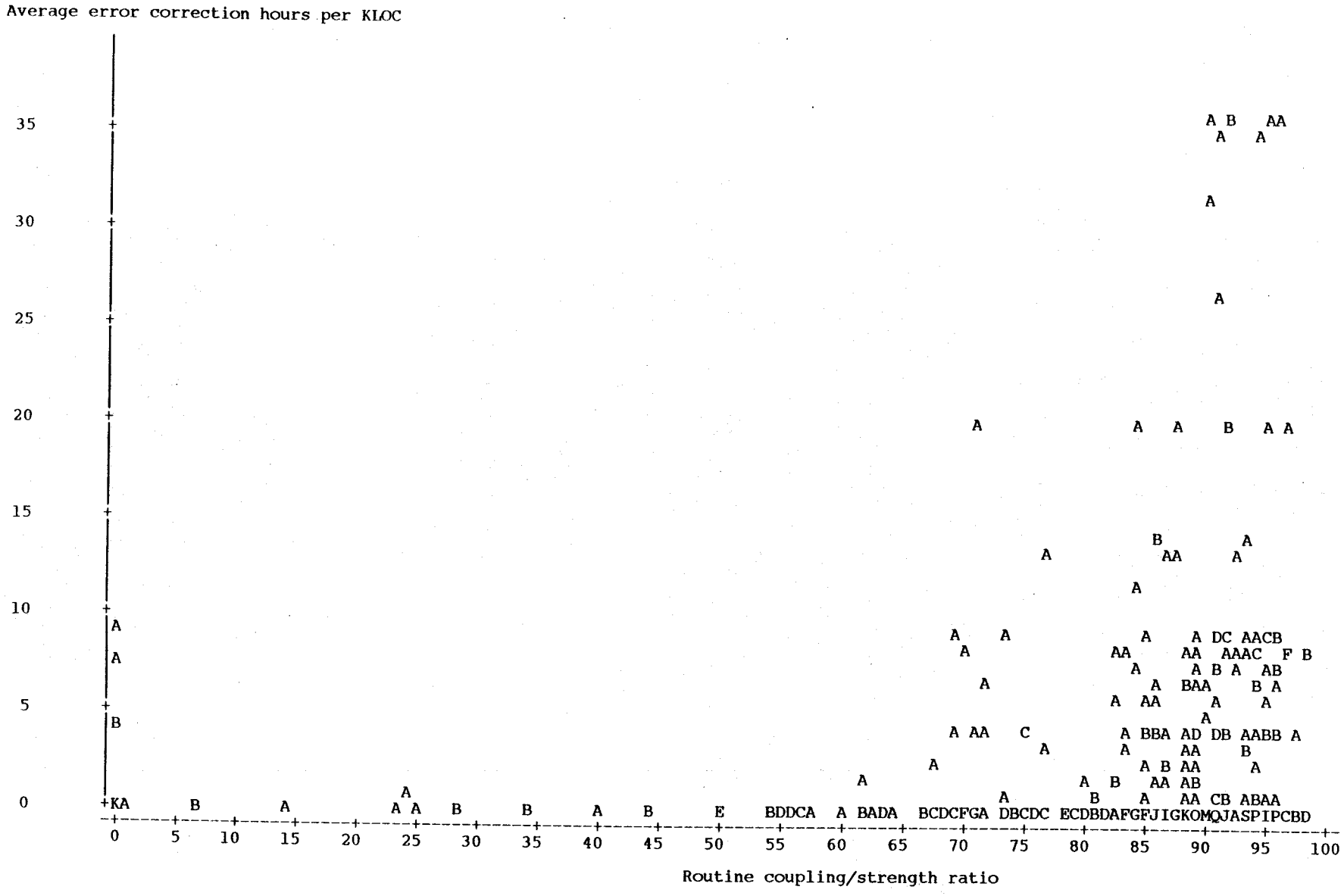


Figure 18: Distribution of errors and error correction effort by routine size.

