INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

University Microfilms International
A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
313/761-4700 800/521-0600

Order Number 9501638

Multivariate modeling of software engineering measures

Lanning, David Lee, Ph.D.

Florida Atlantic University, 1994

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

MULTIVARIATE MODELING OF SOFTWARE ENGINEERING MEASURES

by

David Lee Lanning

A Dissertation Submitted to the Faculty of

The College of Engineering

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy in Computer Science

Florida Atlantic University

Boca Raton, Florida

August 1994

MULTIVARIATE MODELING OF SOFTWARE ENGINEERING MEASURES

by

David Lee Lanning

This dissertation was prepared under the direction of the candidate's dissertation advisor, Dr. Taghi Khoshgoftaar, Department of Computer Science and Engineering and has been approved by the members of his supervisory committee. It was submitted to the faculty of the College of Engineering and was accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science.

SUPERVISORY COMMITTEE:

Taghi M. Khoshgoftaar
Dissertation Advisor,
Taghi M. Khoshgoftaar

G B Fernandez
Eduardo B. Fernandez

R B Fran
Robert B. France

June Perritt
June D. Perritt

Alan B. Marcovitz
Chairperson, Department of Computer Science and Engineering

Dean, College of Engineering

Dean of Graduate Studies and Research

6/22/94
Date

ACKNOWLEDGEMENTS

Financial support and loans of both hardware and software products from the IBM Corporation made this study possible. I thank June Perritt and Shon Saliga for their active roles in providing this support. I also thank my IBM colleagues at Boca Raton, David Blaschke, Glenn Brew, Dan Heacock, Jim Macon, David Medina, and Allen Wynn for the professionalism and dedication that made my absence practical. In addition, I thank Allen Wynn for assisting in the collection and analysis of some data sets that I used in this study.

My advisor, Dr. Taghi Khoshgoftaar, provided invaluable insights, guidance, and inspiration that shaped and sustained this study. I sincerely acknowledge his help, and respect his dedication. I thank my other committee members, Dr. Eduardo Fernandez, Dr. Robert France, and June Perritt, for their constructive comments and excellent feedback during intermediate reviews, and the members of Dr. Khoshgoftaar's measurement group, Ed Allen, Robert Szabo, and Timothy Woodcock, for their collaboration, especially Robert Szabo and Timothy Woodcock for sharing data sets that I used in this study. I thank Lynnae Ehley for sharing with me the trials and tribulations unique to the first students to a new Ph.D. program.

Finally, I acknowledge the encouragement and emotional support of my wife, Robin, my brother, Dennis, and my parents, Viola and Keith. I thank Robin for

enduring those times when this work dominated my attention, and my son, Dennis, for allowing me to postpone many fishing trips.

ABSTRACT

Author:	David Lee Lanning
Title:	Multivariate Modeling of Software Engineering Measures
Institution:	Florida Atlantic University
Dissertation Advisor:	Dr. Taghi Khoshgoftaar
Degree:	Doctor of Philosophy in Computer Science
Year:	1994

One goal of software engineers is to produce software products. An additional goal, that the software production must lead to profit, releases the power of the software product market. This market demands high quality products and tight cycles in the delivery of new and enhanced products. These market conditions motivate the search for engineering methods that help software producers ship products quicker, at lower cost, and with fewer defects.

The control of software defects is key to meeting these market conditions. Thus, many software engineering tasks are concerned with software defects. This study considers two sources of variation in the distribution of software defects: software complexity and enhancement activity. Multivariate techniques treat defect activity, software complexity, and enhancement activity as related multivariate concepts. Applied techniques include principal components analysis, canonical correlation analysis, discriminant analysis, and multiple regression analysis. The objective

of this study is to improve our understanding of software complexity and software enhancement activity as sources of variation in defect activity, and to apply this understanding to produce predictive and discriminant models useful during testing and maintenance tasks. These models serve to support critical software engineering decisions.

To

my wife Robin

my children Roberta, Angela, and Dennis

my parents Viola and Keith

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

1.0.1 Motivation and Objective

One goal of software engineers is to produce software products. An additional goal, that the software production must lead to profit, releases the power of the software product market. This market demands high quality products and tight cycles in the delivery of new and enhanced products. These market conditions motivate the search for engineering methods that help software producers ship products quicker, at lower cost, and with fewer defects. Such methods fall broadly into two classes:

- *macromethods*, which specify disciplines or frameworks within which the entire engineering process is confined, and
- *micromethods*, which offer techniques useful at various generic decision points within whatever process is employed.

Many results have appeared in both classes. This work considers micromethods. For a fresh critique of macromethods refer to [1].

Many software engineering tasks are concerned with software defects. This study considers two sources of variation in the distribution of software defects: software complexity and enhancement activity. Multivariate techniques treat defect activity, software complexity, and enhancement activity as related multivariate concepts. The software products providing data for this study are large commercial products written in procedural languages to run on a single processor. Thus, we do not study either object-oriented or multiprocessing environments.

The objective of this study is to improve our understanding of software complexity and enhancement activity as sources of variation in defect activity, and to apply this understanding to produce predictive and discriminant models useful during testing and maintenance tasks. These models, in serving to support critical software engineering decisions, are software engineering micromethods.

1.0.2 Fundamental Concepts

It is important to differentiate a number of related terms. In a limited sense, a *software product* is an executable resource delivered to a customer. From appearance as a concept to delivery to a customer, such a product passes through a number of forms as software process tasks identify, define, and refine it. The output, or product, of one task in this process serves as the input to another. For example,

- the requirements specification is a product of the requirements gathering task, and this product serves as input to the design task;

- the design specification is a product of the design task, and this product serves as input to the implementation task;
- the source code specification is a product of the implementation task, and this product serves as input to the build and integration task;
- an executable file is a product of the build and integration task and this product serves as input to the system testing task;
- a tested executable file is a product of the system testing task; this product provides the executable resource for delivery to a customer.

Each product in this sequence is actually the same software product expressed at a different level of abstraction, or with added validation. Thus, in a broader sense, a software product is the collection of software process products that ultimately result in an executable resource delivered to a customer. *Software product* takes this sense in the remainder of this work.

Software products often solve complex problems. In arriving at software solutions to complex problems, software engineers often apply stepwise refinement, breaking the problem into multiple simpler subproblems having solutions that collectively solve the original problem [2]. The subproblems themselves are often complex, and thus the process of stepwise refinement iterates until it yields subproblems that are easy to solve directly. This process begins during the design task and continues into the implementation task. Thus, for complex software products, we typically

find a hierarchy of conceptual objects partitioning the source code specification. For example, a line of the source specification might belong, conceptually, to a system, a subsystem, a component, a file, and a function. Depending on the level of granularity of interest, the collection of objects at any of these levels of abstraction might serve as observations for analysis. When the level of interest is unimportant, we refer to observations drawn from source code specifications as *modules*. When the level of interest is important, we specify the level of abstraction to which observations belong.

In this study, a software product fault, or *defect*, results from a human error [3]. Most software engineering tasks are concerned with defects. The requirements gathering task attempts to identify the features that are necessary in the target market. Necessary features unidentified, or unnecessary features identified as requirements are requirements defects. The design and implementation tasks attempt to produce a product that meets the identified requirements. Where requirements are not met there are design or implementation defects. Testing tasks attempt to isolate and remove these defects. Maintenance tasks respond to product defects affecting customer operations. Thus, the distribution of defects across the modules of a software product is important throughout the software development process.

One could discover a defect through inspection, or by observing a *failure*, that is, by observing a departure of software execution from its specified behavior

[3]. Engineers remove discovered defects by changing, at least, the source code specification, that is, by deleting, adding, or modifying program text. Some of these source code specification *changes* imply design specification changes, and some of these imply requirements specification changes. Those that imply design specification changes are *design changes*. Those that imply requirements specification changes either add features or enhance existing features. These changes incorporate *enhancements*. Thus, some design changes respond to design defects while others respond to changing requirements.

A software development process can incorporate *software quality models* to provide an ongoing development effort with feedback from past development efforts. These models exploit the relationships between module attributes that are known early in the development effort and module quality attributes that are known only after a considerable period of test and operation. A past effort provides data used to fit a software quality model, and this model provides predictions useful during the current effort. In this way, knowledge of past performance acts to enhance future performance. For example, using a software quality model, software engineers can predict the number of faults that testing and operation will reveal, and identify high-risk programs [4, 5]. Using these predictions, the engineers can better estimate costs, set schedules, and allocate resources.

1.0.3 Dissertation Overview

Software complexity is a key concept in this dissertation. Chapter 2 gives a brief survey of software complexity literature. This survey provides the history necessary for understanding both where this evolving concept originated, and where it is heading. After defining software complexity in its historical context, this chapter defines the source code and process measures that indicate software complexity and related concepts in the remainder of this work.

Source-code-measure principal components have emerged as important in software quality modeling. Software quality models based upon principal components implicitly assume that source-code-measure principal components are stable. Chapter 3 considers the stability of these principal components. This chapter provides an overview of principal components analysis, and of Krzanowski's method for between-groups comparison of principal components. Using these methodologies, the chapter considers the stability of source-code-measure principal components across the revisions of a single software product throughout its lifecycle, across distinct products developed within the same software development organization, and across distinct products developed by distinct software development organizations. Finally, this chapter considers the impact of source-code-measure principal components instability on software quality models.

To model and analyze the software development process one must consider the relationships between many measures. Often these measures fall in two sets

having a causal relationship that is interesting to software engineers. Chapter 4 applies canonical correlation analysis to investigate the causal relationships between sets of software engineering measures. This chapter describes the canonical correlation modeling technique both as a generalization of the linear regression modeling techniques already applied for this purpose, and as a restricted form of a more general modeling technique. By offering both perspectives, Chapter 4 demonstrates how application of canonical analysis both builds upon past software engineering modeling efforts, and provides direction for future efforts.

Typically a few program modules account for much of the effort expended in developing software products. Researchers have demonstrated that discriminant models based upon source code measures are useful in identifying these high-risk modules early in the development process. These models consider a single source of variation: software complexity. Chapter 5 investigates discriminant modeling of high-risk modules based upon two sources of variation: software complexity and enhancement activity.

Many software development organizations iteratively develop enhanced versions of existing products. The impact of enhancement activity on defect activity is important to the software engineers in these organizations. Chapter 6 investigates enhancement induced defect activity with a canonical correlation model, and, using the results of this model, develops a model for predicting enhancement induced defect activity.

Finally, Chapter 7 brings together the contributions developed in the other chapers, and offers suggestions for future work.

Chapter 2

THE GENERAL CONCEPT OF SOFTWARE COMPLEXITY

2.0.4 Some Historical Background

Some program modules are easy to understand, easy to modify, and account for little of the expense in the development of the software products of which they are components. Other program modules seem almost beyond comprehension, even to their authors. These modules are nearly impossible to modify without inserting multiple defects, and account for much of the expense in the development of the software products of which they are components. Between these extremes lies a range of modules of intermediate complexity. Program modules have many attributes that, considered together, account for some of this variability.

A collection of source code measures quantify program attributes that are related to complexity. Source code measures taken on the modules of a given software product tend to vary in groups with underlying conceptual sources of variation. Measures of different groups quantify attributes related to distinct sources of variation. Consider, for example, Halstead's software science measures [6], and McCabe's

cyclomatic complexity number [7]. Four of Halstead's measures can not be decomposed into other measures:

1. N_1, the total number of operators in a program;
2. N_2, the total number of operands in a program;
3. η_1, the number of unique operators in a program; and
4. η_2, the number of unique operands in a program.

From these primitive measures, Halstead composed non-primitive measures including:

1. $N = N_1 + N_2$, program length;
2. $V = N \log_2(\eta_1 + \eta_2)$, program volume; and
3. $\hat{E} = V[(\eta_1 N_2) \ / \ (2\eta_2)]$, estimated effort.

While these measures are sensitive to program size, they are not sensitive to program control flow, that is, program modules with vastly different control flow structure can have identical Halstead measure values. Halstead's measures do not quantify control flow attributes.

McCabe developed a non-primitive measure, the cyclomatic number, which does quantify control flow attributes [7]. Given a strongly connected graph G, the cyclomatic number of G is the number of independent paths in G. This is given by

$$V(G) = e - n + p,$$

where e is the number of edges, n is the number of nodes, and p is the number of connected components.

McCabe applied this graph theory by constructing a program control flow graph. In this directed graph, nodes represent entry points, exit points, segments of sequential code, or decisions in the program. Edges represent control flow in the program. Strong connectivity is satisfied with the addition of an edge from the exit node to the entry node. McCabe observed that, for a structured program with single entry and exit constructs, $V(G)$ is equal to the number of predicates in the program plus one.

While $V(G)$ is dependent on program control flow complexity, it is often independent of program size, that is, program modules with vastly different counts for operators and operands can have identical cyclomatic numbers. Halstead's measures and McCabe's measure quantify distinct program attributes.

Hansen observed that a single attribute can not explain all differences in module complexity [8]. He asserted that N_1 and $V(G)$ quantify complexity along different dimensions, and proposed that lexicographically ordered 2-tuples, $(N_1, V(G))$ quantify complexity. Baker and Zweben pointed out that this approach says little about the relative complexity of two program modules, P_1 and P_2, if P_1 has the pair (i, j) and P_2 has the pair (k, l) where $i < k$ but $j > l$ [9]. In response, Ramamurthy

and Melton proposed a synthesis of Halstead and McCabe measures achieved by allowing the nesting level of certain operators and operands to add weight to N_1 and N_2, respectively [10]. Khoshgoftaar and Munson proposed a linear combination of measures with weight assigned to a constituent measure proportional to the amount of unique variance that it contributes [11, 12, 13]. These responses do not consider the relationship between source code attributes and human understanding; that is, these methods for resolving the problem identified by [9] determine the influence of a source code attribute on complexity without regard to its observed influence on human understanding.

This view, that program complexity exists apart from human experience, is giving way to a new view, that source code measures indicate complexity in relation to their effect on human understanding. The new view gained explicit support from Melton and Curtis. Melton *et al.* noted that measures of code attributes, like N_1 and $V(G)$, are distinct from the more elusive *psychological complexity measures* quantifying notions like understandability [14]. Curtis noted that a measure of psychological complexity must consider aspects of both an object and the people interacting with this object, that is, the complexity of, say, a program module, is related to both the nature of the module and the difficulty experienced by those working with this module [15].

Many results also lend implicit support for the growing view that source code measures indicate complexity in relation to their observed impact on human

understanding. There are currently over 100 source code measures [16]. Extending Hansen's 2-tuples to n-tuples drawn from this selection of measures, and relating the n-tuples for a set of program modules to the defect content of these modules, we get models that imply what Curtis stated: the complexity of a module is related to both the nature of the module and the difficulty experienced by those that work with it. In these models, the interaction of an n-tuple element with defect content determines the influence of this element in quantifying program complexity, and this quantity is directly related to difficulty experienced by those who worked on the program modules. Briand *et al.* [17], Briand and Basili [18], Gill and Kemerer [19], Henry and Wake [4], Khoshgoftaar *et al.* [20, 21, 5], Lind and Vairavan [22], Porter and Selby [23], and Selby and Porter [24] each present models that predict defect content or related change activity, or discriminate with one of these serving as the criterion. These models predict or discriminate based upon the interaction of source code measures with defect content or related change activity observed in past experience. Model predictions of defect content serve as leading indicators; they also serve to quantify complexity in relation to defect content. Similarly, discrimination serves to provide leading indications; it also serves to segregate program modules into complexity classes that are defined in relation to defect content.

Khoshgoftaar and Munson noted that large collections of source code measures often display multicolinearity, a condition leaving them in violation of an assumption on the independent variables for multiple regression models [25]. To

eliminate this problem, [25] derived the principal components of the source code measures intended as independent variables before fitting multiple regression models. These orthogonal linear combinations then served in place of the source code measures as input to model selection, and those selected served as the independent variables of a multiple regression model predicting defect activity. Regression models fitted with the principal components of source code measures typically display better predictive quality and stability than those fitted with interrelated source code measures [25]. Similar benefits are demonstrated in applications of this technique when building discriminant models [26]. Apart from providing these immediate practical benefits, these results mark a transition in software complexity research. The application of principal components analysis casts source code measures as indicators of software complexity rather than as direct measures of software complexity. So *software complexity* is a multidimensional concept relating source code attributes to human understanding, *and* no single source code measure can serve to perfectly quantify even a single dimension of this concept. This approach stimulates much of the research reported in the remainder of this work. We do not seek to characterize general notions of software complexity by a single real value, but rather, treat software complexity as an indirectly observed multidimensional phenomenon indicated by sets of directly observed and interacting variables [27, 28, 29].

2.0.5 Source Code Measures Related to Software Complexity

As mentioned previously, there are currently over 100 source code measures related to software complexity [16]. This research draws indicators of software complexity from the following 20 source code measures:

1. *XQT* is the number of executable statements [30].

2. η_1 is the number of unique operators [6].

3. N_1 is the total number of operators [6].

4. η_2 is the number of unique operands [6].

5. N_2 is the total number of operands [6].

6. *FFIN* is the number of calls to the function [31] [32].

7. *FFOT* is the number of calls out of the function [31] [32].

8. *GF* is the number of global references [33].

9. $V_1(G)$, McCabe's cyclomatic number, is given by

$$V_1(G) = e - n + 2$$

where e is the number of edges, and n is the number of nodes in the control flow graph [7].

10. $V_2(G) = V_1(G) + \textit{logical operator count}$ is the extended cyclomatic complexity.

11. *Band* Belady's measure, is given by

$$Band = \sum_{i=1}^{m} il_i/n$$

where m, n, and l_i are, respectively, the level of the most nested node, the number of nodes, and the number of nodes at level i in the control flow graph [34].

12. *Paths* is the number of distinct paths in the control flow graph.

13. *MPath* is the length of the longest path in the control flow graph.

14. $\overline{Path}$ is the average path length in the control flow graph.

15. *Loops* is the number of loops in the control flow graph.

16. *Nodes* is the number of nodes in the control flow graph.

17. *Edges* is the number of edges in the control flow graph.

18. *Data* represents the data structure complexity measure [35]. This measure uses a set of values defined for each instance of a data structure combined by the number of variables of each type that are defined by the program module to produce a single number.

19. *Knots* is the number of times the control flow crosses itself [36]. Programs constructed exclusively from the basic structures for sequence, selection, and iteration typical of high level languages will have no knots. These structures

have both a single entry and a single exit through which control always flows. Sequence, selection, and iteration are encapsulated between these points. Using a GOTO statement within such a structure to transfer control to an instruction within another structure will produce at least one knot.

In assembly language, structures for selection and iteration are constructed using test and jump instructions. While these constructions are often more clear when they emulate the high level language structures, the smallest or most efficient coding will often require control flow structures with multiple entry and exit points. Since size and efficiency are typically important in assembly language implementations, knots are common in assembly language code.

20. *AICC*, the average information content classification, is given by

$$AICC = -\sum_{i=1}^{\eta_1} \frac{f_i}{N_1} \log_2 \frac{f_i}{N_1}$$

where f_i is the number of occurrences of the i^{th} operator, η_1 is the number of unique operators, and N_1 is the total number of operators [37]. This applies information theory to software complexity. In this theory, a *message* is a string over an alphabet, $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_{|\Sigma|}\}$. A symbol $\sigma_i \in \Sigma$ occurring with probability p_i in a message provides

$$I_i = -\log_2 p_i$$

bits of information. The entire alphabet provides an average of

$$H = -\sum_{i=1}^{|\Sigma|} p_i \log_2 p_i$$

bits of information per symbol. This quantity is called the *entropy of the information source*, or the *language entropy*. *AICC* is derived by considering the program text to be a message on an alphabet consisting of the unique operators used in the program text. Thus, $|\Sigma| = \eta_1$, $p_i = f_i/N_1$, and $H = AICC$. Programs that yield lower language entropy are assumed to be more complex.

Note that we do not present this as a perfect collection of source code measures for indicating software complexity. In this research, we use these measures to demonstrate various concepts and techniques. We do not intend to justify the use of any particular selection of source code measures. Before collecting source code measures, software engineers must select a set that is suitable in their unique environment. Since this selection process stands apart from the techniques we present, presentation of the details of this process would needlessly obscure the purpose of this work. For details regarding the selection and validation of source code measures, refer to [35, 38, 39].

2.0.6 Process Measures Related to Software Complexity

As Laprie noted, software products are intellectual constructions, and thus, the methods and practices of their production process have a strong influence on

their reliability [40]. This observation, which Laprie put to use to provide reliability estimates at finer granularity, echoes Curtis' observation that a measure of psychological complexity must consider aspects of both an object and the people interacting with this object [15]. Since we hold that source code measures indicate software complexity in relation to their effect on human understanding, we consider the source code measures defined in the previous section in relation to a collection of process measures. These process measures indicate defect activity, a measure of difficulty experienced by those working on the product. This section defines this collection of process measures.

On entry to the system test phase, the source code specification, a collection of modules, defines the software product. During a *build*, software tools translate this collection of modules into an executable product. Individuals can create private versions of the executable product using a *private* build. By running test cases against a private version of the executable product, individual developers assess the quality of their new code. The system test organization runs test cases against the executable product created by a *public* build. A system test case failure on this product generates a defect report. Troubleshooting resolves the defect report by isolating the cause of the failure to defective statements in one or more modules. Correction of these defective statements requires source code changes. A defect report requiring changes to multiple modules records a defect against each of these modules.

If correcting the defective statements results in a change to the design of the software product, then a design change report replaces the defect report for tracking the associated source code changes. Design change reports also track source code changes required to implement new functionality. A design change report requiring changes in multiple modules records a design change against each of these modules. Those design changes required to implement new functionality cause changes to implement enhancements.

Source code changes accumulate without affecting the executable product until the next build. At this time, system testing stops and software tools translate the collection of modules into a new executable product, which incorporates the accumulated source changes. System testing resumes on this new executable product. Typically builds are scheduled periodic events, and system testing progresses without interruption from build to build.

Failures have various impacts on the system testing effort. Some have an extreme impact on the system testing cost and schedule; others have little or no impact. Some system testing organizations apply the following classification scheme to account for these differences:

- *Severity 1* failures prevent the application of all system test resources. For example, a critical test case hangs the machine, and there is no way to work around the problem to allow other test cases to continue.

- *Severity 2* failures prevent the application of most system test resources. For

example, a critical test case hangs the machine, but it is possible to work around the problem to allow some other test cases to continue.

- *Severity 3* failures prevent the application of some system test resources. For example, a test case fails with invalid output. However it is possible to work around this problem to allow other test cases depending upon this output to continue running.

- *Severity 4* failures do not prevent the application of any system test resources. For example, the test case identifies a message having a misspelled word.

As the scheduled system test completion date approaches, less time remains to complete the test plan. Thus, failures that block system test progress became more critical. At this time, the system testing organization often requires demonstration that a private version of the executable product incorporating the proposed changes executes critical test cases without failures before allowing the changes to enter the next public build of the executable product. Further, at this time, the system testing organization becomes reluctant to allow design changes that introduce new functionality.

The following 18 process measures indicate the defect and enhancement activity applied to each module [41, 42]:

1. D is the total number of defects.

2. D_1 is the number of defects that resulted in severity 1 failures.

3. D_2 is the number of defects that resulted in severity 2 failures.

4. D_3 is the number of defects that resulted in severity 3 failures.

5. D_4 is the number of defects that resulted in severity 4 failures.

6. D_c is the number of design changes.

7. E is the number of functional enhancements.

8. A is the number of noncomment source lines added.

9. C is the number of noncomment source lines modified.

10. R is the number of noncomment source lines removed.

11. M is the number of noncomment source lines moved.

12. A_e is the number of noncomment source lines added to provide enhancements.

13. C_e is the number of noncomment source lines changed to provide enhancements.

14. R_e is the number of noncomment source lines removed to provide enhancements.

15. A'_e is the number of noncomment source lines added to introduce a new module that is required to provide enhancements. While an engineer could define a new module to remove a defect, this is unusual in the software development

environments under study. Further, while these changes could be related to a functional enhancement, they can not be related to functional enhancement of the added module. Thus, we do not consider the number of noncomment source lines added to introduce a defect-removing module, A'_d, as an indicator of enhancement induced defect activity.

16. A_d is the number of noncomment source lines added to remove defects.

17. C_d is the number of noncomment source lines changed to remove defects.

18. R_d is the number of noncomment source lines removed to remove defects.

Note that we do not present this as a perfect collection of process measures for indicating defect and enhancement activity. In this research, we use these measures to demonstrate various concepts and techniques. We do not intend to justify the use of any particular selection of process measures. The availability of these measures varies due to differences in the data collection efforts providing historical data. For example, many organizations fail to distinguish the following pairs: (D_c, E), (A_e, A_d), (C_e, C_d), (R_e, R_d). For the same reason, some definitions vary across data sets. For example, some organizations do not record C, C_e, or C_d, but rather, record each changed line as one line added and one line deleted. We will make distinctions of this type for each data set.

Chapter 3

THE STABILITY OF SOURCE-CODE-MEASURE PRINCIPAL COMPONENTS

For a set of source code measures, principal components analysis produces an equal sized set of orthogonal linear combinations, or principal components [43]. The principal components reflect the common structure underlying the source code measures. Typically, a few of the principal components account for most of the variability seen in the original set of measures. Selecting models from these significant principal components reduces the dimensionality of quality models, and satisfies the assumption of nonmulticolinearity. However, models selected from principal components implicitly assume that the principal components of a modeled software product remain stable through out the modeling application. That is, these models assume that the product used to fit a model and the product to which this model is applied for predictive purposes display source code measures having a common underlying structure.

Confirmatory factor analysis acts to confirm or negate an hypothesized structure underlying data [43]. Thus, given that source code measures have multivariate normal distributions—a strong assumption underlying the model—one could

test the structural stability of the measures throughout a development effort using confirmatory factor analysis. Extending the stability question, one could also test the structural stability of the measures across distinct product development efforts within the same organization, and finally to distinct product development efforts across organizations. Unfortunately, researchers have found that source code measures are not normally distributed, and that successful transformations to normality are not likely [44] [45]. We have confirmed this independently for collections of measures from several software development environments. Thus, confirmatory factor analysis can not serve to confirm or refute the stability of the structure underlying a set of source code measures.

Fortunately, Krzanowski offered a method for between-groups comparison of principal components that has no distributional assumptions [46, 47]. We applied this method to test the structural stability of measures across the revisions of a single software product throughout its lifecycle, across distinct products developed within the same organization, and across distinct products developed by distinct organizations [48]. For the products that we studied, we found that measures are structurally stable across the revisions of a single product, and across similar products developed by the same organization. However, we found little structural stability across products developed by different organizations.

Schneidewind noted that source code measure based software quality models can give inconsistent results across development projects due to variations in product

domains and other product characteristics, as well as variations in process maturity levels, development environments, and the skill and experience of people [49]. To minimize the risk in applying quality models, [49] proposed a procedure to assess the stability of a simple regression model across two software product development efforts. The procedure is applied at the completion of one effort and the inception of the other to determine the aptness of a quality model fitted to data from the completed effort for predicting results of the new effort. While this procedure is useful, as presented it is limited to univariate assessments. Repeated applications could extend the procedure to multiple independent variables, however such an extension would not consider interactions between these variables. We know that software complexity is multidimensional, and that source code measures indicating software complexity characteristics have significant interrelationships [25].

To investigate the impact of principal components instability, we use Krzanowski's method to isolate two pairs of products, one pair showing between-groups similarity of principal components, the other dissimilarity. One product is common to both pairs. We fit a software quality model using data collected from this common product, and test the predictive quality of this model using data collected from the remaining two products. This model is based upon the principal components of the common product. We find that the model has greater predictive quality on the product identified as similar.

Thus, this chapter addresses three questions. First, how can software engineering researchers and practitioners quantify stability of principal components across software products? Second, are software product principal components stable across the revisions of a single software product throughout its lifecycle, across distinct products developed within the same organization, and across distinct products developed by distinct organizations? And third, what is the impact of software product principal components instability on software quality models [50]?

Three sections answer these questions. Section 3.1 gives an overview of principal components analysis, and of Krzanowski's method for quantifying the similarity between two sets of principal components. Section 3.2 reports detailed results of our application of Krzanowski's method to determine the degree of stability in software product principal components. Section 3.3 gives an overview of multiple regression modeling, and reports detailed results of our application of Krzanowski's method to determine the impact of software product principal components instability on software quality models. Each of these sections demonstrates the utility of Krzanowski's method in quantifying the stability of principal components across software products.

3.1 An Overview of Between-Groups Comparison of Principal Components

A multivariate data set consists of values for each of m attributes for each of n observations, and thus, can be represented by an n by m matrix. When applying principal components analysis, one typically seeks to account for most of the variability in the m attributes of this matrix with $p < m$ linear combinations of these attributes. For this study, the m attributes are source code measures, and the n objects are the program modules that these source code measures describe. We reduce the m source code measures to $p < m$ linear combinations of these measures by applying principal components analysis to this measure data set.

3.1.1 Principal Components Analysis

Let $\mathbf{R}$ be the correlation matrix for the measure data set. Then $\mathbf{R}$ is a real symmetric matrix, and, assuming that it has distinct roots, can be decomposed as

$$\mathbf{R} = \mathbf{T}\mathbf{\Lambda}\mathbf{T}' \tag{3.1}$$

where

- $\mathbf{\Lambda}$ is a diagonal matrix with the eigenvalues, $\lambda_1 > \lambda_2 > \cdots > \lambda_m$ on its diagonal,
- $\sum_{j=1}^{m} \lambda_j = m$,
- $\mathbf{T}$ is an orthogonal matrix where column j is the eigenvector associated with λ_j, and

- $\mathbf{T}'$ is the transpose of $\mathbf{T}$ [43].

The m eigenvectors in $\mathbf{T}$ give the coefficients that define m uncorrelated linear combinations of the original source code measures. These orthogonal linear combinations are the principal components of the source code measures. λ_j/m gives the percent of source-code-measure variance that is explained by the j^{th} principal component. Thus, each successive principal component explains less variance in the data set.

Figure 1 offers a geometric interpretation of the principal components. This figure shows a swarm of points along two pairs of axes (x_1, x_2) and (x_1', x_2'). Let (x_1, x_2) define a coordinate system. Rotating these axes counterclockwise through an angle of α yields a new coordinate system defined by $(x_1',\ x_2')$. The orthogonal projections of a point onto the axes of a coordinate system give the coordinates of the point in this coordinate system. For example, the orthogonal projections of point P_i onto the rotated coordinate system identify $P_i = (x_{1i}', x_{2i}')$ in this coordinate system. The spread of projections along an axis corresponds to the variance explained by a principal component. A principal components analysis selects α such that the spread of orthogonal projections onto each successive rotated axes decreases. Thus in Figure 1, the spread of orthogonal projections onto x_1' exceeds the spread of orthogonal projections onto x_2'.

Since the variance explained by each successive principal component decreases, the first few principal components can explain a large proportion of the

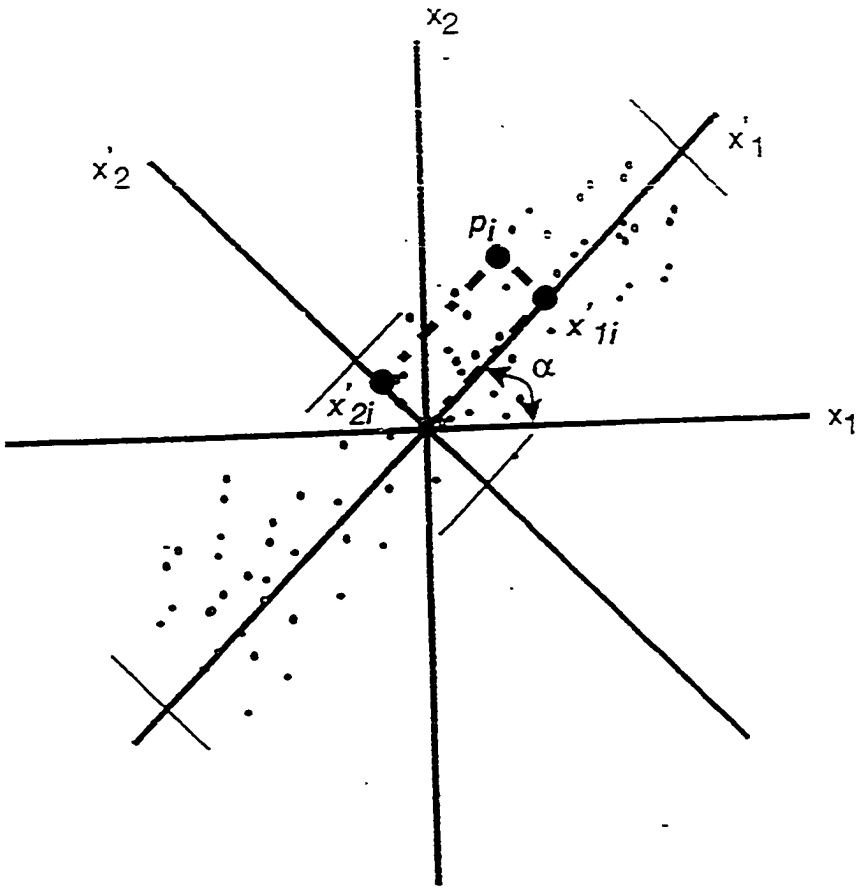

Figure 3.1: A geometric interpretation of the principal components

total variance. Thus restricting attention to the first few principal components can achieve a reduction in dimensionality with an insignificant loss of explained variance. A stopping rule selects $p < m$ principal components such that each one contributes significantly to the total explained variance, and the p selected components collectively account for a large proportion of this variance.

The standardized transformation matrix, $\mathbf{T}^p$, is constructed from $\mathbf{T}$ by excluding all columns greater than p and transforming the remaining eigenvectors such that the resulting principal components are standardized. That is,

$$\mathbf{P}^p = \mathbf{Z}\mathbf{T}^p = [\mathbf{P}_1, \mathbf{P}_2, \ldots, \mathbf{P}_p]$$

gives an n by p matrix of principal component values, the n values for each principal component $\mathbf{P}'_j = [P_{j1}, P_{j2}, \ldots, P_{jn}]$, $1 \leq j \leq p$, being distributed with zero mean and unit variance.

Each of the m observed measures has a correlation with each of the p selected principal components. Since the principal components are not directly observable, we interpret them by considering these correlations, or *loadings*. The loadings form an m by p matrix, the *loading pattern*. The larger the magnitude of a loading, the stronger the relationship between the associated measure and principal component. Often a measure will have a heavy loading on one component and a relatively low loading on each of the other components. However, in some instances, measure loadings are ambiguous with nearly equal values across multiple components. In the loading patterns that we present, the dominating measure loadings appear in a

boldface font. This highlights the pattern of associations between the measures and the principal components.

The decomposition given in Equation 3.1 is not unique. One can post-multiply $\mathbf{T}$ by an orthogonal matrix $\mathbf{O}$ yielding a new orthogonal matrix $\vec{\mathbf{T}} = \mathbf{TO}$. $\vec{\mathbf{T}}$ and $\mathbf{T}$ are both rotations of the original axes. In Figure 1, the spreads of orthogonal projections onto axes x_1 and x_2 differ from the corresponding spreads onto axes x_1' and x_2'. In the same way, the spreads of orthogonal projections onto the axes defined by $\mathbf{T}$ will differ from the corresponding projections onto the axes defined by $\vec{\mathbf{T}}$. That is, the variances explained by the principal components defined by $\mathbf{T}$ will differ from the corresponding variances explained by the principal components defined by $\vec{\mathbf{T}}$. Let $\vec{\mathbf{\Lambda}}$ give the variance for each principal component given in $\vec{\mathbf{T}}$. Then

$$\mathbf{R} = \vec{\mathbf{T}}\vec{\mathbf{\Lambda}}\vec{\mathbf{T}}'$$

where $\vec{\mathbf{T}}$, $\vec{\mathbf{\Lambda}}$, and $\vec{\mathbf{T}}'$ have the characteristics given in Equation 3.1 for, respectively, $\mathbf{T}$, $\mathbf{\Lambda}$, and $\mathbf{T}'$. The matrices $\mathbf{T}$ and $\vec{\mathbf{T}}$, represent two equally valid interpretations of the data. Those applying principal components analysis often rotate the principal components to achieve a structure that aids interpretation. Varimax rotation, a commonly applied method, seeks to rotate components to produce a large variation in squared loadings [43]. This produces patterns having high, medium, and small loadings within a particular principal component. These rotations often reveal a pattern of loadings that aids in interpretation. Typically, varimax rotations

of principal components have served for software quality modeling [25].

3.1.2 Krzanowski's Method

Krzanowski offered a method for quantifying the similarity between two sets of principal components [46, 47]. This method considers two multivariate samples, A and B, having respectively, n_1 and n_2 observations on the same m variables. The method carries no distributional assumptions, and thus is suitable for software engineering applications. In this setting,

- the two multivariate samples represent two software products, either two versions in the lifecycle of a single software product, or versions of two distinct products,
- the m variables are a collection of source code measures, and
- an observation is the unit of software decomposition at which these measures are collected.

Let each of the m source code measures identify an orthogonal axis in m-dimensional Euclidean space. Two swarms of points in this space identify software products A and B. After principal components analysis on A and B, we have two decompositions

$$\mathbf{R}_A = \mathbf{T}_A \mathbf{\Lambda}_A \mathbf{T}'_A,$$

and

$$\mathbf{R}_B = \mathbf{T}_B \mathbf{\Lambda}_B \mathbf{T}'_B,$$

where $\mathbf{R}$, $\mathbf{T}$, $\mathbf{\Lambda}$, and $\mathbf{T}'$ are defined above, and subscripts on these variables identify the applicable product. Suppose that p principal components are adequate for representing each software product. Then

$$\mathbf{P}^p_A = \mathbf{Z}_A \mathbf{T}^p_A,$$

and

$$\mathbf{P}^p_B = \mathbf{Z}_B \mathbf{T}^p_B$$

identify the two swarms of points in p dimensions. This effectively embeds the two products, A and B, into two p-dimensional subspaces of the original m-dimensional space. Thus, we can compare two sets of principal components by comparing the two p-dimensional subspaces that they create. This comparison considers the angles between the best-matching sets of orthogonal axes for the two subspaces.

Given the two sets of principal components, $\mathbf{T}^p_A$, and $\mathbf{T}^p_B$, that define the two p-dimensional subspaces under comparison, define

$$\mathbf{N}^p = \mathbf{T}^{p\prime}_A \mathbf{T}^p_B \mathbf{T}^{p\prime}_B \mathbf{T}^p_A.$$

For $i = 1, 2, \ldots, p$, let λ_i be the i^{th} largest eigenvalue of $\mathbf{N}^p$ with the associated eigenvector $\mathbf{a}_i$. Further, let $\mathbf{b}_i = \mathbf{T}^p_A \mathbf{a}_i$. Then $\{\mathbf{b}_1, \ldots, \mathbf{b}_p\}$ is a set of mutually orthogonal vectors in subspace B and $\{\mathbf{T}^p_B \mathbf{T}^{p\prime}_B \mathbf{b}_1, \ldots, \mathbf{T}^p_B \mathbf{T}^{p\prime}_B \mathbf{b}_p\}$ is a corresponding

set of mutually orthogonal vectors in subspace A such that, of all vectors in these subspaces, the angle formed between $\mathbf{b}_i$ and $\mathbf{T}_B^p\mathbf{T}_B^{p\prime}\mathbf{b}_i$ is the i^{th} smallest. [46] offers a proof for this result. Thus, the similarities between A and B are exhibited solely by the angles formed by the pairs $(\mathbf{b}_i, \mathbf{T}_B^p\mathbf{T}_B^{p\prime}\mathbf{b}_i)$. These *critical angles*, given by $\cos^{-1}\sqrt{\lambda_i}$, vary from zero to 90 degrees as the two compared principal components vary from coincident to orthogonal.

Further, the matrices, $\mathbf{T}_B^p\mathbf{T}_B^{p\prime}\mathbf{b}$ and $\mathbf{b}$, are orthogonal rotations of, respectively, $\mathbf{T}_A^p$ and $\mathbf{T}_B^p$. That is,

$$\mathbf{T}_B^p\mathbf{T}_B^{p\prime}\mathbf{b} = \mathbf{T}_B^p\mathbf{O}_B,$$

and,

$$\mathbf{T}_A^p\mathbf{a} = \mathbf{T}_A^p\mathbf{O}_A,$$

where $\mathbf{O}_B = \mathbf{T}_B^{p\prime}\mathbf{b}$ and $\mathbf{O}_A = \mathbf{a}$ are orthogonal transformation matrices. Thus $\mathbf{T}_B^p\mathbf{T}_B^{p\prime}\mathbf{b}$ and $\mathbf{b}$ represent alternative representations that aid in the interpretation of differences between the two data sets.

This gives Krzanowski's restricted method for between-groups comparison of principal components in which the number of principal components representing the compared data sets is constant, and comparisons are pairwise. Krzanowski generalized the method for comparison of data sets represented by different numbers of principal components, and to n-way comparisons [46]. For this study, the restricted method serves as well as the more complicated general method, and has the added value of a simpler presentation.

3.2 Principal Components Stability in Software Products

In this section, we investigate the stability of source-code-measure principal components in three ways: across releases of distinct software products produced by distinct software development organizations, across releases of distinct software products produced by the same software development organization, and across several releases of the same software product. These studies include comparisons of C and assembly language software product implementations. Section 3.2.1 describes the measures included for each of these languages. Section 3.2.2 reports the results of investigations of principal component stability across releases of distinct software products produced by distinct software development organizations. Section 3.2.3 reports the results of investigations of principal component stability across releases of distinct software products produced by the same software development organization. Section 3.2.4 reports the results of investigations of principal component stability over several releases of the same software product.

3.2.1 The Source Code Measures

We collected measures from the source code for several software products implemented in C and assembly language using our measure analyzers for these languages. Table 3.1 shows the source code measures collected by each of these analyzers.

Note that principal components analysis is not limited to these selections of measures. Our goal in this chapter, is to demonstrate between-groups comparison

Measure	Language	
	C	Assembly
XQT	•	•
η_1	•	•
N_1	•	•
η_2	•	•
N_2	•	•
FFIN	•	•
FFOT	•	•
GF	•	
$V_1(G)$		•
$V_2(G)$	•	
Band	•	
Nodes	•	
Edges	•	
Data	•	
Knots		•
AICC		•

Table 3.1: Measures Collected

of principal components, and not to justify the use of any particular selection of source code measures.

We presented, and will apply, Krzanowski's restricted method of between-groups principal components in which the number of principal components representing the compared data sets is constant. Thus, we restrict our empirical investigation to comparisons of software products represented by the same number of principal components. Applying Krzanowski's unrestricted method allowing comparison of data sets represented by different numbers of principal components is left to future research.

3.2.2 PC Stability Across Development Organizations

This section reports the results of studies on a set of software products implemented in C by distinct software development organizations. The set of C products consists of 16 public-domain software products for which we collected the source code from various internet sites. Table 3.2 shows the selection of C products for comparison. To characterize the size of these products, this table includes the number of executable statements and the number of functions comprising each product. We represent these products with three principal components.

We found the critical angles for all 120 pairwise comparisons of the 16 products. Most of these comparisons revealed significantly different sources of variation between the products, although some comparisons did reveal striking similarities. We summarize by reporting detailed results for the pairs having the most and the

least similarity, along with the average critical angles seen in the 120 comparisons.

We found that SENDMAIL and BINDX4X8 have the most similar principal components. Table 3.3 gives the varimax rotation of the loadings for these products. These rotations are achieved independently, and thus the loading patterns in this pair of rotations simplify within-product principal components interpretations.

In the varimax rotation for SENDMAIL given in Table 3.3, note that the principal components fall in nonincreasing order of explained variance, and that the measures fall in nonincreasing order of loadings on the dominating principal component. However, in the varimax rotation for BINDX4X8 given in this table, the principal components fall in an order such that the dominated source code measure in each principal component most closely match those of the corresponding principal component for SENDMAIL, and the measures fall in the same order as in the SENDMAIL loading pattern. This eases the comparison of the two loading patterns. We follow this convention for each pair of varimax rotations that we present.

The similarities between the loading patterns given in Table 3.3 are immediately apparent. The third principal component is dominated exclusively by *FFIN* in both of these patterns. Further, for both loading patterns, the first nine measures have strong relationships with the first principal component, *Band* clearly loads on the second principal component, and η_1 loads ambiguously across principal components one and two. The loadings of *Data* establish a major difference between the

Product	XQT	Functions
GNUCHESS	7,342	249
SMALLTAL	6,555	370
BINDX4X8	5,421	154
EC	8,973	99
GAS-1X38	11,884	296
GDB	22,452	787
GHOST	22,091	1,218
GNUPLOT	10,808	430
IMAGEMAG	19,193	169
IRC2	4,722	243
NTP	1,824	46
P2C	26,098	907
PERL-3X0	14,294	264
SENDMAIL	6,724	163
TOP	21,468	1,012
XTIFF	2,838	123

Table 3.2: The C Language Products

loading patterns. In the SENDMAIL loading pattern this measure loads ambiguously across principal components 1 and 2, while in the BINDX4X8 loading pattern this measure loads unambiguously on principal component 1.