Routine size	Errors				Error correction hours			
	per KLOC		Total		per KLOC		Total	
	Mean	Std	Mean	Std	Mean	Std	Mean	Std
Large	1.19	2.54	0.47	0.99	3.22	8.72	1.20	3.42
Small	1.39	4.55	0.26	0.80	1.36	3.81	0.26	0.71
Overall	1.28	3.58	0.38	0.92	2.39	7.03	0.78	2.63

Figure 20 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 effected the number of errors per KLOC in the routines ($\alpha < .0001$). Routines in tree location region 3_SHALLOWER had an average of 1.78 errors per KLOC, which was more than any of the other three tree location regions.[5]

The routine location in the data binding tree also statistically effected the error correction effort per KLOC for the routines ($\alpha < .0001$). The routines in tree location region 3_SHALLOWER had the most error correction effort per KLOC (an average of 3.55 hours), those in tree location region 2_DEEPER had the second most, and those in regions 4_ROOT and 1_DEEPEST had the fewest and were not statistically different (they had a combined average of 1.53 hours). A plot of error correction effort per KLOC versus data binding tree location appears in Figure 21. One interpretation for there being less error correction effort per KLOC in regions 4_ROOT and 1_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 Sections 6.3.2 and 6.3.3.

6.4.4 Interactions Across Routine Coupling/Strength Ratio, Size, and Location in Data Binding Tree

Figure 22 presents the errors and error correction effort in the routines with different coupling/strength ratios and different data binding tree locations. There was a significant interaction between the routine coupling/strength ratio and data

[5]Also, note that region 1_DEEPEST had more errors per KLOC than did region 4_ROOT.

Figure 19: Relationship between errors per KLOC in the routines and routine size. Legend: A = 1 observation, B = 2 observations, etc. Note: 34 observations hidden behind Z's.