Table 3.4 gives the critical angle comparison of SENDMAIL and BINDX4X8 principal components. Table 3.5 gives the loading patterns corresponding to these critical angles. The loading patterns given in Table 3.5 correspond to the dimensions compared in Table 3.4. These rotations create the smallest possible angles across corresponding principal components. Thus these rotations simplify cross-product loading pattern interpretations. For example, the critical angle for dimension 1 shown in Table 3.4 falls between the patterns shown for principal component 1 in Tables 3.5. The similarities indicated by the angles are readily apparent in the loading patterns. For example, in each pair of compared principal components, both principal components dominate the same measures, and the dominating loadings are similar across these principal components. These rotations indicate that the compared systems share a strong component of variation indicated by all measures except *FFIN*, another indicated by *FFIN*, and a third indicated by secondary loadings on *FFOT* and *GF*.

We found that TOP and SMALLTAL have the least similar loading patterns. Table 3.6 gives the varimax rotation of the loading patterns for these products. These patterns are so dissimilar that is impossible to discern even one similar principal component.

Measure	Principal Component					
	SENDMAIL			BINDX4X8		
	1	2	3	1	2	3
GF	**0.92**	0.05	0.04	**0.84**	0.03	0.02
FFOT	**0.92**	0.30	0.08	**0.92**	0.22	0.07
η_2	**0.88**	0.42	0.11	**0.90**	0.38	0.02
$V_2(G)$	**0.87**	0.40	0.20	**0.88**	0.40	0.14
N_2	**0.87**	0.45	0.15	**0.93**	0.34	0.08
Edges	**0.86**	0.43	0.19	**0.91**	0.38	0.12
Nodes	**0.86**	0.45	0.20	**0.91**	0.37	0.11
XQT	**0.85**	0.46	0.19	**0.93**	0.35	0.07
N_1	**0.84**	0.44	0.16	**0.92**	0.34	0.12
Band	0.19	**0.87**	0.21	0.24	**0.89**	0.20
η_1	**0.52**	**0.74**	0.13	**0.51**	**0.78**	0.07
Data	**0.51**	**0.58**	0.03	**0.82**	0.36	-0.02
FFIN	0.16	0.18	**0.96**	0.08	0.17	**0.98**
Eigenvalue	7.48	3.11	1.22	8.34	2.55	1.08
% Variance	57.54	23.92	9.35	64.17	19.59	8.31
Cumulative % Variance	57.54	81.46	90.80	64.17	83.76	92.10

Table 3.3: Varimax Rotations for SENDMAIL and BINDX4X8

	Dimension		
	1	2	3
Eigenvalues	0.999657	0.989034	0.879943
Critical Angles	1.06	6.01	20.27

Table 3.4: Critical Angles Between SENDMAIL and BINDX4X8

	Principal Component					
	SENDMAIL			BINDX4X8		
Measure	1	2	3	1	2	3
Edges	**0.99**	0.19	0.43	**0.98**	0.03	0.44
N_2	**0.98**	0.23	0.46	**0.98**	0.07	0.44
N_1	**0.98**	0.20	0.46	**0.96**	0.05	0.42
XQT	**0.98**	0.24	0.45	**0.98**	0.03	0.43
Nodes	**0.98**	0.20	0.44	**0.98**	0.02	0.43
η_2	**0.97**	0.28	0.43	**0.97**	0.11	0.47
$V_2(G)$	**0.97**	0.15	0.39	**0.97**	0.02	0.45
FFOT	**0.92**	0.25	**0.53**	**0.93**	0.14	**0.56**
Data	**0.88**	0.29	0.30	**0.74**	0.11	0.22
η_1	**0.82**	0.11	-0.11	**0.85**	0.03	-0.03
GF	**0.75**	0.27	**0.66**	**0.79**	0.16	**0.68**
Band	**0.64**	-0.09	-0.40	**0.66**	-0.13	-0.39
FFIN	0.28	**-0.90**	-0.11	0.42	**-0.89**	0.02

Table 3.5: Critical Angle Rotations for SENDMAIL and BINDX4X8

Measure	Principal Component					
	TOP			SMALLTAL		
	1	2	3	1	2	3
Edges	**0.86**	0.33	0.24	0.27	**0.76**	0.18
Nodes	**0.86**	0.33	0.23	0.12	**0.89**	0.32
XQT	**0.85**	0.44	0.10	**0.89**	0.25	-0.02
η_2	**0.81**	0.45	-0.04	**0.90**	0.26	0.00
η_1	**0.80**	0.29	-0.01	**0.60**	0.13	0.14
$V_2(G)$	**0.75**	0.45	0.26	**0.95**	0.09	-0.03
Data	**0.75**	0.05	-0.20	0.07	**0.80**	-0.06
Band	**0.65**	0.18	0.23	0.01	0.15	**0.96**
GF	0.05	**0.90**	0.17	0.02	**0.69**	0.39
N_1	0.42	**0.87**	0.04	**0.96**	0.00	-0.02
N_2	**0.58**	**0.78**	0.02	**0.97**	0.07	-0.02
FFOT	**0.44**	**0.71**	-0.07	0.13	**0.84**	0.04
FFIN	0.10	0.05	**0.89**	0.16	**0.86**	-0.14
Eigenvalue	5.79	3.61	1.11	4.86	4.12	1.25
% Variance	44.51	27.76	8.54	37.38	31.69	9.62
Cumulative % Variance	44.51	72.28	80.81	37.38	69.08	78.69

Table 3.6: Varimax Rotations for TOP and SMALLTAL

	Dimension		
	1	2	3
Eigenvalues	0.994508	0.103832	0.003541
Critical Angles	4.25	71.20	86.59

Table 3.7: Critical Angles Between TOP and SMALLTAL

Table 3.7 gives the critical angle comparison of TOP and SMALLTAL principal components. Table 3.8 gives the loading patterns corresponding to these critical angles. These products show similarity in only one dimension, the remaining two being nearly orthogonal. Clearly, *FFIN*, the measure most dominated by principal component 2, has strong opposing association across the two patterns. In addition, there are few similar loadings across principal component 3. Still, with the exception of *FFIN* and *Band*, principal component 1 dominates the same measures across the compared systems. Thus, despite the total lack of similarity found in Table 3.6, the two spaces are similar along one dimension. These results underscore the need for applying an analytical technique to compare principal components. Results based upon visual inspection are suspect since two sets of components that differ substantially in appearance may in fact define the same subspace of the original multivariate space [46].

Table 3.9 gives the average and the standard deviation of the critical angles across the 120 comparisons. While the best matching software products showed considerable similarities, the average critical angles show that similarity across C

Measure	Principal Component					
	TOP			SMALLTAL		
	1	2	3	1	2	3
XQT	**0.95**	0.17	0.05	**0.83**	-0.25	0.06
N_2	**0.92**	0.33	0.10	**0.77**	-0.43	0.17
Nodes	**0.92**	0.09	-0.09	**0.71**	**0.54**	-0.04
Edges	**0.92**	0.09	-0.09	**0.72**	0.42	-0.23
η_2	**0.90**	0.19	0.12	**0.84**	-0.25	-0.09
$V_2(G)$	**0.90**	0.25	-0.11	**0.77**	-0.39	0.07
N_1	**0.85**	0.33	0.03	**0.73**	-0.48	0.19
η_1	**0.80**	0.30	-0.08	**0.57**	-0.18	-0.22
FFOT	**0.75**	0.12	0.22	**0.63**	0.15	0.21
Band	**0.66**	0.15	**-0.66**	0.29	0.23	-0.21
GF	**0.60**	0.20	-0.12	**0.52**	0.21	**0.51**
Data	**0.59**	-0.15	0.32	**0.53**	0.27	-0.29
FFIN	0.27	**-0.65**	0.37	**0.63**	**0.67**	0.08

Table 3.8: Critical Angle Rotations for TOP and SMALLTAL

	Dimension		
	1	2	3
Avg. Critical Angles	3.85	28.09	64.26
Standard Deviation	1.93	14.99	19.68

Table 3.9: Average Critical Angles Over the C Products

products is atypical. However, similarity in one principal component is typical.

3.2.3 PC Stability Across Products Within a Development Organization

This section reports the results of loading pattern comparisons of distinct software products produced by the same software development organization. Table 3.10 describes the two assembly products available for this analysis. To characterize the size of these products, this table includes the number of executable statements and the number of functions comprising each product. The two products shown in Table 3.10, RTS_1 and RTS_2, are the final versions of two distinct Real-Time Systems. The developers implemented these products in assembler to satisfy space and time constraints. Both products are commercial real-time systems developed to run on the same 32-bit microprocessor. We represent these assembly language software products with five principal components. For each product, these five principal components account for over 90% of the source-code-measure variance.

Table 3.11 gives the varimax rotation of the loading patterns for RTS_1 and RTS_2. These two loading patterns appear similar.

Product	XQT	Functions
RTS_1	137,309	99
RTS_2	224,002	152

Table 3.10: The Assembly Language Products

Table 3.12 gives the critical angle comparison of RTS_1 and RTS_2 principal components. Table 3.13 gives the loading patterns corresponding to these critical angles. Differences across these patterns include an increased association of RTS_2 principal component 2 with *FFIN*, and RTS_2 principal component 4 with *Knots*. Still, nearly all of the principal component loadings are similar across these patterns, and with no critical angle exceeding 22 degrees the patterns are similar.

3.2.4 PC Stability Across Releases of a Product

This section reports the results of studies on a Telecommunications System (TS), a medium-sized commercial real-time software product implemented in C. The TS development history spanned 28 months during which the product evolved through 45 versions taking it through unit test, system test, integration test, and into maintenance. From the 45 versions of TS, we selected five for our analysis. These represent the product

1. as it entered unit test,

2. as it entered system test,

Measure	Principal Component									
	RTS_1					RTS_2				
	1	2	3	4	5	1	2	3	4	5
N_2	**0.98**	-0.09	0.08	0.02	0.04	**0.98**	-0.08	0.05	0.06	0.04
N_1	**0.97**	0.13	0.11	0.14	0.07	**0.96**	0.10	0.09	0.22	0.07
XQT	**0.82**	0.30	0.35	0.24	0.18	**0.82**	0.34	0.18	0.36	0.19
$AICC$	-0.02	**0.95**	0.10	0.07	0.13	-0.08	**0.94**	0.11	0.16	0.04
η_1	0.09	**0.88**	0.33	0.06	0.20	0.14	**0.89**	0.25	0.21	0.14
$V_1(G)$	0.40	**0.65**	0.07	0.48	-0.03	0.36	**0.51**	0.14	**0.65**	-0.05
η_2	0.21	0.11	**0.84**	0.03	0.32	0.31	**0.56**	0.37	0.13	0.34
$FFOT$	0.15	0.38	**0.68**	0.17	-0.24	0.13	0.31	**0.90**	0.18	-0.04
$Knots$	0.14	0.12	0.11	**0.96**	0.00	0.20	0.16	0.14	**0.92**	0.07
$FFIN$	0.13	0.19	0.09	0.01	**0.91**	0.11	0.13	-0.02	0.03	**0.96**
Eigenvalue	2.86	2.51	1.69	1.09	1.02	2.83	2.43	1.45	1.26	1.09
% Variance	28.60	25.10	16.90	10.90	10.20	28.30	24.30	14.50	12.60	10.90
Cumulative % Variance	28.60	53.70	70.60	81.50	91.70	28.30	52.60	67.10	79.70	90.60

Table 3.11: Varimax Rotations for RTS_1 and RTS_2

	Dimension				
	1	2	3	4	5
Eigenvalues	0.999995	0.998048	0.991807	0.925858	0.859853
Critical Angles	0.13	2.53	5.19	15.80	21.98

Table 3.12: Critical Angles Between RTS_1 and RTS_2

3. as it entered integration test,

4. as it entered maintenance, and

5. in its current form.

Table 3.14 describes this selection of TS versions. To characterize the size of these product versions, this table includes the number of executable statements and the number of functions comprising each version. We represent each TS version with four principal components. For each version, these four principal components account for about 93% of the source-code-measure variance.

We compared the loading pattern of TS_1 with that of each of the remaining four TS versions. As expected the differences grew in the comparison with each successive version. Table 3.15 gives the varimax rotation of the loading patterns for TS_1 and TS_5. The loadings vary only slightly across these loading patterns.

Table 3.16 gives the critical angle comparison of TS_1 and TS_5 principal components. Table 3.17 gives the loading patterns corresponding to these critical angles.

Measure	Principal Component									
	RTS_1					RTS_2				
	1	2	3	4	5	1	2	3	4	5
XQT	**0.99**	0.54	0.34	-0.05	0.22	**0.98**	0.53	0.28	-0.10	0.18
N_1	**0.93**	0.23	0.53	-0.09	0.07	**0.93**	0.24	0.44	-0.12	0.03
N_2	**0.81**	0.02	0.60	-0.11	-0.13	**0.82**	0.03	0.54	-0.10	-0.17
$V_1(G)$	**0.73**	0.62	0.09	0.07	**0.70**	**0.70**	0.56	-0.02	-0.03	**0.67**
η_1	0.60	**0.85**	-0.42	0.31	0.49	0.57	**0.79**	-0.54	0.25	0.42
η_2	0.63	**0.76**	-0.10	0.19	0.04	0.54	**0.63**	-0.03	0.13	-0.06
AICC	0.39	**0.76**	**-0.60**	0.33	0.56	0.42	**0.67**	**-0.64**	0.24	0.52
FFOT	0.41	**0.71**	0.25	0.58	0.39	0.34	**0.69**	0.29	0.49	0.12
FFIN	0.29	0.49	-0.19	**-0.55**	-0.35	0.37	**0.64**	-0.10	-0.50	-0.29
Knots	0.56	0.54	0.30	-0.29	**0.65**	0.49	0.45	0.31	**-0.53**	**0.59**

Table 3.13: Critical Angle Rotations for RTS_1 and RTS_2

Product	XQT	Functions
TS_1	13,333	773
TS_2	13,445	766
TS_3	13,902	780
TS_4	14,086	754
TS_5	14,537	703

Table 3.14: The TS Product Versions

As shown in Table 3.16, the loading patterns for these versions are essentially identical with three critical angles less than three degrees, and the remaining angle about nine degrees. Thus it is clear that the loading pattern remained stable throughout the development of TS.

3.2.5 Conclusions

In this chapter, we applied an analytic method to determine the stability of source-code-measure principal components in several settings. We found considerable variability in loading patterns. The greatest principal component variability fell across distinct products developed by distinct organizations. Considerably less principal component variability fell across distinct products developed by the same organization, and still less fell across versions of the same product. These findings indicate that product loading patterns vary, possibly due to differences in the nature of products or the nature of development organizations.

Clearly, the principal components of source code measures are not stable

Measure	Principal Component							
	TS_1				TS_5			
	1	2	3	4	1	2	3	4
FFOT	**0.95**	0.03	0.07	-0.03	**0.95**	0.04	0.09	-0.02
XQT	**0.95**	0.21	0.14	0.14	**0.95**	0.21	0.14	0.13
Edges	**0.93**	0.25	0.07	0.06	**0.93**	0.26	0.05	0.07
N_1	**0.92**	0.24	0.22	0.13	**0.93**	0.23	0.20	0.13
Nodes	**0.91**	0.32	0.12	0.10	**0.92**	0.31	0.10	0.09
N_2	**0.91**	0.22	0.25	0.14	**0.92**	0.21	0.23	0.11
$V_2(G)$	**0.90**	0.19	0.02	0.02	**0.90**	0.22	0.01	0.03
GF	**0.85**	0.33	-0.05	-0.05	**0.88**	0.23	-0.01	-0.06
η_2	**0.72**	0.43	0.35	0.14	**0.75**	0.41	0.36	0.13
Band	0.36	**0.88**	0.13	0.05	0.34	**0.89**	0.08	0.04
η_1	0.27	**0.73**	**0.53**	0.06	0.26	**0.75**	**0.50**	0.09
Data	0.09	0.21	**0.94**	0.11	0.11	0.19	**0.94**	0.10
FFIN	0.10	0.06	0.11	**0.99**	0.09	0.07	0.10	**0.99**
Eigenvalue	7.46	2.00	1.48	1.08	7.62	1.97	1.42	1.08
% Variance	57.38	15.38	11.38	8.31	58.62	15.15	10.92	8.31
Cumulative % Variance	57.38	72.77	84.42	92.46	58.62	73.77	84.69	93.00

Table 3.15: Varimax Rotations for TS_1 and TS_5

	Dimension			
	1	2	3	4
Eigenvalues	0.999961	0.999494	0.998368	0.976994
Critical Angles	0.36	1.29	2.32	8.72

Table 3.16: Critical Angles Between TS_1 and TS_5

	Principal Component							
	TS_1				TS_5			
Measure	1	2	3	4	1	2	3	4
XQT	**0.98**	0.18	0.17	0.12	**0.98**	0.18	0.16	0.14
Nodes	**0.98**	0.11	0.15	0.00	**0.98**	0.13	0.14	0.05
N_1	**0.98**	0.10	0.21	0.13	**0.98**	0.12	0.20	0.15
N_2	**0.96**	0.09	0.23	0.16	**0.97**	0.11	0.19	0.18
Edges	**0.95**	0.17	0.08	0.04	**0.96**	0.18	0.08	0.08
$V_1(G)$	**0.90**	0.21	0.01	0.06	**0.91**	0.21	0.02	0.10
η_2	**0.89**	-0.16	0.36	-0.01	**0.91**	-0.14	0.35	0.01
FFOT	**0.89**	0.26	-0.05	0.25	**0.90**	0.25	-0.04	0.29
GF	**0.88**	0.15	-0.05	-0.03	**0.88**	0.17	-0.06	0.03
BW	**0.67**	-0.39	0.31	**-0.60**	**0.64**	-0.38	0.30	**-0.60**
η_1	**0.62**	**-0.62**	0.52	-0.32	**0.60**	**-0.60**	0.52	-0.31
Data	0.36	**-0.68**	**0.66**	0.26	0.36	**-0.66**	**0.63**	0.32
FFIN	0.21	0.30	**0.84**	-0.07	0.20	0.30	**0.85**	-0.09

Table 3.17: Critical Angle Rotations for TS_1 and TS_5

across all software products. That is, one can not expect to see the same loading patterns across all software products and all development organizations. However, as we saw, loading pattern stability is possible throughout the development history of a given product, and across similar products developed by the same organization. Thus, principal components based regression models can make sense throughout the lifecycle of a product, *and* across similar products produced by the same organization. Still, when modeling current product development with a model fitted to data collected from past product development, one must take care to ensure that the current product is similar to the past product. Large critical angles between the principal components of these products serve to question the choice of models.

For a given product, each of the principal components identify a source of variation underlying a collection of source code measures. Consider, for example, a principal component that associates strongly with measures related to control flow complexity. Variations in this principal component are variations in control flow complexity. The complexity difference across program modules is an interplay of several such sources of complexity. In this section, we found that different products can produce vastly different loading patterns. The following section focuses on understanding the impacts of these differences on software quality.

3.3 The Impact on Software Quality Models

In this section, we investigate the effects of product similarity, and the lack of it, on regression modeling results. We collected source code attribute data and

failure report data from historical data retained during the development of seven commercial real-time software products. Both failure reports and source code measures quantify characteristics of the files that compose the products. The source code measures represent these files as they entered the system test phase, and failure reports represent failures that occurred during this phase. The products are assembly language implementations. Thus, the source code measures are those identified for assembly language in Table 3.1.

From the collection of seven products, we selected two product pairs: the pair having the most similarity and the pair having the least similarity. One product is common to both pairs. We quantified similarity analytically. Section 3.3.2 describes the seven products under study, and the selection of the two pairs showing the most and least similarity.

After selecting the two product pairs, we developed a multiple regression model using data collected from the product common to both pairs. Section 3.3.1 gives some regression modeling terminology. Model selection took independent variables from selections of principal components. The dependent variable was the number of failure reports. The data for the second product in each pair served to test the predictive quality of the model. Section 3.3.3 reports the results of this modeling analysis.

3.3.1 An Overview of Multiple Regression Analysis

A multiple linear regression model expresses the linear relationship between a set of variables and another variable. Typically, one employs a linear regression model to predict future values of one variable based upon known values of the others. The predicted and known variables are called, respectively, the dependent and independent variables.

Let x_{ij}, $1 \leq i \leq n$, $1 \leq j \leq p$ represent the values of p independent variables for each of n observations. A model selection technique selects a subset of $p^* \leq p$ independent variables that contribute significantly to the model. Several techniques are available for selecting this subset. These include the $R^2_{p^*}$ criterion, C_{p^*} criterion, forward selection, stepwise regression, and backward elimination [43].

Using the $R^2_{p^*}$ criterion and C_{p^*} criterion, the researcher considers all regression models formed by combinations of p^* independent variables, $1 \leq p^* \leq p$. $R^2_{p^*} = \frac{SSR_{p^*}}{SST}$, where SSR_{p^*} is the regression sum of squares for a model that includes p^* independent variables, and SST is the total sum of squares. With the addition of independent variables, $R^2_{p^*}$ will rise quickly toward unity until some cutoff point at which it will begin to rise slowly. Using the $R^2_{p^*}$ criterion, the researcher takes the model with greatest $R^2_{p^*}$ near the cutoff point. C_{p^*} measures the total squared error of a regression model with p^* independent variables. Using the C_{p^*} criterion, the researcher takes a model with C_{p^*} nearly equal to p^*.

Stepwise regression, forward selection, and backward elimination are iterative

procedures. In stepwise regression the initial model uses the independent variable having the largest squared simple correlation with the dependent variable. In each subsequent iteration, the procedure selects new variables for inclusion based on their partial correlation with variables already in the regression equation. After including a new variable, the procedure removes any variable that no longer contributes significantly to the explained variance. Forward selection acts identically except that it never removes variables from the model. In backward elimination the initial model uses all of the independent variables. In each subsequent iteration, the procedure removes variables that do not contribute significantly to the model. Refer to Myers for further details concerning model selection [51].

Model fitting proceeds with the selected model. For each predicted value of the dependent variable, $\hat{y}_i$, $1 \leq i \leq n$, there is a corresponding actual and residual value, y_i and $\epsilon_i = y_i - \hat{y}_i$, respectively. The total deviation of the y_i values about the mean, $\bar{y} = \frac{1}{n}\sum_{i=1}^{n} y_i$, is expressed as a sum of squares,

$$\sum_{i=1}^{n}(y_i - \bar{y})^2 = \sum_{i=1}^{n}(\hat{y}_i - \bar{y})^2 + \sum_{i=1}^{n}\epsilon_i^2,$$

where these sums are the total sum of squares (SST), the regression sum of squares (SSR), and the error sum of squares (SSE), respectively. The least squares estimation technique yields estimated model parameters, $\hat{\beta}_0, \hat{\beta}_1, \ldots, \hat{\beta}_{p^*}$, such that

$$\hat{y}_i = \hat{\beta}_0 + \hat{\beta}_1 x_{i1} + \hat{\beta}_2 x_{i2} + \cdots + \hat{\beta}_{p^*} x_{im^*}, \quad 1 \leq i \leq n,$$

and SSE is minimized.

The R^2_{fit} statistic quantifies the model's quality of fit,

$$R^2_{fit} = \frac{\text{SSR}}{\text{SST}}$$

R^2_{fit} varies from zero to one, with values closer to one indicating a better fit. The PRESS statistic quantifies the model's predictive quality,

$$\text{PRESS} = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

where y_i is defined above, and $\hat{y}_i$ is the predicted value of y_i from a regression equation fitted with all observations except the i^{th}. Lower average PRESS values, $\overline{\text{PRESS}} = \text{PRESS}/n$, indicate better predictive quality. PRESS and SST yield an R^2-like statistic reflecting predictive quality,

$$R^2_{pred} = 1 - \frac{\text{PRESS}}{\text{SST}}.$$

R^2_{pred} varies from zero to one, with values closer to one indicating a better predictive quality. The average absolute error of a model quantifies its predictive performance on a given data set. The average absolute error is given by

$$\text{AAE} = \frac{1}{N} \sum_{i=1}^{N} |y_i - \hat{y}_i|.$$

3.3.2 The Selection of Software Products

This section reports the results of a similarity study on a set of software products implemented in assembly language. Table 3.18 describes the assembly products available for this analysis. To characterize the size of these products, this table includes the number of executable statements and the number of files

comprising each product. The seven products shown in Table 3.18 are distinct Real-Time Systems ($RTS_1, RTS_2, \ldots, RTS_7$). The developers implemented these products in assembler to satisfy space and time constraints. All products are commercial real-time systems developed to run on the same 32-bit microprocessor.

We represent these assembly language software products with six principal components. For each product, these six principal components account for over 96% of the source-code-measure variance. Note that there is no universally accepted method for determining the number of principal components with which to represent a data set [43]. While a number of "rules" have been proposed, the decision rests largely on judgement and personal taste. We apply a method for quantifying the similarity of two sets of principal components after this decision is made. While the similarity results could provide feedback useful in evaluating the selection of components, we do not apply the method in this way in this study.

We compare sets of principal components using the *critical angle* method presented in Section 3.1. We found the critical angles for all 21 pairwise comparisons of the seven products. Table 3.19 summarizes these pairwise critical angle comparisons. It is interesting that we found the critical angle variation shown in this table, given that these products provide similar function, run on similar platforms, and were developed by the same organization using the same process. We selected two product pairs for further study: (RTS_5, RTS_6) and (RTS_5, RTS_4). As

Product	XQT	Files
RTS_1	267,481	172
RTS_2	159,374	122
RTS_3	180,774	109
RTS_4	71,912	57
RTS_5	163,704	104
RTS_6	135,350	99
RTS_7	222,740	152

Table 3.18: The Assembly Language Products

shown in Table 3.19, these pairs have respectively, the greatest and the least principal components similarity. After discussing these pairs with a product manager, we found that they differed in target markets, RTS_5 and RTS_6 satisfying a high-cost, high-quality demand, and RTS_4 satisfying a low-cost, high-volume demand.

Tables 3.20, 3.21, and 3.22 give the varimax principal components rotations for, respectively, RTS_5, RTS_6, and RTS_4. In the varimax rotation for RTS_5, note that the principal components fall in nonincreasing order of explained variance, and that the measures fall in nonincreasing order of loadings on the dominating component. However, in the varimax rotations for RTS_6 and RTS_4 the principal components fall in an order such that the loading patterns for each component most closely match those of the corresponding component for RTS_5, and the measures fall in the same order as in the RTS_5 rotation. This eases the comparison of the two

Pair	Critical Angle						Total
	1	2	3	4	5	6	
(RTS_5, RTS_6)	0.00	0.01	0.64	3.74	6.24	9.09	19.72
(RTS_3, RTS_6)	0.00	0.02	0.55	2.14	2.83	17.22	22.76
(RTS_3, RTS_7)	0.01	0.03	0.54	1.36	7.36	13.89	23.19
(RTS_2, RTS_3)	0.00	0.01	1.33	3.29	3.82	16.79	25.25
(RTS_6, RTS_7)	0.00	0.00	0.75	2.76	8.79	14.23	26.53
(RTS_2, RTS_6)	0.00	0.04	2.20	3.94	5.74	18.68	30.61
(RTS_5, RTS_7)	0.00	0.01	0.57	4.48	5.47	20.11	30.65
(RTS_1, RTS_2)	0.00	0.05	1.29	4.04	5.47	22.13	32.98
(RTS_2, RTS_7)	0.00	0.02	1.53	4.55	7.77	19.65	33.53
(RTS_3, RTS_5)	0.00	0.02	1.04	2.51	5.78	24.28	33.63
(RTS_2, RTS_5)	0.00	0.01	0.72	2.95	7.86	25.60	37.14
(RTS_1, RTS_3)	0.03	0.04	2.09	3.85	8.12	24.05	38.18
(RTS_1, RTS_7)	0.03	0.05	1.43	2.53	5.55	31.83	41.41
(RTS_1, RTS_6)	0.03	0.03	2.94	5.99	9.85	36.25	55.09
(RTS_3, RTS_4)	0.00	0.02	0.66	3.35	8.75	42.32	55.10
(RTS_1, RTS_4)	0.00	0.02	1.02	4.86	11.37	38.38	55.65
(RTS_4, RTS_7)	0.00	0.03	0.81	3.90	8.25	46.14	59.13
(RTS_4, RTS_6)	0.00	0.04	0.67	1.27	6.83	50.97	59.77
(RTS_1, RTS_5)	0.00	0.03	3.02	4.29	8.35	44.14	59.84
(RTS_2, RTS_4)	0.00	0.04	1.43	5.29	12.49	46.13	65.37
(RTS_5, RTS_4)	0.00	0.04	0.65	2.70	5.66	57.59	66.64

Table 3.19: The Pairwise Comparison

Measure	Principal Component					
	1	2	3	4	5	6
N_2	**0.98**	0.11	0.03	0.01	0.00	0.03
N_1	**0.98**	0.04	0.17	0.07	0.01	0.07
XQT	**0.84**	0.23	0.31	0.33	0.03	0.14
AICC	-0.11	**0.94**	0.15	0.08	0.08	0.12
η_1	0.13	**0.88**	0.20	0.25	0.07	0.18
KNOTS	0.15	0.14	**0.95**	0.03	-0.02	0.10
V(G)	0.30	**0.52**	**0.68**	0.17	-0.06	0.09
η_2	0.19	0.26	0.09	**0.92**	0.09	0.19
FFIN	0.02	0.09	-0.04	0.07	**0.99**	0.05
FFOT	0.13	0.24	0.13	0.18	-0.06	**0.93**
Eigenvalue	2.83	2.14	1.57	1.09	1.01	1.00
% Variance	28.30	21.40	15.70	10.90	10.10	10.00
Cumulative % Variance	28.30	49.70	65.40	76.30	86.40	96.40

Table 3.20: Varimax Rotation Loadings for RTS_5

loading patterns.

Considerable similarity is evident between the varimax rotations of RTS_5 and RTS_6. The pattern of dominating loadings remains constant across these two products, and all loadings remain similar. Similarity is also evident in the varimax rotations of RTS_5 and RTS_4. There is similarity in the pattern of dominating loadings for the first, fifth, and sixth principal components. However, differences show in

Measure	Principal Component					
	1	2	3	4	5	6
N_2	**0.98**	-0.10	0.05	0.05	0.03	0.04
N_1	**0.97**	0.08	0.20	0.07	0.08	0.08
XQT	**0.81**	0.29	0.34	0.29	0.16	0.12
AICC	0.07	**0.94**	0.14	0.14	0.07	0.14
η_1	0.13	**0.86**	0.19	0.25	0.15	0.25
KNOTS	0.19	0.15	**0.92**	0.11	0.06	0.13
V(G)	0.35	**0.54**	**0.63**	0.10	0.03	0.12
η_2	0.21	0.34	0.16	**0.87**	0.17	0.16
FFIN	0.12	0.12	0.04	0.12	**0.98**	0.01
FFOT	0.13	0.29	0.16	0.14	0.01	**0.93**
Eigenvalue	2.82	2.26	1.51	0.99	1.04	1.02
% Variance	28.20	22.60	15.10	9.90	10.40	10.20
Cumulative % Variance	28.20	50.80	65.90	75.8	86.20	96.40

Table 3.21: Varimax Rotation Loadings for RTS_6

Measure	Principal Component					
	1	2	3	4	5	6
N_2	**0.98**	-0.18	-0.02	-0.04	-0.02	-0.01
N_1	**0.99**	-0.04	-0.07	0.06	0.02	0.04
XQT	**0.88**	0.28	0.24	0.16	0.12	0.16
AICC	0.15	**0.93**	0.11	0.21	0.09	0.10
η_1	0.01	**0.92**	0.22	0.14	0.13	0.19
KNOTS	0.12	0.29	**0.91**	0.18	-0.04	0.02
$V(G)$	0.20	0.43	0.47	**0.69**	0.05	0.20
η_2	0.18	**0.72**	0.42	-0.10	0.23	0.33
FFIN	0.05	0.19	0.02	0.02	**0.98**	0.01
FFOT	0.09	0.27	0.04	0.10	0.01	**0.95**
Eigenvalue	2.84	2.71	1.35	0.63	1.05	1.12
% Variance	28.40	27.10	13.50	6.30	10.50	11.20
Cumulative % Variance	28.40	55.50	69.00	75.30	85.80	97.00

Table 3.22: Varimax Rotation Loadings for RTS_4

the second, third, and fourth principal component, the fourth principal component showing little similarity.

The loadings given in Tables 3.20, 3.21, and 3.22 demonstrate the utility of varimax rotation. It is clear from these tables that the three products have mutual components dominated by

- N_2, N_1, and *XQT*,
- *FFIN*, and
- *FFOT*.

RTS_5 and RTS_6 have three more mutual components dominated by

- *AICC*, η_1, and $V(G)$,
- *Knots* and $V(G)$, and
- η_2.

The first two of these are shared to some extent with RTS_4, although with different loadings on $V(G)$ and η_2. The component dominated by η_2 in RTS_5 and RTS_6 has little in common with the remaining component of RTS_4. Thus, differences between RTS_5 and RTS_4 are established by differences in the associations of $V(G)$ and η_2 across the second, third, and fourth principal components. This interpretation seems to agree with the critical angles given in Table 3.19. The product pair (RTS_5, RTS_6) has small critical angles along all six comparisons, while both

of the product pairs (RTS_4, RTS_5) and (RTS_4, RTS_6) have three near-zero critical angles, two small critical angles, and one large critical angle.

While interpretation of the varimax rotation seems to agree with the critical angle analysis, it is important to note that the critical angles do not fall between pairs of principal components from these rotations. The critical angles given for each pairwise comparison in Table 3.19 fall between pairs of principal components from rotations selected to achieve the smallest angles between paired components, that is, between pairs of critical angle principal component rotations. Table 3.23 gives the loadings for the critical angle principal component rotations of RTS_5 and RTS_6. For example, the first critical angle shown for RTS_5 and RTS_6 in Table 3.19 falls between the patterns shown for the first principal component in Table 3.23. Table 3.24 gives the loadings for the critical angle principal component rotations of RTS_5 and RTS_4. The critical angle rotations aid in the interpretation of similarities and differences between the sets of compared principal components.

The critical angle rotations for RTS_5 and RTS_6 shown in Table 3.23 display similarity just as the varimax rotations for these products do. Table 3.24 gives the critical angle rotations for RTS_5 and RTS_4. The first five principal component pairs shown in this table display strong similarity, although a pattern of difference in the loadings of η_2 becomes apparent in the third, fourth, and fifth components. η_1 also displays difference across the fourth and fifth components. The sixth principal component pair displays a strong difference in the loadings on $V(G)$.

Measure	Principal Component											
	RTS_5						RTS_6					
	1	2	3	4	5	6	1	2	3	4	5	6
XQT	**0.88**	**-0.62**	**0.50**	0.19	**0.50**	0.32	**0.92**	**-0.61**	**0.55**	0.04	0.32	0.35
N_2	**0.61**	**-0.82**	0.05	0.44	**0.54**	0.06	**0.62**	**-0.82**	0.09	0.39	0.46	0.08
N_1	**0.74**	**-0.79**	0.25	0.34	**0.55**	0.10	**0.77**	**-0.78**	0.30	0.26	0.43	0.12
$V(G)$	**0.63**	-0.26	**0.85**	-0.29	0.30	0.13	**0.67**	-0.28	**0.83**	-0.28	0.22	0.23
Knots	0.34	-0.43	**0.77**	-0.27	0.07	0.16	0.45	-0.44	**0.77**	-0.26	-0.03	0.28
η_1	**0.72**	0.26	**0.74**	-0.35	0.18	0.11	**0.76**	0.24	**0.77**	-0.36	0.08	0.19
AICC	**0.51**	0.47	**0.66**	**-0.50**	0.05	-0.07	**0.57**	0.43	**0.71**	-0.48	0.06	0.05
FFOT	0.48	0.13	**0.64**	**0.50**	0.01	0.19	**0.52**	0.12	**0.70**	0.43	-0.11	0.14
FFIN	0.38	0.05	-0.19	-0.36	**-0.70**	0.02	**0.54**	-0.08	-0.01	-0.34	**-0.67**	0.19
η_2	**0.67**	0.07	0.44	0.03	0.15	**0.86**	**0.73**	0.02	**0.52**	-0.13	0.05	**0.85**

Table 3.23: Critical Angle Rotation Loadings for RTS_5 and RTS_6

Measure	Principal Component											
	RTS_6						RTS_4					
	1	2	3	4	5	6	1	2	3	4	5	6
XQT	**0.94**	**0.52**	0.27	0.04	**0.74**	-0.06	**0.93**	**0.59**	0.29	0.17	**0.75**	-0.02
FFOT	0.24	**0.87**	0.20	0.17	-0.02	0.23	0.30	**0.88**	0.16	0.25	0.06	0.16
η_2	0.45	**0.81**	**0.75**	0.26	0.07	-0.47	0.42	**0.73**	0.49	0.08	0.24	-0.46
η_1	0.25	**0.80**	**0.89**	0.42	0.04	-0.05	0.29	**0.77**	**0.82**	0.43	0.32	0.08
AICC	0.06	**0.71**	**0.88**	0.45	-0.05	0.06	0.03	**0.62**	**0.84**	0.45	0.14	0.17
V(G)	**0.59**	**0.61**	**0.65**	**0.56**	0.07	0.06	**0.66**	**0.57**	**0.62**	**0.62**	0.27	0.43
Knots	**0.59**	0.38	**0.54**	**0.61**	-0.14	0.00	**0.65**	0.31	0.41	**0.59**	-0.07	0.08
FFIN	0.16	0.21	**0.55**	**-0.61**	-0.01	0.13	0.07	0.04	**0.50**	**-0.68**	-0.04	0.13
N_1	**0.87**	0.19	-0.12	-0.16	**0.88**	0.04	**0.90**	0.34	0.01	0.01	**0.85**	0.03
N_2	**0.78**	0.03	-0.30	-0.26	**0.88**	0.04	**0.81**	0.19	-0.18	-0.13	**0.84**	-0.05

Table 3.24: Critical Angle Rotation Loadings for RTS_6 and RTS_4

The differences concentrate on the relationships of η_1, η_2, and $V(G)$ across the fourth, fifth, and sixth principal components. The fifth component in RTS_5 has strong loadings on only N_1, N_2, and XQT. In RTS_4, this component retains these loadings and picks up weak loadings on η_1, η_2, and $V(G)$. This removes much of what distinguishes the first and fifth components of RTS_5. Where RTS_5 has a component loading exclusively on measures related to size, RTS_4 has a component loading on η_1, η_2, and $V(G)$ as well as size.

We note that in several RTS_5 components, η_2 and $V(G)$ share similar loadings. The sixth component is an exception having a moderate negative loading on η_2 and no loading on $V(G)$. In the sixth component of RTS_4, this relationship is exaggerated with the addition of a moderate positive loading on $V(G)$. The nature of the fourth component changes in a similar way, losing much of its loading on η_2 while gaining loading on $V(G)$. Thus in RTS_4, η_2 and $V(G)$ have components showing opposing association and independence. This is the greatest contrast with between RTS_5 and RTS_4.

3.3.3 The Modeling Results

The two product pairs, (RTS_5, RTS_6) and (RTS_5, RTS_4), provide data for assessing the impact of principal components instability in models based upon principal components. Source code attribute and quality data collected from RTS_5 serve to fit a model for predicting quality data for RTS_6 and RTS_4. The model predicts based upon the principal components of RTS_5.

Parameter	Degrees of Freedom	Est.	Std. Error	T for H_0: Parameter $= 0$	Prob. $> \|T\|$
Intercept	1	23.94	0.6651	36.00	0.0001
Principal Component 2	1	3.94	0.6812	5.781	0.0001
Principal Component 3	1	1.50	0.6861	2.180	0.0317
Principal Component 4	1	5.81	0.6685	8.689	0.0001
Principal Component 6	1	3.13	0.6658	4.699	0.0001

Table 3.25: The RTS_5 Model

Table 3.25 gives the RTS_5 model. The four principal components identified by model selection as significant in this model are the second, third, fourth, and sixth principal components from the varimax principal components rotation given for RTS_5 in Table 3.20. We observed in Section 3.3.2 that the second, third, and fourth principal components of this rotation showed the greatest differences. Table 3.26 gives the significant principal components. The model fitted with these linear combinations of the standardized RTS_5 measure data yielded $R^2_{fit} = 0.59$ and $R^2_{pred} = 0.53$, values indicating satisfactory model fit and predictive quality.

Critical angle analysis identified the principal components of product pair (RTS_5, RTS_6) as similar, and the principal components of product pair (RTS_5, RTS_4) as dissimilar. Since the RTS_5 model is based upon principal components, we expected it to demonstrate greater predictive quality on RTS_6 than on RTS_4. We tested this by applying the RTS_5 model to both RTS_6 and RTS_4. Applying the

Measure	Principal Component			
	2	3	4	6
N_2	0.0031	-0.1690	-0.1791	-0.0005
N_1	0.0449	-0.0881	-0.1565	-0.0211
XQT	0.0183	0.0185	0.1581	-0.0692
AICC	0.6354	-0.1884	-0.2439	-0.0923
η_1	0.5265	-0.1782	-0.0520	-0.0798
Knots	-0.2798	0.8595	-0.0468	0.0282
V(G)	0.1218	0.4023	-0.0104	-0.1466
η_2	-0.1860	-0.0330	1.1331	-0.1563
FFIN	-0.0813	0.0707	-0.0914	0.0909
FFOT	-0.1416	-0.0457	-0.1939	1.1575

Table 3.26: The Significant Principal Components

model to the standardized RTS_6 measure data yielded an Average Absolute Error (AAE) of about 6.9 with a standard deviation of about 5.19. Applying the model to the standardized RTS_4 measure data yielded AAE = 21.9 with a standard deviation of about 7.26. Thus, the model demonstrated better predictive quality when applied to the product having principal components more closely matching those used for fitting the model.

3.3.4 Conclusions

Software quality models provide predictions that can help engineers to estimate costs, set realistic schedules, and allocate resources. Thus, development organizations that apply these models can hold a competitive advantage over those that do not. Further, those that develop models having the greatest predictive quality have the greatest advantage. This has prompted research effort toward improving

software quality models.

Still, we have seen no analytical tools to help engineers select appropriate software quality models. Product function, schedule pressure, process, and people affect the product. To some extent, these represent sources of variation reflected by source code measures, and thus, differences in software quality models. In Section 3.2, we saw wide variation in the principal components underlying source code measures. In this section, we presented evidence suggesting that this variation can affect the predictive quality of software models. We temper this conclusion with the knowledge that sources of variation other than those reflected by source code measures can affect the predictive quality of software quality models. Still, we considered products developed by the same organization using the same process, and thus controlled many influences that might not be reflected by source code measures. Further, we noted that the three least similar principal components of the dissimilar product pair were among the four principal components included by model selection. Thus, the evidence presented strongly supports our intuition: that a model with good quality of fit is likely to have poor predictive quality if the selected principal components vary substantially from those of the product to which it is applied.

This result has two important implications. First, the critical angle method that we employed to isolate pairs of similar and dissimilar products can serve software engineers as a tool for selecting an appropriate model for a given product. This

method carries no distributional assumptions, and thus is not weakened by the lack of multivariate normality often found among software engineering measures.

Second, since the critical angle method indicates the degree of similarity between *critical angle rotations* of principal components, the power of this method increases when models are selected from these principal components rather than from the principal components of a varimax rotation. For software quality modeling, the similarity of the components included by model selection is important; the similarity of the components excluded by model selection is not. Thus, a high critical angle sum does not imply that the model is inappropriate; a high critical angle sum among components included by model selection does. Since the critical angles do not map to components of the varimax rotation, the accuracy of the method is not enhanced by knowledge of the selected components for models selected from these principal components. For models selected from the principal components of critical angle rotations, this knowledge does enhance the accuracy of the method. We intend to pursue this point with further research.

Chapter 4

CANONICAL MODELING OF SOFTWARE ENGINEERING MEASURES

Increasingly, software product suppliers recognize that software development process capability is a key source of competitive advantage [52]. Competition forces suppliers to improve processes to meet the opposing demands of higher quality, lower costs, and compressed schedules. Further, software purchasers are beginning to require certification that suppliers apply development processes capable of delivering products within quality, cost, and schedule constraints. Measures of source code attributes and of process activity provide for quantitative analyses of existing processes. Such analyses lay the foundation for continuous process improvement.

To model and analyze the software development process one must consider the relationships between many measures. Often these measures fall in two sets. Software engineers are interested in the causal relationship between these sets. For example, knowing the influence of a set of measures extracted from a software product implementation on a resulting set of measures quantifying activity during the system test phase, software engineers can make more informed staffing and training decisions in preparation for system testing. A similar benefit comes with knowledge

of the influence of a set of measures extracted from a software product design upon a set of measures extracted from the resulting software product implementation.