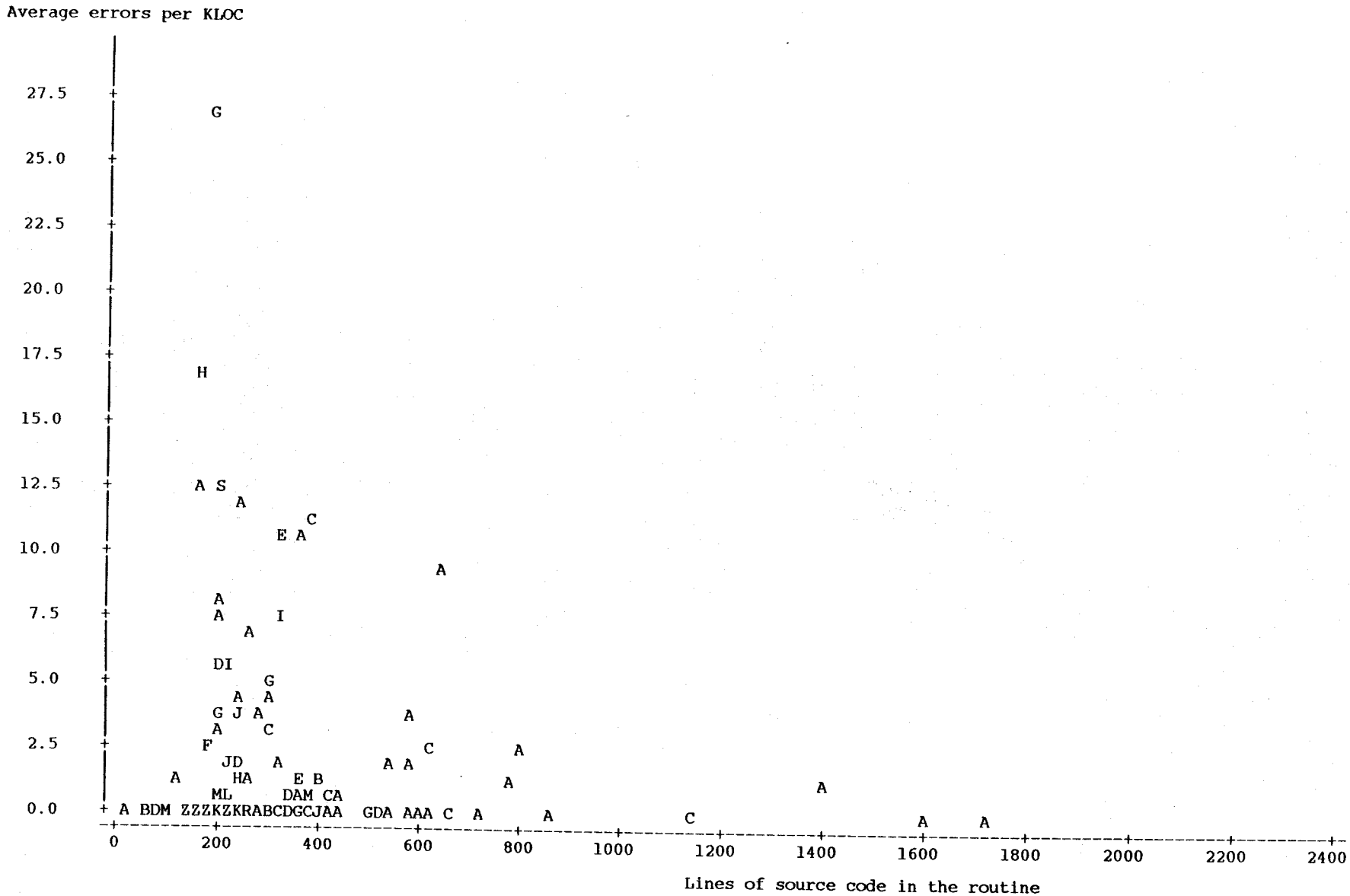


Figure 20: Distribution of errors and error correction effort by routine location in data binding tree.

Routine tree location	Errors				Error correction hours			
	per KLOC		Total		per KLOC		Total	
	Mean	Std	Mean	Std	Mean	Std	Mean	Std
4_Root	0.88	2.82	0.30	0.77	1.30	4.82	0.37	1.59
3_Shallower	1.78	4.44	0.51	1.12	3.55	8.88	1.19	3.36
2_Deeper	0.96	2.48	0.27	0.63	2.51	7.39	0.83	2.82
1_Deepest	1.28	3.73	0.38	0.96	1.76	5.08	0.57	1.95
Overall	1.28	3.58	0.38	0.92	2.39	7.03	0.78	2.63

binding tree location for the number of errors per KLOC in the routines ($\alpha <$ .0001). 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 effected the error correction effort per KLOC for the routines (all at $\alpha <$ .0001). Routines with the highest coupling/strength ratios (4_HIGHEST) and a location in the "central portion" of the data binding tree (3_SHALLOWER or 2_DEEPER) had the most error correction effort per KLOC (a combined average of 6.46 hours).

6.4.5 Summary of Results

1. The routines with the highest coupling/strength ratios had the most errors per KLOC and the most error correction effort per KLOC.

2. The routines with the lowest coupling/strength ratios had the fewest errors per KLOC and the least error correction effort per KLOC.

3. The routines with the highest coupling/strength ratios had over eight times as many errors per KLOC than did routines with the lowest coupling/strength ratios.

4. The routines with the highest coupling/strength ratios had over 27 times as much error correction effort per KLOC than did routines with the lowest coupling/strength ratios.

Figure 21: Relationship between error correction effort per KLOC in the routines and routine location in the data binding tree. Legend: A = 1 observation, B = 2 observations, etc. Note: 5 observations hidden behind Z's.