Researchers have explored several approaches to modeling causal relationships among sets of software engineering measures. These approaches include optimized set reduction [17], classification trees [23], neural networks [21], linear discriminant analysis [5], and linear regression analysis [25, 53]. Each of these approaches have yielded promising results, and each warrants further research, extension, and refinement. In this chapter, we consider the linear regression approach to modeling causal relationships among sets of software engineering measures.

A simple linear regression models the linear relationship between a single predictor variable and a single response variable. Henry and Selig applied simple linear regressions to model the linear relationships between measures extracted from a software product detailed design and measures extracted from the resulting software product implementation [53]. Since both detailed designs and implementations yield sets of measures with interesting relationships, [53] formed many simple regressions, each addressing the relationship between a unique pair of measures. For one of these pairs, the measures were aggregates of the measures from each of the sets. Henry and Selig concluded that the aggregate measures provided more consistent results than any of the constituent measures taken alone. Others have reported similar conclusions when using simple regressions to model relationships between software engineering measures [54, 55].

A multiple linear regression models the linear relationship between a set of predictor variables and a single response variable. Khoshgoftaar and Munson applied multiple linear regression to model the linear relationship between measures extracted from a software product implementation and a resulting process measure quantifying activity during the system test phase [25]. Principal components analysis dispensed with multicolinearity among the source code measures. Although multiple regression produces a linear combination of predictor variables, this aggregate measure differs from those employed in the cited applications of simple linear regression. In these applications, the structure of the aggregate measure represents either the interaction among the source code measures [54], or a theoretical association assumed to hold in general [55, 53]. The aggregate produced by a multiple regression represents the interaction among the source code measures, and the interaction between the source code measures and a process measure.

A canonical correlation models the linear relationship between a set of predictor variables and a set of response variables. In a model of the relationship between source code and process measures, several interactions among measures are important: the interactions among the set of source code measures, the interactions among the set of process measures, and the interactions between these sets of measures [56]. In considering each of these interactions, canonical correlation analysis produces a model that is useful in both ways identified by Neil and Bache [57]. First, interpretation of the canonical correlation model can help software engineers

understand attributes of their products, and the relationship of these attributes with software process activity. Second, a canonical correlation model yields two sets of aggregate measures: a set of source code measure aggregates and a set of process measure aggregates. Each source code measure aggregate has maximal correlation with a corresponding process measure aggregate. The source-code-measure aggregates are useful for predicting future process activity. Typically, a small set of these aggregate measure pairs will explain the relationship between larger sets of source code and process measures. Thus, canonical correlation analysis also serves as a data reduction technique.

In this chapter, we describe three canonical correlation models of the relationship between a set of source code measures and a set of process measures. Source code and process measures collected during the development of a commercial real-time product provide the data for these analyses. To this point, our development of canonical correlation analysis has taken the perspective that it is a generalization of simple and multiple linear regression modeling. Since Hotelling presented this generalization [58], further research has identified canonical correlation analysis as itself a special case of a more general modeling technique [59, 60]. Section 4.1 takes this perspective in giving an overview of canonical correlation analysis. By offering both perspectives, we demonstrate how application of this modeling technique both builds upon past software engineering modeling efforts, and provides a platform for future efforts. Section 4.2 describes the three canonical models developed in this

study.

4.1 An Overview of Canonical Correlation Analysis

Canonical correlation analysis is a restricted form of the soft modeling methodology introduced by Wold [29]. A soft model involves both *manifest* and *latent* variables, respectively, variables that are directly observed, and variables that are indirectly observed. A weighted aggregate of manifest variables quantifies each latent variable. For example, a social model might quantify "population change" in terms of the observable variables "natality", "mortality", and "migration". The weighted aggregates quantifying the latent variables give the *outer relations* of the model. *Inner relations* specify a causal relationship among the latent variables. Latent variables explained by an inner relation are *endogenous*; those that are not are *exogenous*. Correspondingly, the manifest variables are either endogenous or exogenous. Canonical correlation analysis applies a soft model restricted to one latent exogenous variable, one latent endogenous variable, one inner relation, and to linear relationships among variables.

A *path diagram* gives a visual soft model specification. In a path diagram, manifest variables appear as blocks, latent variables appear as circles, and relations appear as directed paths. The path diagram in Figure 4.1 gives the model for the first canonical correlation. In this model, ξ and ζ are latent variables indicated by blocks of manifest variables, respectively, $\mathbf{x} = [x_1, x_2, \ldots, x_n]$ and $\mathbf{y} = [y_1, y_2, \ldots, y_m]$. The path from ξ to ζ gives the inner relation. The direction of this path specifies that ξ

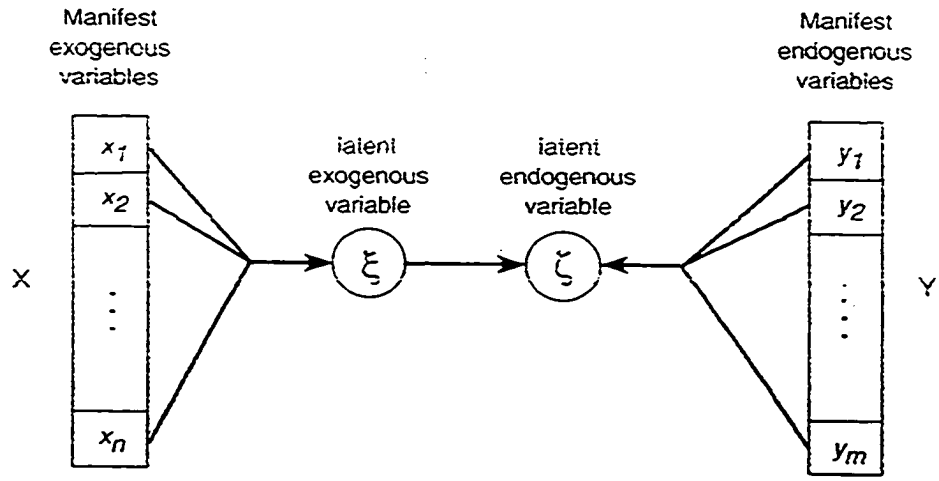

Figure 4.1: The First Canonical Correlation

explains ζ. Thus, ζ is endogenous. A weight associated with the inner relation is a model parameter giving the influence of ξ on ζ. Having no paths in from other latent variables, ξ is exogenous; its cause is outside of the model. The paths from x to ξ and from y to ζ represent the outer relations. Weights associated with each of these paths are model parameters giving the coefficient of each manifest variable in the aggregate quantifying the related latent variable. For the first canonical correlation, the weights defining the outer relations are evaluated such that the latent variables are maximally correlated. The resulting correlation is the *canonical correlation* between the two latent variables.

As explained above, Figure 4.1 gives the model for the first canonical correlation. Where there are n manifest exogenous variables and m manifest endogenous variables a canonical correlation analysis yields $d = \min(n, m)$ dimensions of canonical correlation. Figure 4.2 shows this graphically. Superscripts identify the dimensions of the latent variables. For example, ξ is d-dimensional having dimensions $\xi^{(1)}, \xi^{(2)}, \ldots, \xi^{(d)}$. Similarly, ζ is d-dimensional having dimensions $\zeta^{(1)}, \zeta^{(2)}, \ldots, \zeta^{(d)}$. The directed paths from $\xi^{(k)}$ to $\zeta^{(k)}$, $1 \leq k \leq d$, give a d-dimensional inner relation. The weights defining the outer relations are evaluated such that the d dimensions of endogenous latent variables are mutually orthogonal, the d dimensions of exogenous latent variables are mutually orthogonal, and the pairs of latent variables at each dimension are maximally correlated. The correlations between each of these pairs give the d canonical correlations of the canonical model.

Canonical correlation analysis is helpful in understanding the relationships between two sets of variables: the manifest exogenous set, and the manifest endogenous set. The corresponding latent variables elicit the structure between these sets. Since latent variables are not directly observable, they are best interpreted in terms of the manifest variables most related to them. Each manifest variable in the inner relation defining a latent variable will have a correlation, or *loading*, with this latent variable. The pattern of loadings for a latent variable can suggest its nature.

While there are d dimensions of canonical correlation, it is often reasonable to interpret just a few. First, there is no reason to interpret relationships that are

below a reasonable statistical significance. Second, dimensions having low canonical correlations can be ignored. Third, each dimension accounts for some percentage of the total variance explained by the analysis. Those accounting for a small percentage of this variance can be ignored.

4.2 A Canonical Model of Complexity and Defect Activity

With the general background discussed in Section 4.1, it is possible to discuss the specific models of interest in this chapter. In this section, we apply canonical correlation analysis to investigate the relationship between source code complexity and defect activity during the system test phase for a commercial real-time product (RTS). The developers implemented RTS in assembly language to satisfy space and time constraints. The RTS source code consists of 152 files containing 222,740 lines of code. Ten source code measures, evaluated for each of the 152 RTS files *on entry to* the system test phase, provide a cross-section of product data for the three canonical models: η_1, η_2, N_1, N_2, *XQT*, $V_1(G)$, *Knots*, *FFOT*, *FFIN*, and *AICC*.

The following nine process measures, evaluated for each of the 152 RTS files *on exit from* the system test phase, provide a cross-section of process data for the canonical models: D, D_1, D_2, D_3, D_4, D_c, A, R, and M. Unique selections of these measures serve to indicate defect activity for each model. This data set did not include a tally for source lines changed. Changed lines recorded one line added and one line deleted.

To remove variance due to differences in units of measure, we normalize each

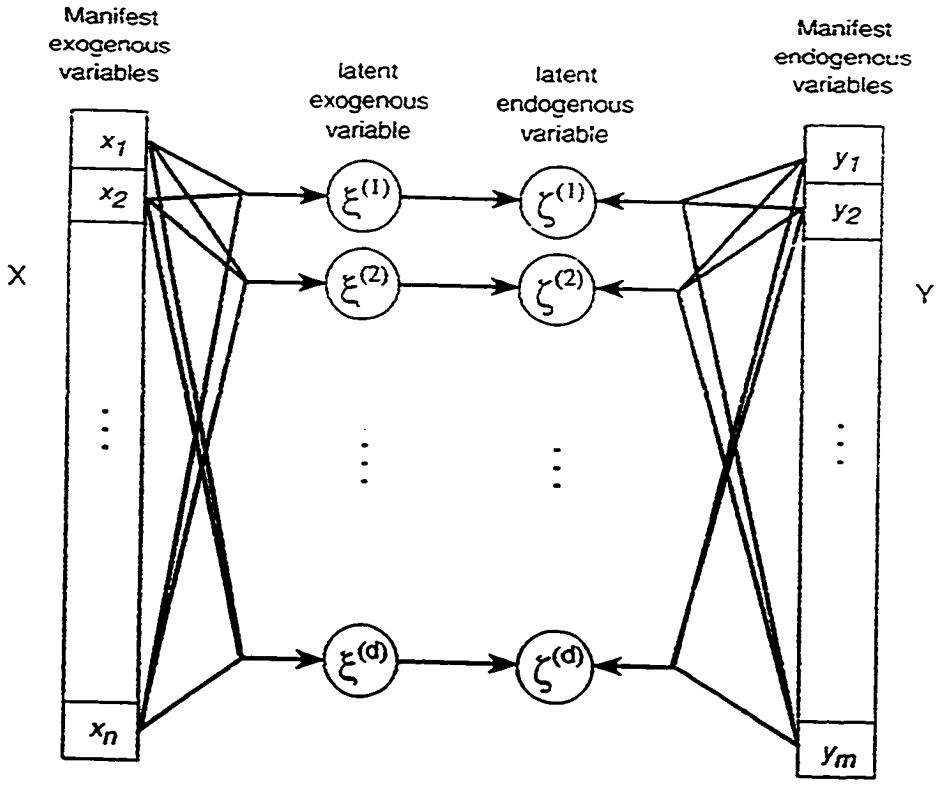

Figure 4.2: The General Canonical Model

Product Measures	Process Measures								
	A	R	M	D	D_c	D_1	D_2	D_3	D_4
η_1	0.21	0.21	0.17	0.35	0.22	0.11	0.39	0.29	0.29
η_2	0.80	0.80	0.70	0.56	0.77	-0.01	0.62	0.35	0.64
N_1	0.15	0.14	0.10	0.26	0.13	0.15	0.27	0.24	0.18
N_2	0.17	0.17	0.13	0.17	0.14	0.08	0.18	0.15	0.14
XQT	0.36	0.36	0.27	0.46	0.35	0.18	0.49	0.39	0.37
$V(G)$	0.03	0.03	-0.02	0.34	0.04	0.39	0.35	0.34	0.15
Knots	0.15	0.15	0.07	0.17	0.22	0.09	0.20	0.15	0.08
FFOT	0.23	0.24	0.14	0.31	0.24	0.05	0.34	0.21	0.34
FFIN	0.21	0.22	0.20	0.25	0.21	-0.06	0.24	0.20	0.27
AICC	0.02	0.01	0.00	0.27	0.05	0.15	0.28	0.26	0.17

Table 4.1: Correlations Between the Product and Process Measures

variable to zero mean and unit variance. Table 4.1 gives the correlations between the product and the process measures.

Note that different selections of measures could be interesting for each unique software development process. Our goal in this section is to apply canonical correlation analysis to investigate the relationship between source code and process measures, not to justify the use of any particular selection of measures. For details regarding the selection and validation of software engineering measures refer to [38, 39].

We develop three canonical models from this data set. Each model measures source code complexity as a latent exogenous variable indicated by all ten source code measures. While each model measures defect activity as a latent endogenous variable, the selection of process measures indicating defect activity differentiates the models. Section 4.2.1 describes model $M_{4.1}$, a model indicating defect activity by D and D_c. Section 4.2.2 describes model $M_{4.2}$, a model indicating defect activity by D, D_c, A, R, and M. Section 4.2.3 describes model $M_{4.3}$, a model indicating defect activity by D_1, D_2, D_3, D_4, D_c, A, R, and M. In each of these sections, we hypothesize that the complexity of the source code defining RTS had a causal effect on the defect activity demonstrated during the system test phase.

4.2.1 Indicating Defect Activity with Defects and Design Changes

In this section, we describe a canonical model indicating defect activity with D and D_c. Figure 4.3 gives the path diagram for this canonical model. This figure shows the ten product and two process measures that indicate, respectively, source code complexity and defect activity. Both source code complexity and defect activity have two dimensions. The directed paths from each dimension of source code complexity to the corresponding dimension of defect activity represent the causal influence of source code complexity on defect activity.

Table 4.2 gives the canonical correlations. Both dimensions of correlation are statistically significant; both have strong canonical correlations; and both account for a reasonable proportion of the total explained variance. Thus, we select both

Canonical Dimension	Canonical Correlation	Statistically Significant ($p < 0.05$)	Proportion of Variance	Cumulative Proportion
1	0.852	yes	0.872	0.872
2	0.529	yes	0.128	1.000

Table 4.2: Model $M_{4.1}$ Canonical Correlations

dimensions of canonical correlation for interpretation.

Recall that each dimension of canonical correlation relates two linear combinations, one combining the product measures and one combining the process measures. In interpreting a dimension of canonical correlation, we consider the loadings of each of these linear combinations with its constituent variables. Table 4.3 gives these loadings along with the weights that define the linear combinations. For example,

$$\zeta^{(1)} = 0.1737D + 0.8938D_c$$

gives the linear combination of process measures forming the first dimension of defect activity, and this linear combination has loadings of 66.6 and 98.94% on, respectively, D and D_c. Thus, for the first dimension, defect activity loads strongly on both D and D_c, though notably more strongly on D_c, and source code complexity loads strongly on η_2 and moderately on XQT, both measures of size. For the second dimension, defect activity loads strongly on D and negligibly on D_c, while defect

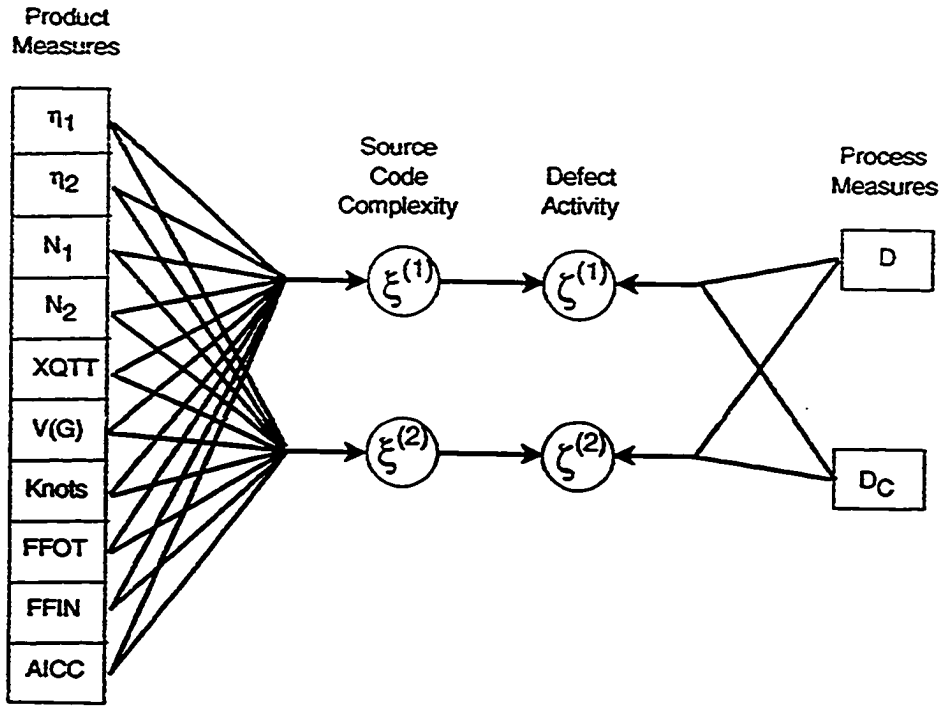

Figure 4.3: Model $M_{4.1}$ Path Diagram

activity loads strongly on $V(G)$ and moderately on η_1, XQT, and $AICC$. Both η_1 and $AICC$ are related to source code information content.

Differences in loadings across the dimensions also influence interpretations. Both dimensions load about equally on N_2, XQT, $FFOT$, and $FFIN$; thus these measures do not help in distinguishing the two dimensions. η_1, η_2, N_1, $V(G)$, $Knots$, and $AICC$ have loading differences making them helpful in distinguishing the dimensions. Considering the loadings, for both magnitude and discriminant power, the first dimension relates source code size and control flow structure to design change and source code repair activity, while the second dimension relates source code information content and predicate occurrence to source code repair activity. The correlations associated with these variates are, respectively, about 85 and about 53%.

The objective of this section was to model the relationship between source code complexity and defect activity by applying canonical correlation analysis. Product and process measures collected during the development of a commercial real-time product provided the data for this analysis. We hypothesized that source code complexity exerted a causal influence on defect activity during the system test phase of this product. Significant canonical correlations along two dimensions support this hypothesis. Interpretation of these two dimensions of canonical correlation revealed relationships between the sets of manifest variables that were not immediately apparent from their simple correlations. Specifically, η_1, η_2, $V(G)$, and $AICC$

		Dimension 1		Dimension 2	
		Weights	Loadings	Weights	Loadings
Source Code Complexity (ξ)					
η_1	(x_1)	-0.6534	0.2987	-0.9130	0.4696
η_2	(x_2)	0.4496	0.9195	-0.9273	0.0889
N_1	(x_3)	-3.0424	0.1905	-0.5051	0.3950
N_2	(x_4)	1.2095	0.1791	-1.5853	0.1849
XQT	(x_5)	2.3885	0.4622	3.6214	0.5042
$V(G)$	(x_6)	-0.3230	0.1089	-0.1926	0.6931
Knots	(x_7)	0.0382	0.2653	-0.8437	0.0419
FFOT	(x_8)	0.0934	0.3154	0.0577	0.3348
FFIN	(x_9)	-0.0358	0.2765	-0.0214	0.2427
AICC	(x_{10})	0.4457	0.1039	0.6169	0.5391
Defect Activity (ζ)					
D	(y_1)	0.1737	0.6660	1.1855	0.7460
D_c	(y_2)	0.8938	0.9894	-0.7979	-0.1449

Table 4.3: Model $M_{4.1}$ Canonical Weights and Loadings

associated with similar influences upon D , but η_2 associated with a substantially greater influence upon D_c. These relationships differentiate the two dimensions of correlation identified by the model. While *Knots* did not display a dominating loading on either dimension, its loading on the first dimension was substantial relative to its loading on the second dimension, and thus, this measure helps in differentiating the dimensions of correlation.

Note that interaction between D and D_c is likely. For example, it is reasonable to expect that a design change will introduce defects. If this explains the loadings of D and D_c on the first dimension of defect activity, then the model suggests two subsets of product measures, one related to design change activity that results in defects, and another related directly to defects. However, the model does not make a distinction, at the abstract level, between D and D_c. To make this distinction, requires a model of the relationships between three sets of variables: a set of product measures indicating source code complexity, a set of process measures indicating defect activity, and a set of process measures indicating design change activity. This suggests that soft models of greater generality than canonical correlation will provide more insight into the relationships among software engineering measures.

4.2.2 Enhancing the Model to Include Code Churn Measures

In this section, we describe a canonical model indicating defect activity with D, D_c, A, R, and M. The new indicators, A, R, and M, add code churn information to the model described in Section 4.2.1. Figure 4.4 gives the path diagram for

this canonical model. This figure shows the ten product and five process measures that indicate, respectively, source code complexity and defect activity. Both source code complexity and defect activity have five dimensions. The directed paths from each dimension of source code complexity to the corresponding dimension of defect activity represent the causal influence of source code complexity on defect activity.

Table 4.4 gives the canonical correlations. We select the first two dimensions for interpretation. Several considerations lead to this selection. First, dimensions greater than two are not statistically significant. This narrows the selection to the first two dimensions. Second, both of these dimensions have strong canonical correlations: 88.9% for the first, and 54.4% for the second. And finally, each of the first two dimensions accounts for a reasonable proportion of the total explained variance: 84.8% for the first, and 9.4% for the second. By retaining the first two dimensions, we retain about 94% of the overall variance explained by the model.

In interpreting the latent variables forming the first two dimensions of canonical correlation, we consider the loadings of the latent variables with the manifest variables. Table 4.5 gives these loadings along with the weights that form the inner relations. For example,

$$\zeta^{(1)} = -1.0442A + 1.8112R - 0.1285M + 0.1902D + 0.2495D_c$$

evaluates the first dimension of defect activity, and this latent variable has a loading of 97.23% on A. As shown in Table 4.5, the first dimension of source code complexity loads strongly on η_2 and moderately on XQT, both measures of size. The first

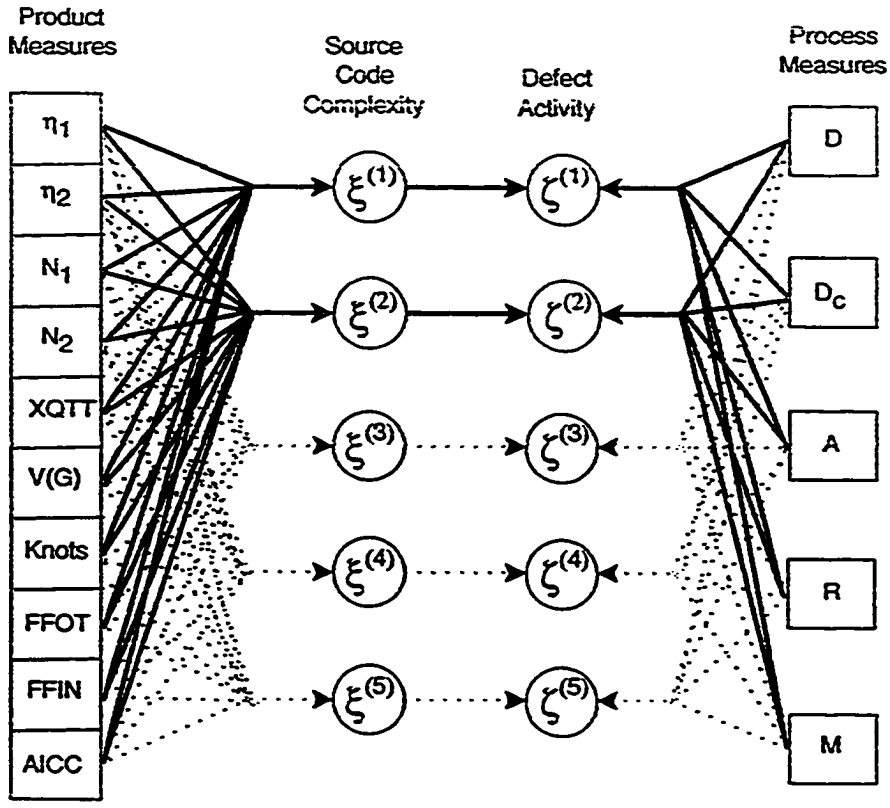

Figure 4.4: Model $M_{4.2}$ Path Diagram

Canonical Dimension	Canonical Correlation	Statistically Significant ($p < 0.05$)	Proportion of Variance	Cumulative Proportion
1	0.889	yes	0.848	0.848
2	0.544	yes	0.094	0.942
3	0.337	no	0.029	0.971
4	0.289	no	0.021	0.992
5	0.187	no	0.008	1.000

Table 4.4: Model $M_{4.2}$ Canonical Correlations

dimension of defect activity loads strongly on A, R, M, and D_c. It also loads moderately on D . Thus the first dimension of canonical correlation relates source code size to a general notion of change. The correlation at this dimension is excellent at 89%. Note that this correlation between the latent constructs exceeds that of any pair of manifest variables given in Table 4.1, the closest simple correlation in this table being about 80% between η_2 and R.

The second dimension of source code complexity loads moderately on η_1, N_1, XQT, $V(G)$, and $AICC$. Note that η_1 and N_1 are used to derive $AICC$. These three loadings are directly related to information content. The second dimension of defect activity loads strongly on D alone. This dimension of canonical correlation relates source code information content and predicate content to program defects. The correlation at this dimension is about 54%.

Differences in loadings across dimensions are also important in interpreting the latent variables. The first dimension of source code complexity has a loading of 92.61% with η_2. In the second dimension, this loading is 7.45%, about as close to 0% as 92.61% is to 100%. Thus, η_2 is useful in distinguishing the two dimensions of source code complexity. While their loading differences are not as extreme, similar observations hold for $V(G)$ and $AICC$. However, both dimensions load about equally on XQT; thus this measure does not help in distinguishing the two dimensions of source code complexity.

The two dimensions of defect activity have large differences in loadings on A, R, M, and D_c. The difference in loading on D is not as extreme. However, the loading on D is the lowest loading of the first dimension, and the highest of the second. While D contributes to both dimensions of defect activity, its relation to the other manifest endogenous variables differentiates its contribution across these dimensions. Its appearance in the first dimension of defect activity suggests that both the number of program defects and the number of modifications required to remove defects varied directly with the first dimension of complexity. That is, more defects fell in larger files, and these files required more modifications to remove defects. Its appearance in the second dimension of defect activity suggests that the number of program defects varied directly with the second dimension of complexity, but the number of modifications required to remove defects is nearly independent of this dimension of complexity. That is, more defects fell in files having more

predicates and information content, but the number of modifications required to remove these defects was independent of the predicate and information content measures.

The objective of this section was the same as that of Section 4.2: to model the relationship between source code complexity and defect activity by applying canonical correlation analysis. Again we hypothesized that source code complexity exerted a causal influence on defect activity experienced during the system test phase of RTS. In this analysis, an expanded set of process measures indicated defect correction activity. Significant canonical correlations along two dimensions support our hypothesis. Interpretation of these two dimensions of canonical correlation revealed that η_2, $V(G)$, and $AICC$ associated with similar influences upon D , but η_2 associated with a substantially greater influence upon A, R, M, and D_c.

Note that the interaction between D_c, A, R, and M is likely to be more pronounced than the interaction between D , A, R, and M. That is, design changes either introduce functionality or correct design defects, and thus, it is reasonable to expect that design changes will typically result in more file changes than defect removals will. Thus, the results of this section support the same conjecture given in Section 4.2: the two significant dimensions identify two subsets of product measures, one related to design change activity that results in defects, and another related directly to defects.

		Dimension 1		Dimension 2	
		Weights	Loadings	Weights	Loadings
Source Code Complexity (ξ)					
η_1	(x_1)	-0.4744	0.2964	-0.9999	0.4530
η_2	(x_2)	0.3500	0.9261	-1.0338	0.0745
N_1	(x_3)	-3.6774	0.2028	-1.0062	0.4070
N_2	(x_4)	1.6920	0.2011	-1.3722	0.2018
XQT	(x_5)	2.5449	0.4677	4.0669	0.5108
$V(G)$	(x_6)	-0.2069	0.1114	-0.2655	0.6935
Knots	(x_7)	-0.0244	0.2186	-0.8136	0.0711
FFOT	(x_8)	0.1126	0.3316	0.0396	0.3097
FFIN	(x_9)	-0.0004	0.2834	-0.1110	0.1795
AICC	(x_{10})	0.2865	0.0798	0.7208	0.5392
Defect Activity (ζ)					
D	(y_1)	0.1902	0.6349	1.1657	0.7278
D_c	(y_2)	0.2495	0.9437	-0.8930	-0.1327
A	(y_3)	-1.0442	0.9723	3.6177	-0.1609
R	(y_4)	1.8112	0.9743	-3.1652	-0.1664
M	(y_5)	-0.1285	0.8215	-0.4030	-0.2190

Table 4.5: Model $M_{4.2}$ Canonical Weights and Loadings

4.2.3 Enhancing the Model to Include Defect Severity

Failures have various impacts on the system testing effort. Some have an extreme impact on the system testing cost and schedule; others have little or no impact. In this section, we describe a canonical model that considers the impacts of defect severity on defect activity. To achieve this, we indicate defect activity with D_1, D_2, D_3, D_4, D_c, A, R, and M. Figure 4.5 gives the path diagram for the canonical model. This figure shows the ten product and eight process measures that indicate, respectively, source code complexity and defect activity. Both source code complexity and defect activity have eight dimensions. The directed paths from each dimension of source code complexity to the corresponding dimension of defect activity represent the causal influence of source code complexity on defect activity.

We select the first two dimensions of canonical correlation for interpretation. Thus, Figure 4.5 differentiates these dimensions. Several considerations led to this selection. Lack of significance excludes dimensions greater than four. Of the four remaining, the first two account for about 88% of the overall variance explained by the model, the first accounting for about 78%, the second about 10%. The third and fourth dimensions combined account for only about 7% of the overall variance explained by the model, and thus, relative lack of explained variance excludes these dimensions. Finally, both of the selected dimensions have strong canonical correlations: 89.5% for the first, and 59.3% for the second. Table 4.6 summarizes this

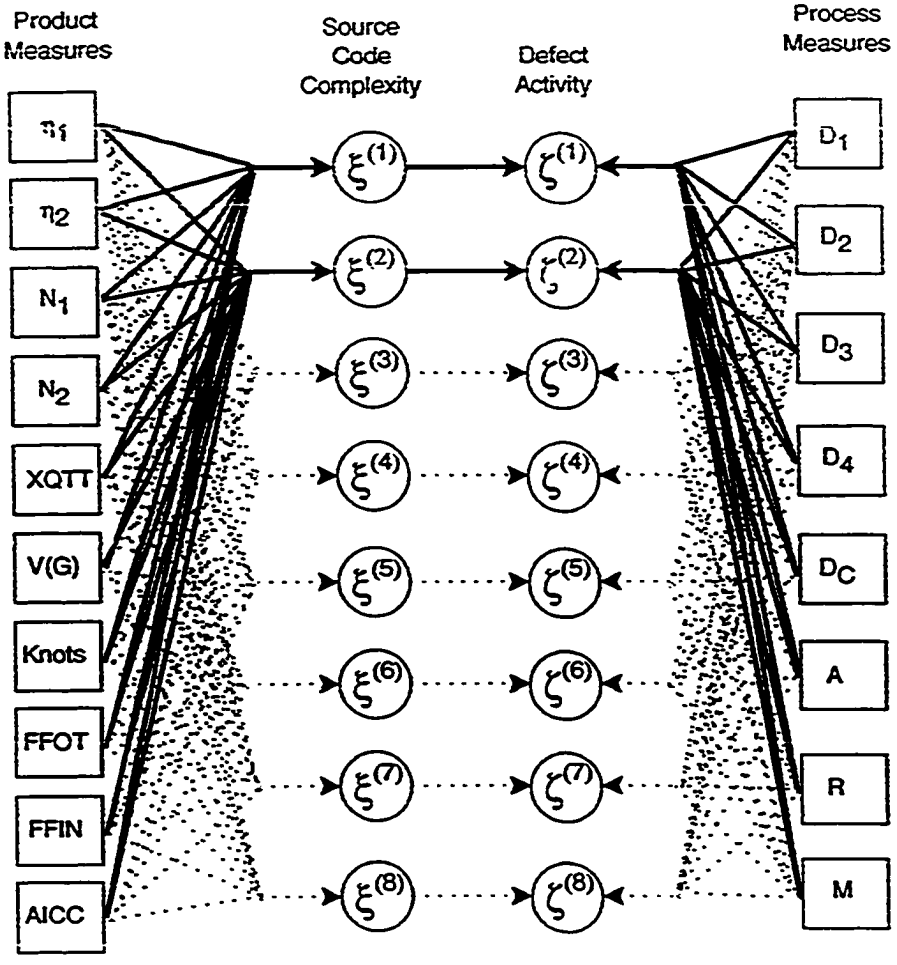

Figure 4.5: Model $M_{4.3}$ Path Diagram

Canonical Dimension	Canonical Correlation	Significant ($p < 0.05$)	Proportion of Variance	Cumulative Proportion
1	0.895	yes	0.780	0.780
2	0.593	yes	0.105	0.885
3	0.422	yes	0.042	0.927
4	0.375	yes	0.032	0.958
5	0.312	no	0.021	0.979
6	0.252	no	0.013	0.992
7	0.164	no	0.005	0.998
8	0.111	no	0.002	1.000

Table 4.6: Model $M_{4.3}$ Canonical Correlations

information. These two dimensions of canonical correlation support the hypothesis that source code complexity exerted a causal influence on the defect activity experienced during the system test phase of RTS.

In interpreting the first two dimensions of canonical correlation, consider the loadings of the latent variables with the manifest variables. Table 4.7 gives these loadings along with the weights that form the inner relations. For example,

$$\zeta^{(1)} = -0.5581A + 1.3643R - 0.1156M + 0.0823D_c$$

$$-0.1041D_1 + 0.2562D_2 - 0.0409D_3 + 0.0940D_4$$

evaluates the first dimension of defect activity, and this latent variable has a loading

of 96.52% on A. As shown in Table 4.7, the first dimension of source code complexity loads strongly on η_2 and moderately on XQT, both measures of size. The first dimension of defect activity loads strongly on A, R, M, and D_c. It also loads moderately on F_2 and F_4.

The second dimension of source code complexity loads strongly on $V(G)$. It also loads moderately on η_1, N_1, XQT, and $AICC$. Note that η_1 and N_1 are used to derive $AICC$. These three loadings are directly related to information content. The second dimension of defect activity loads strongly on F_3, and moderately on D_1 and D_2.

Differences in loadings across dimensions are also important in interpreting the latent variables. The first dimension of source code complexity has a loading of 93.32% with η_2. In the second dimension, this loading is 9.28%. Thus, η_2 is useful in distinguishing the two dimensions of source code complexity. While their loading differences are not as extreme, similar observations hold for N_1, $V(G)$, $FFIN$, and $AICC$. However, both dimensions load about equally on η_1, N_2, XQT, $FFOT$, and $Knots$; thus these measures do not help in distinguishing the two dimensions of source code complexity. The two dimensions of defect activity have large differences in loadings on A, R, M, D_1, D_3, D_4, and D_c.

Note that, in the first dimension of correlation, D_c and the code churn measures, A, R, and M, all load strongly. In the second dimension, these loadings are weak. Since design changes either introduce functionality or correct design defects,

it is reasonable to expect that design changes will typically result in more file changes than defect removals will. The two dimensions of correlation suggest that two subsets of product measures had different relationships to the process measures. One subset, $\{\eta_2, FFIN\}$, was related to design change activity that resulted in defects, and the other, $\{N_1, V(G), AICC\}$ was related directly to defects.

Both dimensions load about equally on D_2. However, their loadings on the remaining defect severity classes differentiate the risk that they present to the system test process. The first dimension loads strongly on D_4; the second on D_1 and D_3. Thus, in the RTS system testing effort, defects having less impact on the system test process associated with design change activity that occurred during the system test phase, while those having more impact associated with source code complexity at entry to the system test phase. Recall that RTS evolved from a previous product to incorporate support for new platform features. Planned design changes required to support these features entered the product before the product entered the system test phase. Thus, developers implementing planned design changes got no feedback from the system test organization about the impacts of their source code changes on system testing effort. This feedback began with the completion of planned design change activity, and the entry of the product into the system test phase. This process attribute, and the tendency for the system test organization to become more selective in allowing design changes as the system test completion date approaches, may account for the differing associations with defect severity seen across the two

dimensions of canonical correlation.

4.3 Conclusions

In this chapter, we demonstrated the use of canonical correlation analysis to model the relationships between software engineering measures. Three canonical correlation models supported the hypothesis that source code complexity exerted a causal influence on the defect activity experienced during system test for RTS. Interpretation of these models suggested that two subsets of product measures have different relationships with process activity. One is related to design change activity that results in defects, and the other is related directly to defects. Further, defects having less impact on the system test process associated with design change activity that occurred during the system test phase, while those having more impact associated with source code complexity at entry to the system test phase.

We emphasize two aspects of our presentation. First, we restrict our findings to the development effort that we studied. We do not imply that either the weights or the loadings of the relations generalize to all software development efforts. Such generalization is untenable since the models did not represent many important influences on the modeled latent variables, for example, schedule pressure, testing effort, product domain, and level of engineering expertise. We demonstrated canonical correlation analysis as a useful exploratory tool for software engineers interested in understanding influences that affected past development efforts. These influences

	Dimension 1		Dimension 2	
	Weights	Loadings	Weights	Loadings
Source Code Complexity (ξ)				
η_1 (x_1)	-0.4338	0.3068	-0.6492	0.4337
η_2 (x_2)	0.4148	0.9332	-1.1535	0.0928
N_1 (x_3)	-3.2406	0.1973	-3.0343	0.4462
N_2 (x_4)	1.4521	0.1968	0.2687	0.2406
XQT (x_5)	2.3081	0.4600	4.2199	0.5674
$V(G)$ (x_6)	-0.2582	0.0882	0.2892	0.8466
$Knots$ (x_7)	-0.0338	0.1956	-0.5813	0.2454
$FFOT$ (x_8)	0.1271	0.3434	-0.0282	0.2623
$FFIN$ (x_9)	0.0056	0.2933	-0.1906	0.0625
$AICC$ (x_{10})	0.2643	0.0830	0.4089	0.4908
Defect Activity (ζ)				
A (y_1)	-0.5581	0.9652	1.4126	-0.0979
R (y_2)	1.3643	0.9674	-0.6069	-0.1026
M (y_3)	-0.1156	0.8191	-0.5865	-0.1825
D_c (y_4)	0.0823	0.9363	-0.6528	-0.0839
D_1 (y_5)	-0.1041	-0.0853	0.3747	0.6579
D_2 (y_6)	0.2562	0.6804	0.6187	0.5802
D_3 (y_7)	-0.0409	0.4004	0.5462	0.7228
D_4 (y_8)	0.0940	0.7411	-0.3945	0.2180

Table 4.7: Model $M_{4.3}$ Canonical Weights and Loadings

could also affect current development efforts, however work remains to specify subsets of indicators and development efforts for which the technique becomes useful as a predictive tool.

Second, we explained canonical correlation analysis as a restricted form of soft modeling. We chose this approach not only because the terminology and graphical devices of soft modeling allow straightforward high-level explanations, but also because we are interested in the general method. The general method allows models involving many latent variables having interdependencies. The results of this study suggest that models involving more than two latent variables will provide more insight into the relationships among software engineering measures than canonical models can. The general method is intended for modeling complex interdisciplinary systems having many variables and little established theory. Further, it incorporates parameter estimation techniques relying on no distributional assumptions. Our future research will focus upon developing general soft models of the software development process for both exploratory analysis and prediction of future performance.

Chapter 5

ENHANCEMENT ACTIVITY AND THE DETECTION OF HIGH-RISK PROGRAM MODULES

Researchers have exploited the relationship between software software product measures and defect distribution to develop models that identify program modules presenting high-risk during system testing [18, 61, 4, 62, 5, 26]. Knowing the modules that present high-risk, software engineers can make more informed staffing, training, and scheduling decisions in preparation for system testing. Concentrating testing effort on areas of high-risk can result in more efficient defect removal and delivery of a higher quality product [63]. In software development efforts characterized by entry of functional enhancement during system testing, this knowledge allows software engineers to better understand and predict the impact of these functional enhancements on the testing schedule. Used in these ways, models identifying high-risk program modules help engineers to control and cope with the defect injection rates associated with the functional enhancement of existing products.

Recently, researchers have reported a relationship between functional enhancement and defect distribution [64, 63, 41, 42]. Defects in enhanced program modules often concentrate within the modified code [63]. In this chapter, we exploit

this relationship between enhancement activity and defect distribution to improve models that identify high-risk program modules. Based upon data collected during the functional enhancement of a commercial programming language processing utility, we develop three discriminant models that classify program modules as high- or low-risk. One model classifies based upon a selection of source code measures; another based upon a selection of both source code and enhancement activity measures; and another based upon a selection of enhancement activity measures. This work builds upon recently reported modeling efforts [26] by considering a source of variation other than source code complexity. Comparison of the models reveals the importance of the enhancement activity in models of risk.

5.1 An Overview of Discriminant Analysis

Generally, discriminant models optimally assign observations to two or more labeled classes based upon properties having observed values that vary somewhat from class to class. Discriminant models are derived to produce the fewest misclassifications in a sample of observations with known class memberships, and are applied to give leading indications of future performance based on known property values.

In this chapter, we apply discriminant techniques to derive models that classify program modules as either high- or low-risk. Thus, we fit two-class discriminant models in which the observations are program modules, and the independent variables, or properties on which classification is based, are either measures indicating

software complexity, or measures indicating both software complexity and enhancement activity. A module is known to be high- or low-risk after a suitable period of testing reveals the number of defects discovered in it. That is, the criterion variable for discrimination is the number of discovered defects. A cutoff value for the criterion variable divides the set of modules into two classes: modules having more than the cutoff value of discovered defects are high-risk, while those having no more than this number are low-risk. The sizes of the two classes, and thus, the appropriate cutoff value will vary with aspects of the product under development, and the software development process. Software engineers select this cutoff value based upon the history of similar projects.

In the linear discriminant model that we develop, an observation, $\mathbf{x}_{ij}$, $i = 1,2$, $1 \leq j \leq n_i$, is a vector of p measures for the j^{th} program module of class C_i, $i = 1$ designating low-risk, $i = 2$ designating high-risk. We expect a certain proportion, π_1, of program modules to fall in C_1, and the remaining proportion, $\pi_2 = 1 - \pi_1$, to fall in C_2. Let $f_i(\mathbf{x})$ give the probability density function of those $\mathbf{x}$ falling in C_i, $i = 1,2$. Then to minimize the total probability of misclassification, the discriminant model assigns an observation, $\mathbf{x}$, to class C_1 if

$$\frac{f_1(\mathbf{x})}{f_2(\mathbf{x})} > \frac{\pi_2}{\pi_1}, \tag{5.1}$$

and to class C_2 otherwise. This is equivalent to assigning $\mathbf{x}$ to the class having greater posterior probability of membership,

$$q_i(\mathbf{x}) = \frac{f_i(\mathbf{x})\pi_i}{f_1(\mathbf{x})\pi_1 + f_2(\mathbf{x})\pi_2} \tag{5.2}$$

The problem becomes how to define $f_i(\mathbf{x})$, $i = 1, 2$. There are many approaches to this, giving rise to a large collection of models differentiated by assumptions drawn on the independent variables and their distributions in each of the modeled classes, or by methods employed to estimate these distributions [65]. We limit our discussion to two approaches,

1. one among those assuming, at least, that the classes have known distributions, that is, among the *parametric* models, and

2. the other among those that do not require this assumption, that is, among the *nonparametric* models.

For the parametric approach, assume that the observations of class C_i have a multivariate normal distribution with mean μ_i, and covariance matrix Σ_i. With these assumptions we have

$$f_i(\mathbf{x}) = (2\pi)^{\frac{-n_i}{2}} |\Sigma_i|^{-\frac{1}{2}} e^{-\frac{1}{2}(\mathbf{x}-\mu_i)'\Sigma_i^{-1}(\mathbf{x}-\mu_i)}$$

All that remains to derive the discriminant function is an application of equation 5.1.