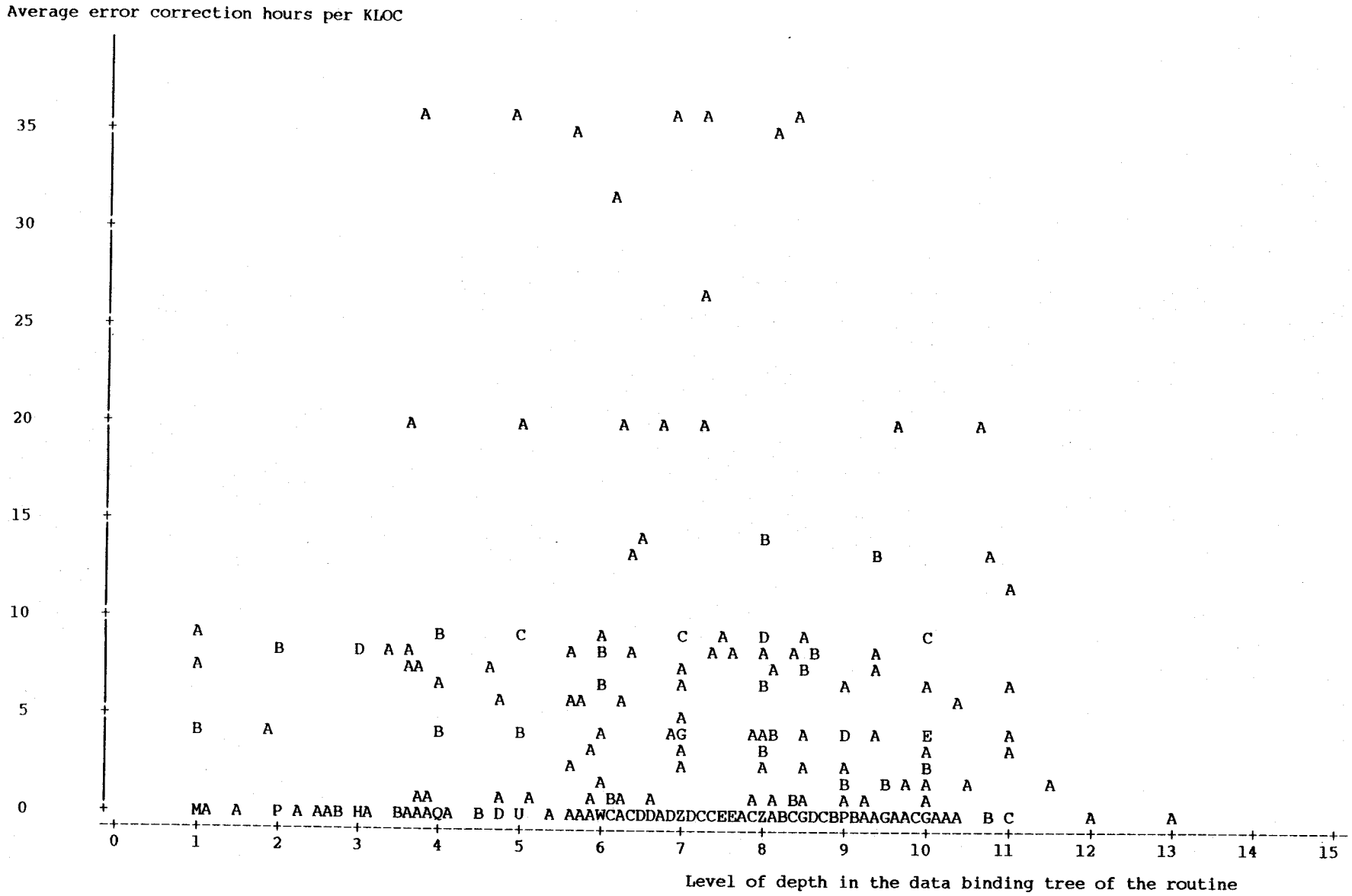


Figure 22: Distribution of errors and error correction effort across routine coupling/strength ratios and routine tree location.