For the nonparametric model, having no knowledge regarding the distributions of the observations in class C_i, we estimate the probability density functions directly from the sample data, $\mathbf{x}_{ij}$, $i = 1, 2$, $1 \leq j \leq n_i$. Employing the potential function, or *kernel* method of multivariate density estimation for this purpose [65], we estimate $f_i(\mathbf{x})$ with

$$\hat{f}_i(\mathbf{x}|\lambda) = \frac{1}{n_i} \sum_{j=1}^{n_i} K_i(\mathbf{x}|\mathbf{x}_{ij}, \lambda),$$

where $K_i(\mathbf{y}|\mathbf{z},\lambda)$ gives a kernel probability density function on $\mathbf{y}$ with mode at $\mathbf{z}$ and smoothing parameter λ. A large collection of kernel probability density functions differentiates applications of this approach. For this study, we selected the *normal kernel*, and thus,

$$K_i(\mathbf{y}|\mathbf{z},\lambda) = (2\pi\lambda^2)^{\frac{-n_i}{2}}|\mathbf{S}_i|^{-\frac{1}{2}}e^{\frac{-1}{2\lambda^2}(\mathbf{y}-\mathbf{z})'\mathbf{S}_i^{-1}(\mathbf{y}-\mathbf{z})}$$

where $\mathbf{S}_i$ gives the covariance matrix for the sample data in class C_i. Substitution into equation 5.1 with $f_i(\mathbf{x}) = \hat{f}_i(\mathbf{x}|\lambda)$ yields the discriminant function.

Before applying either approach, a model selection technique—in this study, stepwise discriminant analysis [43]—selects from a set of $m \geq p$ measures, the p independent variables defining each observation. These p measures each contribute significantly to the model. It is likely that observations comprised of a large selection of source code measures will have some measures with strong correlations [54]. At the same time, larger selections of source code measures are likely to provide more information. By selecting models from the first few principal components of a relatively large selection of source code measures, we consider a large proportion of source-code-measure variance with relatively few orthogonal measures. This avoids model selection problems resulting from correlations among the independent variables [43].

5.2 A Discriminant Model of Software Risk

With the general background discussed in Section 5.1, it is possible to discuss the specific models of interest in this chapter. While the methodology employed would be equally useful in a wide range of software development environments, we stress that the specific models developed in this section are intended for classifying the program modules that result from the next iteration of the same development process, in production of the next release of the modeled product. Further, in this application of the models, we assume that the same key people will implement the software changes required to produce the next release, and that the development environment will remain unchanged. With care, engineers can loosen these assumptions. For example, if the assignment of key people changes, one should expect the classification results to degrade in relation to the differences this introduces in programming skill level and product understanding. Further, one can quantify product differences [66], and expect classification results to degrade in relation to this quantity.

The product development effort that yielded the data under study provided functional enhancements to a commercial programming language processing utility (LPU). Responsibility for the design and implementation of these enhancements rested primarily with one software engineer. The same engineer supported an independently managed system testing organization while this organization exposed the enhanced product to system test cases. This engineer also supported field testing at

a selection of customer sites. The source code defining LPU is primarily C although some support code is assembly language. This study considers the C code; thus, in this study, *modules* are C functions. LPU is composed of 29 source files defining 369 C functions.

The following 17 source code measures, evaluated for each of the 369 LPU functions *on entry to* the system test phase, provide a cross-section of software complexity data for both discriminant models: XQT, η_1, N_1, η_2, N_2, GF, $Paths$, $MPath$, $\overline{Path}$, $Loops$, $Nodes$, $Edges$, $V_2(G)$, $Band$, $Data$, $FFOT$, and $FFIN$. The following four measures, evaluated for each of the 369 LPU functions *on entry to* the system test phase, provide a cross-section of enhancement activity data for one discriminant model: E, A_e, C_e, and R_e. The number of defects, D, evaluated for each of the 369 LPU functions *on exit from* the system test phase, gives the criterion variable for classification.

The proportion of modules that introduce high-risk to system testing is likely to vary across development efforts. Discussions with the software engineer having primary responsibility for both enhancing the product and supporting the enhanced product during system testing support the choice of cutoff value $D = 1$ for the project under study.

It is important to note that the enhancement activity measures are known before functional testing of the enhancements can begin since the changed code

provides the functionality under functional test. In many software development organizations, these enhancements are implemented before the system testing effort begins. In others, functional enhancements can enter during the system testing effort. Discriminant models are useful to organizations having either of these characteristics. As previously mentioned, these models carry an additional benefit to organizations that allow functional enhancement during the system testing effort: knowledge of high-risk enhancements allows software engineers to better understand and predict the impact of these functional enhancements on the testing schedule.

Also, note that other selections of measures could serve to quantify software complexity and enhancement activity. Further, another measure, or selection of measures, could serve as the criterion for classification, and other cutoff values of the selected criterion could be appropriate. Our goal in this chapter is to present an improved discriminant model for the identification of high-risk program modules by extending the set of independent variables to include enhancement activity measures as well as complexity measures. We do not intend to justify the use of any particular selection of measures or any specific criterion variable cutoff value.

Presented with a sample of observations, a two-class discriminant model must assign each observation to one and only one of two classes. Thus, our two-class discriminant models can commit two types of misclassification: Type 1 in which the model assigns a low-risk function to the high-risk class, and Type 2 in which the model assigns a high-risk function to the low-risk class. We compare the two models

that we develop by considering their Type 1 and Type 2 misclassification rates. Data splitting allows us to quantify these model attributes. Applying this technique, we randomly partitioned the LPU functions into two subsets: a subset for developing the models, and a subset for quantifying their misclassification rates. The subset for model development contains *fitting* data: the complexity, enhancement activity, and criterion measure values for 246 LPU functions. The subset for quantifying misclassification rates contains *testing* data: the complexity, enhancement activity, and criterion measure values for 123 LPU functions. The testing data do not contribute to model development, and thus, application of a fitted model to the testing data gives an indication of the model's predictive quality.

In studies of five telecommunications software products, Le Gall *et al.* identified only 4 to 6% of the modules as high-risk [67]. Others set the proportion of high-risk modules as high as 20% [23]. The criterion variable cutoff value for this study, $D = 1$, isolates about 11% of the fitting data set observations as high-risk.

After data splitting we derived the principal components of the standardized source code measures in the fitting data set. Table 5.1 gives the loadings on the first four principal components after varimax rotation. As shown in this table, these four principal components account for nearly 88% of the source-code-measure variance. The first component has dominating loadings on *GF*, *FFOT*, and seven measures related to size: *XQT*, N_2, *Edges*, *Nodes*, $V_2(G)$, η_2, and N_1. The second component has dominating loadings on *FFIN*, and three measures related to control flow: *Paths*,

$\overline{Path}$, *MPath*, and *Loops*. The third component has dominating loadings on η_1, and the control flow measure *Band*. Finally, the fourth component has the dominating loading on *Data*. Note that *MPath*, *Loops*, and η_1 display ambiguous association having nearly equal loadings across three or more principal components. The first four principal components appear to represent, respectively, size, paths, nesting, and data structures complexity. *GF* and *FFOT* vary with size, and *FFIN* varies with paths.

Table 5.2 gives the standardized transformation matrix, $\mathbf{T^4}$, for the first four principal components. Using this 17 by 4 matrix and $\mathbf{Z}_{fit}$, the 246 by 17 matrix of standardized source code measures for the functions in the fitting data set, we computed $\mathbf{P}^4_{fit}$, the 246 by 4 matrix of principal components values for the fitting data set,

$$\mathbf{P}^4_{fit} = \mathbf{Z}_{fit}\mathbf{T^4} = [\mathbf{P}_1, \mathbf{P}_2, \mathbf{P}_3, \mathbf{P}_4]$$

where the vector of 246 values for principal component $\mathbf{P}_i$, $1 \leq i \leq 4$, are distributed with zero mean and unit variance.

Similarly, using the standardized transformation matrix and $\mathbf{Z}_{test}$, the standardized source code measures for the functions in the testing data set, we computed four principal components values for each function of the testing data set,

$$\mathbf{P}^4_{test} = \mathbf{Z}_{test}\mathbf{T^4}$$

The principal component values, $\mathbf{P}^4_{fit}$ and $\mathbf{P}^4_{test}$, served as the independent variables quantifying software complexity for, respectively, fitting and testing the two

Measure	Principal Component			
	1	2	3	4
XQT	**0.87**	0.31	0.31	0.18
N_2	**0.86**	0.20	0.22	0.33
Edges	**0.85**	0.36	0.34	0.09
Nodes	**0.84**	0.36	0.34	0.11
$V_2(G)$	**0.83**	0.39	0.33	0.04
η_2	**0.81**	0.22	0.26	0.40
N_1	**0.80**	0.19	0.18	0.36
GF	**0.78**	0.28	0.19	0.20
FFOT	**0.75**	0.23	0.13	0.33
FFIN	0.21	**0.89**	0.05	0.02
Paths	0.50	**0.74**	0.14	0.22
$\overline{Path}$	0.46	**0.56**	0.39	0.45
MPath	0.46	**0.54**	0.43	0.46
Loops	0.10	**0.52**	0.47	0.51
Band	0.35	0.12	**0.87**	0.12
η_1	0.56	0.16	**0.57**	0.44
Data	0.59	0.09	0.14	**0.67**
Eigenvalues	7.58	3.04	2.32	1.97
% Variance	44.58	17.86	13.64	11.57
Cumulative % Variance	44.58	62.43	76.07	87.64

Table 5.1: Principal Components Loadings

discriminant models.

Table 5.3 gives the independent variables presented to model selection for these models. For model $M_{5.1}$, we follow [26] by presenting the model selection procedure with only source code measures. For model $M_{5.2}$, we extend the set of independent variables to include enhancement activity measures. For model $M_{5.3}$, we include only enhancement activity measures. Model selection for the three models proceeded on the fitting data set, selecting the independent variables given in Table 5.3. Models $M_{5.1}$ and $M_{5.2}$ included the complexity measures P_1 and P_4, measures related to size and data structure complexity, respectively. The inclusion of P_4 underscores the importance of considering variations in data structure complexity by demonstrating a relationship between this measure and the distribution of defects [68, 35]. In addition to the two complexity measures, model $M_{5.2}$ included the enhancement activity measures E and A_e, the number of enhancements and the number of noncomment source lines added to provide these enhancements, respectively. Model $M_{5.3}$ included only the enhancement activity measures E and A_e. For each independent variable included by model selection, Table 5.4 gives the high, low, and average value of fitting data set observations.

In fitting the models, we defined $\pi_1 = 0.89$ and $\pi_2 = 0.11$. These values are proportional to the occurrence of fitting data set observations in C_1 and C_2, respectively. For models $M_{5.1}$ and $M_{5.2}$ we applied the parametric technique described in Section 5.1. For model $M_{5.3}$ we applied the nonparametric technique. In this fitting,

Measure	Principal Component			
	1	2	3	4
XQT	0.18523	-0.02462	0.02139	-0.17775
N_2	0.17823	-0.10707	-0.12095	0.08392
Edges	0.18589	0.02265	0.10106	-0.32653
Nodes	0.17803	0.02482	0.09234	-0.30065
$V_2(G)$	0.19090	0.06342	0.09821	-0.39080
η_2	0.12465	-0.10800	-0.09431	0.17883
N_1	0.15456	-0.10725	-0.16206	0.16936
GF	0.17819	-0.00704	-0.10007	-0.07946
FFOT	0.14966	-0.05169	-0.20760	0.15048
FFIN	-0.09352	0.59995	-0.16309	-0.20691
Paths	-0.02386	0.38952	-0.18694	-0.02152
$\overline{Path}$	-0.12936	0.17673	0.04884	0.23800
MPath	-0.14322	0.15367	0.10283	0.24035
Loops	-0.31589	0.19086	0.21174	0.40679
Band	-0.10942	-0.16061	0.85664	-0.32128
η_1	-0.05573	-0.16442	0.32221	0.18317
Data	0.00392	-0.18144	-0.27240	0.69404

Table 5.2: Standardized Transformation Matrix (T^4)

Independent Variable	Model					
	$M_{5.1}$		$M_{5.2}$		$M_{5.3}$	
	P	S	P	S	P	S
Complexity						
P_1	•	•	•	•		
P_2	•		•			
P_3	•		•			
P_4	•	•	•	•		
Enhancement						
E			•	•	•	•
A_e			•	•	•	•
C_e			•		•	
R_e			•		•	

Table 5.3: Independent Variables Presented (P) to and Selected (S) by Model Selection

Independent Variable	Low Value	Average Value	High Value
P_1	-1.6	0.0	10.6
P_4	-3.6	0.0	3.7
E	0.0	1.5	4.0
A_e	0.0	1.5	51.0

Table 5.4: Statistics for Variables Selected by Model Selection

we varied λ from 0.3 to 0.9 noting little difference across models fitted with λ in this range. $\lambda = 0.6$ produced the best fit, and thus, we report results of the model fitted with this value.

Note that, in fitting two-class models with the classes low-risk and high-risk, [26] biased the fitting data set by excluding observations having criterion variable values in a range near the cutoff value. In testing model performance, [26] excluded modules in the testing data set having criterion variable values in this range. That

is, misclassifications of modules in the testing data set having criterion variable values in the range excluded from the fitting data set did not contribute to the reported misclassification rates. By definition, these modules must belong to one of the two discriminant classes. Thus, as reported, the biasing procedure understated the misclassification rates. Further, we find that observations in the range excluded from the fitting data fall near the hyperplane that separates the two classes, and are thus, critical to the fitting procedure. We did not bias the fitting data set.

Applying the fitted models to the observations in the testing data set yielded the misclassification rates reported in Table 5.5. Providing enhancement information resulted in substantial misclassification rate improvements. For observations of both classes, model $M_{5.2}$, the model including both enhancement activity and source code measures, committed half as many misclassifications as model $M_{5.1}$, the model including only source code measures. Model $M_{5.2}$ misclassified only 10 of 108 low-risk functions and 3 of 15 high-risk functions.

While the overall misclassification rate rose slightly from model $M_{5.2}$ to model $M_{5.3}$ with the exclusion of source code measures, we note that the distribution of misclassification across the two error types improved with model $M_{5.3}$. A Type 1 error occurs when the model identifies a low-risk module as high-risk. This could result in some wasted attention to a low-risk module. A Type 2 error occurs when the model identifies a high-risk module as low-risk. This could result in either the release of a lower quality product or an extension of the scheduled release date as

Model	Misclassification Type		
	1	2	Overall
$M_{5.1}$	18.52%	40.00%	21.14%
$M_{5.2}$	9.26%	20.00%	10.57%
$M_{5.3}$	11.10%	13.30%	11.40%

Table 5.5: Misclassification Rates

more effort is required than planned for. The nature of the impacts of these error types suggests that the Type 2 error rate is more important than the Type 1 error rate in considering the quality of discriminant models.

As mentioned previously, $M_{5.1}$ and $M_{5.2}$ are parametric models. Since violations of the distribution assumptions of this method could have degraded the performance of the fitted models, we repeated the comparison using the nonparametric discriminant modeling technique described in Section 5.1. For both $M_{5.1}$ and $M_{5.2}$, the function-to-class mapping produced by fitting with the nonparametric technique was identical to that produced by fitting with the parametric technique. Thus, the misclassification rates reported in Table 5.5 also hold for the models when they are fitted using the nonparametric technique.

Discussions of the misclassified functions with the engineer having primary responsibility for designing and implementing the enhancements yielded interesting insights. For example, one low-risk function that $M_{5.2}$ classified as high-risk had slightly greater than average size, $P_1 = 0.35$, and greater than average data structure

complexity, $P_4 = 2.76$. It was also subject to more than the average number of enhancements, $E = 3$, which added to it a greater than average number of lines, $A_e = 8$. Based on this information, the model identified this function as high-risk. The primary LPU engineer identified this function as a high-level control routine. While this function was changed three times to enhance LPU, these changes were merely the addition of calls to the lower-level functions that implemented the new functionality. Due to their broad scope of control, high-level control routines are likely to require changes as a product is enhanced. Yet, these changes are often simple compared with the related changes required in the lower-level routines. This leads us to hypothesize that the level of routines in the calling hierarchy introduces variance in defect distribution. This study did not include a measure for calling hierarchy level.

Another example, involves a high-risk function that $M_{5.2}$ classified as low-risk. This relatively simple function, having complexity measures $P_1 = -0.30$ and $P_4 = -0.99$ was subject to one functional enhancement involving no added lines of code, $E = 1$ and $A_e = 0$. Based upon this information, model $M_{5.2}$ classified this function as low-risk. However, system testing recorded two defects against it, $D = 2$, and thus, the module actually appeared to be high-risk. Discussion of this function with the primary engineer revealed that the two defects recorded against it resulted from the substitution of one code sequence for another, followed by the restoration of the original code sequence. The first substitution was intended to simplify the

code, but was latter found to degrade performance. This prompted the restoration of the original code. Thus, the net result of the two defect removal changes left the code as it would have been had no defect been recorded. This demonstrates that some indication of defect type or impact could improve the discriminant results.

5.3 Conclusions

In this chapter, we exploited the relationship between enhancement activity and defect distribution to improve models that identify high-risk program modules. We achieved this by extending the set of independent variables to include enhancement activity measures as well as source code measures. Comparison of models fitted both with and without the enhancement activity measures demonstrated that inclusion of enhancement information yields substantial misclassification rate improvements. Comparison of models fitted both with and without the source code measures demonstrated that enhancement activity measures are sufficient for discriminating between high- and low-risk modules. Parametric and nonparametric discriminant modeling techniques yielded similar results.

One of the models considered included two sources of variation in defect distribution: source code complexity and enhancement activity. Other sources of variation, such as differences in the product to be enhanced, in programmer skill level, in programmer product understanding, and in the software development process, were not modeled, but remained constant throughout the development effort

that yielded the modeled data. In software development organizations that iteratively produce enhanced releases of the same product, software engineers can control these unmodeled sources of variation when applying discriminant models.

Such models are intended for classifying the program modules that result from the next iteration of the same development process, in production of the next release of the modeled product, with the same key people implementing the software changes. In this application, the unmodeled sources of variation should contribute little to the distribution of defects. However, we expect the learning experience of the previous release to result in some change in programming skill level and product understanding. These changes will vary with individuals, the scope of their responsibility, and the time between releases. Care in changing the assignments of key people could control several sources of variation in software development organizations that lack stable assignments. The discriminant technique scales to larger development efforts involving several key people by either developing unique models for each area of responsibility, or adding independent variables that account for variance introduced by differing skill and understanding levels.

Finally, consideration of the misclassified functions led us to hypothesize that the level of routines in the calling hierarchy introduces variance in defect distribution. Further, we noted that the impact of a defect is an indicator of the risk that it presents, and thus consideration of defect classifications could improve the discriminant results [69, 70]. Pursuing these points remains as future work.

Chapter 6

AN EMPIRICAL MODEL OF ENHANCEMENT INDUCED DEFECT ACTIVITY

Many software development organizations iteratively produce functionally enhanced versions of the same product, or the same group of products [61]. When these organizations produce complex products for dynamic markets, product requirements will change during the development process. For these products, functional enhancements often enter during the system testing phase. Typically, the enhancement process injects defects. The test process isolates some of these defects, field use isolates others, and others remain latent. As the scheduled system test completion date approaches, less time remains to isolate defects without customer involvement. Thus, defect injection rates become more critical.

At this time, proposed changes to incorporate functional enhancements stimulate debate between those employed to support system test and those employed to support marketing. Allowing late-arriving functional enhancements can result in the delivery of a product

- with unacceptable defect content, or

- at a later than planned ship date,

depending upon the system testing schedule, defect removal resources, and the impact of the enhancements on defect activity. Forbidding late-arriving functional enhancements can result in the delivery of a product

- lacking functionality required by its market.

Note that these ills can occur in combination; that is, a particularly bad decision might result in shipment of a product with unacceptable defect content, at a later than planned date, and with functionality required by its market so flawed as to be essentially missing.

Since shipped defect content and system test completion dates indicate the effectiveness of system testing, allowing late-arriving functional enhancements increases risk for the system test group. Since the presence or absence of functionally competitive product features indicates the effectiveness of market planning, forbidding late-arriving functional enhancements increases risk for the market planning group. Thus, people employed in different groups often take opposing sides in debates over late-arriving functional enhancements. Still decisions to allow or forbid late-arriving functional enhancements affect product quality, and thus, customer satisfaction, and thus, the profitability of the development effort. Thus, engineers should base these decisions on objective measures quantifying the impact of functional enhancement on product quality.

Recently, researchers have reported a relationship between functional enhancement and defect distribution [64, 63, 41]. Defects in enhanced program modules often concentrate within the modified code [63]. In this study, we exploit this relationship to produce a model for predicting enhancement induced defect activity. Sets of enhancement and defect measures collected during the enhancement of a commercial programming language processing utility provide data for analysis. A canonical model expresses the relationship between these sets as a relationship between two multidimensional latent variables: one representing enhancement activity, the other defect activity. Regressing the first dimension of defect activity on the first dimension of enhancement activity yields a model predicting enhancement induced defect activity. Analysis of the predictive quality of this model demonstrates that its predictions are of sufficient quality to support allow-or-forbid decisions regarding late-arriving functional enhancements. Those functional enhancements that cause more defect activity present greater risk of increasing product defect content and extending product ship dates. Predictions of enhancement induced defect activity offer leading indications of these risks to product quality. Further, for allowed functional enhancements, the predictions offer the system test group guidance in applying test resources.

6.1 The Modeling Methodology

In this study, we model the relationship between functional enhancement activity and defect activity. Both of these activities have multiple indicators. For

example, both the number of enhancements and the extent of change required to implement these enhancements indicate enhancement activity. Similarly, both the number of defects and the extent of change required to remove these defects indicate defect activity. Thus, the relationship between these activities is a relationship between two sets of indicators, one set indicating functional enhancement activity, the other defect activity. Canonical correlation analysis expresses the relationship between these sets as a relationship between two multidimensional latent variables. For each dimension, this expression gives two weighted aggregates of indicators, one for each set, each producing a distribution having zero mean and unit variance, such that

- within each dimension, the across-set aggregates correlate maximally, and
- within each set, the across-dimension aggregates are uncorrelated.

Each within-set aggregate evaluates a dimension of the latent variable indicated by this set. The latent variables at each dimension of canonical correlation describe a dimension of relationship between the analyzed sets. In this study, the latent variables of the canonical model represent enhancement and defect activity. A strong dimensional relationship between these variables is promising since there is benefit in predictions of enhancement induced defect activity, and each dimension having a strong relationship could yield a simple linear regression model giving these predictions. We find a strong correlation along the first dimension of the canonical model, and fit a simple linear regression model between the latent variables at

this dimension, with enhancement and defect activity serving, respectively, as the independent and dependent variables. The resulting model produces high-quality predictions of enhancement induced defect activity.