Routine coupling/ strength	Routine tree location	Errors				Error correction hours			
		per KLOC		Total		per KLOC		Total	
		Mean	Std	Mean	Std	Mean	Std	Mean	Std
4_Highest	4_Root	2.26	4.71	0.55	1.07	3.85	8.09	1.10	2.74
	3_Shallower	2.37	4.71	0.63	1.06	6.85	12.05	2.33	4.68
	2_Deeper	1.62	2.67	0.45	0.63	6.06	11.12	2.10	4.49
	1_Deepest	1.33	1.88	0.83	1.18	0.27	0.38	0.17	0.24
3_Higher	4_Root	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	3_Shallower	1.31	4.26	0.36	0.95	0.93	3.89	0.27	1.29
	2_Deeper	1.23	2.75	0.36	0.68	3.58	9.11	1.21	3.41
	1_Deepest	1.04	2.55	0.32	0.64	1.77	5.49	0.58	2.08
2_Lower	4_Root	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	3_Shallower	2.04	4.99	0.68	1.53	1.36	3.26	0.44	0.90
	2_Deeper	0.93	2.65	0.25	0.69	1.43	3.96	0.42	1.63
	1_Deepest	1.61	4.54	0.47	1.19	2.04	5.17	0.65	2.00
1_Lowest	4_Root	0.35	0.89	0.23	0.60	0.25	1.19	0.07	0.32
	3_Shallower	0.09	0.87	0.03	0.21	0.10	0.74	0.03	0.21
	2_Deeper	0.15	0.90	0.05	0.23	0.16	1.01	0.04	0.24
	1_Deepest	0.46	2.41	0.11	0.43	0.21	1.10	0.06	0.30
Overall		1.28	3.58	0.38	0.92	2.39	7.03	0.78	2.63

5. Routines in data binding tree location region 3_SHALLOWER had more errors per KLOC and more error correction effort per KLOC than did routines in the other tree regions.

6. Small routines had more unit and integration test (ESW) errors per KLOC than did large routines.

7. Large routines had more error correction effort per KLOC than did small routines when either all errors or just unit and integration test (ESW) errors were considered.

8. Large routines tended to have a higher average amount of correction effort per error for unit and integration test (ESW) errors than did small routines.

6.5 Data Bindings for System Documentation and Evaluation

The following observations resulted from dialogue with project personnel regarding the data binding trees generated.

1. The data binding clusterings were able to detect major system data structures.
2. 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.
3. Analyzing the clusters of data bindings provided insights to the development and maintenance team.

7 Interpretations and Conclusions

In this study, we have merged two goals:

- To collect and analyze data from an ongoing software project without negatively impacting the software developers; and
- To investigate the software engineering principles of coupling and strength (or cohesion) and their relationship to software errors and error correction effort.

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.

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% 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% 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 of the errors found during inspections were "extras."

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 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 times more effective in terms of errors found per inspection than engineering inspections. Design inspections were 3.6 times more effective in finding major severity errors.

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 general, with clearly indicated required/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.

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 4.3 times fewer errors per KLOC than did larger subsystems.
- Smaller routines had a slightly higher average of errors per KLOC than did larger routines, although the difference was not statistically significant. When just unit and integration test errors are considered, however, smaller routines had significantly more errors per KLOC than did larger routines. Overall, errors in smaller routines were 2.4 times less expensive to fix.

Coupling/Strength

High strength and low coupling are desirable.

- Routines with the lowest coupling/strength ratios had 8.1 times fewer errors per KLOC than routines with the highest coupling/strength ratios and errors were 27.9 times less costly to fix.
- Large subsystems with high coupling/strength ratios had routines with 4.6 times more errors per KLOC than did the other categories of subsystems.

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 50% 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 21% fewer errors per KLOC.

8 Acknowledgement

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 non-disclosure agreement. The authors appreciate the assistance of D. Hutchens in developing the data bindings analysis tools and S. Wilkin in collecting the data.

References

[Bas85] Victor R. Basili. Quantitative evaluation of software methodology. In *Proceedings of the First Pan Pacific Computer Conference*, Melbourne, Australia, Sept. 10–13 1985.

[BE82] L.A. Belady and C.J. Evangelisti. System partitioning and its measure. *Journal of Systems and Software*, 2(1):23–29, February 1982.

[BK83] V. R. Basili and E. E. Katz. Metrics of interest in an ada development. In *IEEE Workshop on Software Engineering Technology Transfer*, pages 22–29, Miami, FL, April 1983.

[Boe81] B. W. Boehm. *Software Engineering Economics*. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[BP84] V. R. Basili and B. T. Perricone. Software errors and complexity: an empirical investigation. *Communications of the ACM*, 27(1):42–52, Jan. 1984.

[BS84] V. R. Basili and R. W. Selby. Data collection and analysis in software research and management. In *Proceedings of the American Statistical Association and Biometric Society Joint Statistical Meetings*, Philadelphia, PA, August 13-16, 1984.

[BSP83] V. R. Basili, R. W. Selby, and T. Y. Phillips. Metric analysis and data validation across fortran projects. *IEEE Trans. Software Engr.*, SE-9(6):652–663, Nov. 1983.

[BT75] V. R. Basili and A. J. Turner. Iterative enhancement: a practical technique for software development. *IEEE Transactions on Software Engineering*, SE-1(4), Dec. 1975.

[BW84] V. R. Basili and D. M. Weiss. A methodology for collecting valid software engineering data*. *Trans. Software Engr.*, SE-10(6):728–738, Nov. 1984.

[Eme84] T. Emerson. A discriminant metric for module cohesion. In *Proc. Seventh Intl. Conf. Software Engr.*, pages 294–303, Orlando, FL, 1984.

[Eve80] B. S. Everitt. *Cluster Analysis, 2nd ed.* Heineman Educational Books Ltd., London, 1980.

[Fag76] M. E. Fagan. Design and code inspections to reduce errors in program development. *IBM Sys. J.*, 15(3):182–211, 1976.

[Fag86] M. E. Fagan. Advances in software inspections. *Trans. Software Engr.*, SE-12(5):744–751, July 1986.

[Glo79] S. A. Gloss-Soler. *The DACS Glossary: A Bibliography of SoftwareEngineering Terms.* Technical Report GLOS-1, Data & Analysis Center for Software, Griffiss Air Force Base, NY 13441, Oct. 1979.

[HB85] D. H. Hutchens and V. R. Basili. System structure analysis: clustering with data bindings. *IEEE Trans. Soft. Engr.*, SE-11(8), Aug. 1985.

[HK81] S. Henry and D. Kafura. Software quality metrics based on interconnectivity. *Journal of Systems and Software*, 2(2):121–131, 1981.

[Hut87] David H. Hutchens. *Software Tools for Data Bindings Analysis.* Technical Report, Dept. of Computer Science, Clemson University, Clemson, SC, 1987. (in preparation).

[IEE83] IEEE. *IEEE Standard Glossary of Software Engineering Terminology.* Technical Report IEEE-STD-729-1983, IEEE, 342 E. 47th St, New York, 1983.

[Ins82] SAS Institute. *Statistical Analysis System (SAS) User's Guide.* Technical Report, SAS Institute Inc., Box 8000, Cary, NC, 27511, 1982.

[JS71] N. Jardine and R. Sibson. *Mathematical Taxonomy.* John Wiley and Sons, New York, 1971.

[KR78] B. W. Kernighan and D. M. Ritchie. *The C Programming Language.* Prentice Hall, Englewood Cliffs, New Jersey, 1978.

[Sch59] H. Scheffe. *The Analysis of Variance.* John Wiley & Sons, New York, 1959.

[Sel85] R. W. Selby. *Evaluations of Software Technologies: Testing, CLEANROOM, and Metrics.* Technical Report TR-1500, Ph.D. Dissertation, Dept. Com. Sci., Univ. Maryland, College Park, 1985.

[SMC74] W. P. Stevens, G. J. Myers, and L. L. Constantine. Structural design. *IBM Systems Journal*, 13(2):115–139, 1974.