Refer to Chapter 3 for overviews of canonical correlation analysis and regression analysis. For complete treatments of these subjects refer to [43].

6.2 A Model of Enhancement and Defect Activity Interaction

In this section, we apply canonical correlation analysis to investigate the relationship between enhancement activity and defect activity during the system test phase of a commercial product. Then, using the results of the canonical model, we apply regression analysis to fit a simple linear regression model predicting enhancement induced defect activity.

While the methodology employed would be equally useful in a wide range of software development environments, we stress that the specific predictive model developed in this section is intended for application during the next iteration of the same development process, in production of the next release of the modeled product. Further, in this model application, we assume that the same key people implement the software changes required to produce the next release, and that the development environment remains unchanged. With care, engineers can loosen these assumptions. For example, if the assignment of key people changes, one should expect predictive quality to degrade in relation to the differences this introduces in programming skill level and product understanding.

The remaining text of this section is arranged as follows. Section 6.2.1 describes the product under study and the measures, or indicators, of enhancement and defect activity. Section 6.2.2 describes and interprets the canonical model. And finally, Section 6.2.3 describes the regression model.

6.2.1 The Enhancement and Defect Measures

The product development effort that yielded the data under study provided functional enhancements to a commercial programming language processing utility (LPU). Responsibility for the design and implementation of these enhancements rested primarily with one software engineer. The same engineer supported an independently managed system testing organization while this organization exposed the enhanced product to system test cases. This engineer also supported field testing at a selection of customer sites. The source code defining LPU is primarily C although some support code is assembly language. This study considers the C code; thus, in this study, *modules* are C functions. LPU is composed of 29 source files defining 369 C functions.

The following five measures, evaluated for each of the 369 LPU functions *on entry to* the system test phase, provide a cross-section of enhancement data for the canonical model: E, A_e, A'_e, C_e, and R_e. It is important to note that these enhancement measures are known before functional testing of the enhancements can begin since the changed code provides the functionality under functional test. We standardize each measure to zero mean and unit variance.

The following four measures, evaluated for each of the 369 LPU functions *on exit from* the system test phase, provide a cross-section of defect data for the canonical model: D, A_d, C_d, and R_d. Again, we standardize each measure to zero mean and unit variance.

Note that other selections of measures could serve to quantify enhancement activity, and other selections of measures could serve to quantify defect activity. Our goal in this section, is to present a model predicting enhancement induced defect activity. We do not intend to justify the use of any particular selections of measures.

Data splitting allows us to quantify the predictive quality of the predictive model. Applying this technique, we randomly partitioned the LPU functions into two subsets: a subset for developing models, and a subset for quantifying predictive quality. The subset for model development contains *fitting* data: standardized enhancement and defect measures for 246 LPU functions. The subset for quantifying predictive quality contains *testing* data: standardized enhancement and defect measures for 123 LPU functions. The testing data do not contribute to model development, and thus, application of a fitted model to the testing data gives an indication of the model's predictive quality.

6.2.2 Interpreting the Canonical Model

Canonical correlation analysis of the fitting data set yielded the canonical model, $M_{6.1}$. Figure 6.1 gives the path diagram for this model. This figure shows the

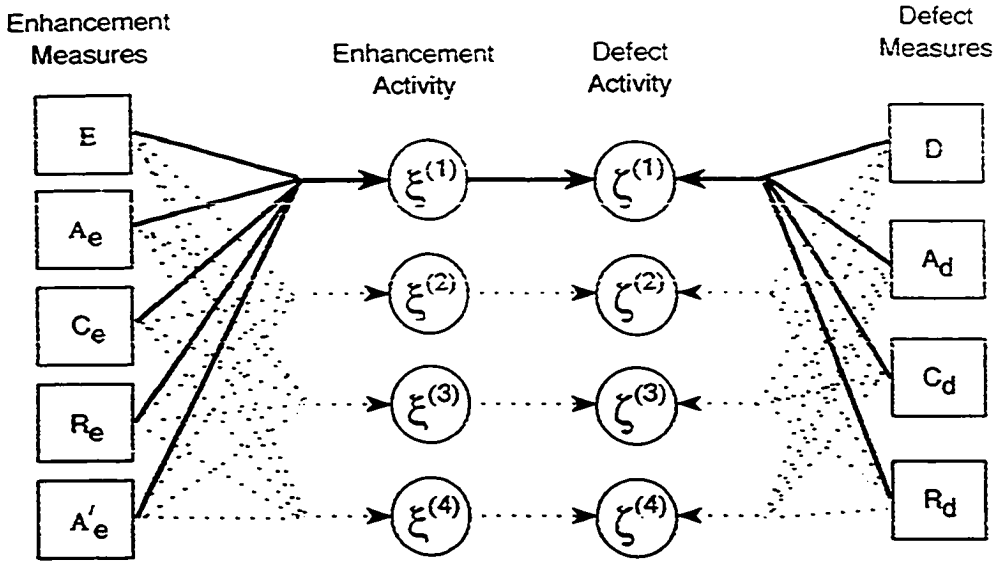

Figure 6.1: Model $M_{6.1}$ Path Diagram

five enhancement and the four defect measures that indicate, respectively, enhancement activity and defect activity. Both enhancement activity and defect activity have four dimensions. The directed paths from each dimension of enhancement activity to the corresponding dimension of defect activity represent the causal influence of enhancement activity on defect activity.

We select the first dimension of canonical correlation for interpretation. Thus, Figure 6.1 differentiates this dimension. Two considerations led to this selection.

Canonical Dimension	Canonical Correlation	Proportion of Variance	Cumulative Proportion
1	0.855	0.905	0.905
2	0.424	0.073	0.978
3	0.224	0.018	0.996
4	0.115	0.004	1.000

Table 6.1: Model $M_{6.1}$ Canonical Correlations

Of the four dimensions, the first accounts for about 90% of the overall variance explained by the model, the second about 7%, and the remaining two combined about 3%. Their relative lack of explained variance excludes the last two dimensions. While the second dimension might explain enough variation for consideration, its relative lack of explained variance along with its moderate canonical correlation, about 42%, suggest that it is not worthy of interpretation. The selected dimension has a strong canonical correlations, 85.5%. Table 6.1 summarizes this information.

In interpreting the first dimension of canonical correlation, consider the loadings of the latent variables with the manifest variables. Table 6.2 gives these loadings along with the weights that form the inner relations. That is,

$$\xi^{(1)} = -0.1831E + 0.1229A_e + 1.0026C_e + 0.0587R_e + 0.0669A'_e \qquad (6.1)$$

evaluates the first dimension of enhancement activity, and, for example, this latent

variable has a loading of 98.34% on C_e. Similarly,

$$\zeta^{(1)} = -0.1529D - 0.0819A_d + 1.0920C_d - 0.0178R_d \tag{6.2}$$

evaluates the first dimension of defect activity, and this latent variable has a loading of 98.74% on C_d. As shown in Table 6.2, the first dimension of enhancement activity loads strongly on A_e and C_e, and moderately on E. The first dimension of defect activity loads strongly on C_d, and moderately on D. Thus, the first dimension of correlation relates the number of enhancements and the number of noncomment lines added and changed to provide these enhancements, to the number of defects and number of noncomment lines changed to remove these defects.

6.2.3 The Regression Model

From the canonical correlation analysis of the previous section, we have two related latent variables, $\xi^{(1)}$ and $\zeta^{(1)}$, representing a dimension of, respectively, enhancement activity and defect activity. We wish to predict defect activity based upon enhancement activity. Thus we apply Equations 6.1 and 6.2 to evaluate, respectively, $\xi^{(1)}$ and $\zeta^{(1)}$ for each of the functions in the fitting data set. With these variables we fit a regression model, $\xi^{(1)}$ serving as the independent variable, $\zeta^{(1)}$ as the dependent variable. This yielded model $M_{6.2}$,

$$\hat{\zeta^{(1)}} = 0.80\xi^{(1)} \tag{6.3}$$

$\hat{\zeta^{(1)}}$ giving a prediction of $\zeta^{(1)}$. Since latent variables are distributed with zero mean, the intercept is zero.

	Dimension 1	
	Weights	Loadings
Enhancement Activity (ξ)		
E (x_1)	-0.1831	0.4374
A_e (x_2)	0.1229	0.7155
C_e (x_3)	1.0026	0.9834
R_e (x_4)	0.0587	0.0298
A'_e (x_5)	0.0669	0.0665
Defect Activity (ζ)		
D (y_1)	-0.1529	0.5107
A_d (y_2)	-0.0819	0.0064
C_d (y_3)	1.0920	0.9874
R_d (y_3)	-0.0178	-0.0201

Table 6.2: Model $M_{6.1}$ Canonical Weights and Loadings

Table 6.3 reports the R^2_{fit}, the average PRESS, and the R^2_{pred} corresponding to this model fit, $R^2_{fit} = 0.70$ indicating satisfactory fit, $\overline{\text{PRESS}} = 0.26$ and $R^2_{pred} = 0.67$ suggesting satisfactory predictive quality. Table 6.3 also reports AAE = 0.28, an indication of satisfactory predictive quality on model application to the testing data set. In this application, given the data for each of the observations in the testing data set, Equations 6.1 and 6.2 yield testing data set values of, respectively, $\xi^{(1)}$ and $\zeta^{(1)}$. Given the testing data set $\xi^{(1)}$ values, the model, Equation 6.3, yields predictions, $\hat{\zeta^{(1)}}$. These predictions, along with the testing data set $\zeta^{(1)}$ values, yield the absolute error in prediction for each testing data set observation, $|\zeta^{(1)} - \hat{\zeta^{(1)}}|$. The standard deviation of absolute error for this model application, 0.50, demonstrates that wide variations in predictive error are not typical of the model's predictions for observations in the testing data set. The demonstrated model qualities are consistent with models that are useful as predictive tools.

The risk of late-arriving functional enhancements varies with the system testing schedule, defect removal resources, and the impact of the enhancements on defect activity. With knowledge of their defect removal resources, software engineers can use high-quality predictions of enhancement induced defect activity to demonstrate the impact of functional enhancements on the system testing schedule. In this way, allow-or-forbid decisions regarding late-arriving functional enhancements reduce to decisions regarding the relative importance of the marketing window, product defect content, and product functionality.

Quality Statistic			
R^2_{fit}	$\overline{\text{PRESS}}$	R^2_{pred}	AAE
0.70	0.26	0.67	0.28

Table 6.3: Model $M_{6.2}$ Model Quality Statistics

6.3 Conclusions

In this chapter, we exploited the relationship between enhancement activity and defect distribution to produce a model for predicting enhancement induced defect activity. We achieved this in two steps. First, we applied canonical correlation analysis to model the relationship between a set of enhancement activity indicators and a set of defect activity indicators. This analysis isolated one dimension of this relationship having strong correlation. Then, we modeled the relationship between the latent variables at this dimension as a simple linear regression. This model demonstrated predictive quality sufficient for application as a software engineering tool.

The predictive model considers enhancement activity as the sole source of variation in defect activity. Other sources of variation, such as differences in the product to be enhanced, in programmer skill level, in programmer product understanding, and in the software development process, were not modeled, but remained constant throughout the development effort that yielded the modeled data. In software development organizations that iteratively produce enhanced releases of the same product, software engineers can control these unmodeled sources of variation

when applying predictive models.

Such models are intended for predicting defect activity in the program modules that result from the next iteration of the same development process, in production of the next release of the modeled product, with the same key people implementing the software changes that introduce functional enhancements. In this application, the unmodeled sources of variation should contribute little to the distribution of defects. However, we expect the learning experience of the previous release to result in some change in programming skill level and product understanding. These changes will vary with individuals, the scope of their responsibility, and the time between releases. Care in changing the assignments of key people could control several sources of variation in software development organizations that lack stable assignments. The modeling technique scales to larger development efforts involving several key people by either developing unique models for each area of responsibility, or adding independent variables that account for variance introduced by differing skill and understanding levels.

Finally, consider this study in relation to others producing discriminant or predictive models for software engineering applications [18, 4, 62, 5, 71]. Each of the cited studies relates aspects of source code complexity to either software change or defect measures. This study departs, in two major ways, from these studies. First, we relate enhancement activity to change activity with no consideration of source code complexity. Many software development organizations do not collect source

code measures. This study demonstrates that many of these organizations can benefit from data that they do collect. Further, the results of this study, along with previous results demonstrating effective models based upon source code complexity [62], suggest modeling based upon both sources of variation: enhancement activity and source code complexity. This remains as future work.

Second, in software engineering models, both independent and dependent variables tend to represent complex concepts each requiring multiple indicators. Previous research has demonstrated discriminant and predictive models based upon multiple input indicators, but either discriminating based upon, or predicting a simple output [62, 5, 18, 71]. This study demonstrates a model relating enhancement activity with defect activity, representing each of these activities with multiple indicators. Applying canonical correlation analysis, the linear combinations of these indicators model relationships among the set of enhancement activity indicators, among the set of defect activity indicators, and across these sets of indicators. Thus, empirical methods yield the linear combinations of indicators representing both the independent and the dependent variable.

Chapter 7

CONTRIBUTIONS AND FUTURE RESEARCH

7.1 Contributions

Many results have demonstrated the utility of software quality modeling [72, 19, 53, 31, 73, 74, 22, 75, 76]. Other results have established that the relationship between software complexity and software quality is between two multidimensional concepts [77, 20, 25]. This work applied multivariate techniques to build upon this foundation.

Application of Krzanowski's method for between-groups comparison of principal components revealed that software engineers should not expect source-code-measure principal components stability across distinct products developed by distinct organizations. This application also revealed that source-code-measure principal components stability is more likely across the revisions of a single software product throughout its lifecycle, and across distinct products developed within the same organization. Further, this application established that source-code-measure principal components instability can degrade the predictive quality of software quality models.

Together, these results, that source-code-measure principal components are not necessarily stable, and that instability in these principal components can degrade the predictive quality of software quality models, demonstrate the importance of analytically quantifying the source-code-measure principal components stability in software quality model applications. In application of these models, the source-code-measure principal components of the product under current development should be similar to those of the product that provided the historical data for fitting the model. Krzanowski's method for between-groups comparison of principal components can serve software engineers as a tool for quantifying source-code-measure principal components stability across products, and thus, serve as a guide in the selection of an appropriate historical data set for modeling current development.

Since both software complexity and software quality are multidimensional concepts, multiple measures indicate both of these concepts. Neither simple nor multiple linear regression models can capture the interactions within a set of indicators for software complexity, the interactions within a set of indicators for software quality, and the interactions across these sets of indicators. This work demonstrated the application of canonical correlation analysis to sets of software engineering measures that serve to indicate related multidimensional concepts. Three canonical models revealed the relationship between software complexity and defect activity, each model indicating defect activity with a unique selection of process measures. The analyzed data supports the hypothesis that source code complexity exerts a

causal influence on the defect activity experienced during system testing. Further, for the analyzed data, two subsets of product measures displayed different relationships with process activity, one showing a causal influence on design change activity that results in defects, the other showing a direct causal influence on defects. Defects having less impact on the system test process associated with design change activity that occurred during the system test phase, while those having more impact associated with source code complexity at entry to the system test phase.

The results of the canonical analysis of software complexity and defect activity revealed the causal influence of design change activity on defect activity. Some design change activity corrects design defects; some provides functional enhancements. For software development organizations that iteratively produce enhanced versions of products, the relationship between enhancement activity and defect activity is important. This work extended discriminant models for identifying high-risk modules to include enhancement activity as a source of defect activity variation. This extension demonstrated that, in software quality modeling, the relationship between enhancement activity and defect activity can be more important than the relationship between code complexity and defect activity. Pursuing the importance of enhancement activity this work demonstrated an empirical model of enhancement induced defect activity.

7.2 Future Research

The applications of multivariate modeling techniques suggested several areas for continued research.

- The critical angle method indicates the degree of similarity between critical angle rotations of principal components. This method provides an analytical tool for choosing an historical data set appropriate for modeling current development. In this choice, the similarity of the components included by model selection is important; the similarity of the components excluded by model selection is not. Thus, the power of the critical angle method as a tool for choosing a historical data set increases with models selected from critical angle rather than varimax principal components. That is, a high critical angle sum does not imply that the model is inappropriate; a high critical angle sum among components included by model selection does. Since the critical angles do not map to components of the varimax rotation, the accuracy of the method is not enhanced by knowledge of the selected components for models selected from these principal components. For models selected from the principal components of critical angle rotations, this knowledge could enhance the accuracy of the method.

- The canonical modeling results suggest that models involving more than two latent variables will provide more insight into the relationships among software engineering measures than canonical models can. Soft modeling allows

more than two latent variables. This methodology is intended for modeling complex interdisciplinary systems having many variables and little established theory. Further, it incorporates parameter estimation techniques relying on no distributional assumptions. General soft models of the software development process could be appropriate for both exploratory analysis and prediction of future performance.

- Consideration of the discriminant modeling results led us to hypothesize that the level of routines in the calling hierarchy introduces variance in defect distribution. Further, we noted that the impact of a defect is an indicator of the risk that it presents, and thus consideration of defect classifications could improve discriminant results.

Pursuing these points remains as future work.

REFERENCES

[1] S. Lawrence-Pfleeger, N. E. Fenton, and S. Page. Examples of measurement for software engineering standards evaluation. To appear in *IEEE Computer*, September 1994.

[2] R. S. Pressman. *Software Engineering: A Practitioner's Approach.* McGraw-Hill, New York, 1992.

[3] J. D. Musa, A. Iannino, and K. Okumoto. *Software Reliability: Measurement, Prediction, Application.* McGraw-Hill, Inc., New York, 1987.

[4] S. M. Henry and S. Wake. Predicting maintainability with software quality metrics. *Software Maintenance: Research and Practice*, 3:129–143, 1991.

[5] T. M. Khoshgoftaar, D. L. Lanning, and A. S. Pandya. A comparative study of pattern recognition techniques for quality evaluation of telecommunications software. *IEEE Journal of Selected Areas in Communications*, 12(2):279–291, February 1994.

[6] M. H. Halstead. *Elements of Software Science.* Elsevier North-Holland, New York, 1977.

[7] T. J. McCabe. A complexity metric. *IEEE Transactions on Software Engineering*, 2(4):308–320, December 1976.

[8] W. J. Hansen. Measurement of program complexity by the pair. *SIGPLAN Notices*, 12(10):61–64, October 1978.

[9] A. L. Baker and S. H. Zweben. A comparison of measures of control flow complexity. *IEEE Transactions on Software Engineering*, 6(6):506–512, November 1980.

[10] B. Ramamurthy and A. Melton. A synthesis of software science measures and the cyclomatic number. *IEEE Transactions on Software Engineering*, 14(8):1116–1121, August 1988.

[11] J. C. Munson and T. M. Khoshgoftaar. Applications of a relative complexity metric for software project management. *Journal of Systems and Software*, 12:283–291, 1990.

[12] T. M. Khoshgoftaar and J. C. Munson. Applications of a relative complexity metric for predicting source code complexity at the design phase. In *Proceedings of the Second Software Engineering Research Forum*, pages 191–198. Melbourne, Florida, November 1992.

[13] T. M. Khoshgoftaar and J. C. Munson. The use of a relative complexity metric to compare software designs: An empirical investigation. In *Proceedings of the ISMM International Conference on Intelligent Distributed Processing*, pages 69–72, Ft. Lauderdale, FL., 1989.

[14] A. Melton, D. A. Gustafson, J. M. Bieman, and A. L. Baker. A mathematical perspective for software measure research. *Software Engineering Journal*, 5(5):246–254, September 1990.

[15] B. Curtis. Conceptual issues in software metrics. In *Proceedings of the Nineteenth Hawaii International Conference on Systems Sciences*, pages 154–157, January 1986.

[16] V. Cote, P. Bourque, S. Oligny, and N. Rivard. Software metrics: An overview of recent results. *Journal of Systems and Software*, 8(2):121–131, March 1988.

[17] L. C. Briand, V. R. Basili, and C. J. Hetmanski. Providing an empirical basis for optimizing the verification and testing phases of software development. In *Proceedings of the Third International Symposium on Software Reliability Engineering*, pages 329–338, Research Triangle Park, North Carolina, October 1992.

[18] L. C. Briand and V. R. Basili. A classification procedure for the effective management of changes during the maintenance process. In *Proceedings of the 1993 IEEE International Conference on Software Maintenance*, pages 328–336, Orlando, Florida, November 1992.

[19] G. K. Gill and C. F. Kemerer. Cyclomatic complexity density and software maintenance productivity. *IEEE Transactions on Software Engineering*, 17(12):1284–1288, December 1991.

[20] T. M. Khoshgoftaar, J. C. Munson, B. B. Bhattacharya, and G. D. Richardson. Predictive modeling techniques of software quality from software measures. *IEEE Transactions on Software Engineering*, 18(11):979–987, November 1992.

[21] T. M. Khoshgoftaar, D. L. Lanning, and A. S. Pandya. A neural network modeling methodology for the detection of high-risk programs. In *Proceedings of the Fourth International Symposium on Software Reliability Engineering*, pages 302–309, Denver, Colorado, November 1993.

[22] R. K. Lind and K. Vairavan. An experimental investigation of software metrics and their relationship to software development effort. *IEEE Transactions on Software Engineering*, 15(5):649–651, May 1989.

[23] A. A. Porter and R. W. Selby. Empirically guided software development using metric-based classification trees. *IEEE Software*, 7(2):46–54, March 1990.

[24] R. W. Selby and A. A. Porter. Learning from example: Generation and evaluation of decision trees for software resource analysis. *IEEE Transactions on Software Engineering*, 14(12):1743–1757, December 1988.

[25] T. M. Khoshgoftaar and J. C. Munson. Predicting software development errors using complexity metrics. *IEEE Journal of Selected Areas in Communications*, 8(2):253–261, February 1990.

[26] J. C. Munson and T. M. Khoshgoftaar. The detection of fault-prone programs. *IEEE Transactions on Software Engineering*, 18(5):423–433, May 1992.

[27] N. E. Fenton. Software measurement: A necessary scientific basis. *IEEE Transactions on Software Engineering*, pages 199–206, March 1994.

[28] P. F. Velleman and L. Wilkinson. Nominal, ordinal, interval, and ratio typologies are misleading. *The American Statistician*, 47(1):65–72, February 1993.

[29] H. Wold. *Systems Under Indirect Observation*, volume 2, chapter Soft Modeling: The Basic Design and Some Extensions, pages 1–54. North–Holland Publishing Company, Amsterdam, Netherlands, 1982.

[30] V. R. Basili and R. W. Selby. Calculation and use of an environment characteristic software metric set. In *Proceedings of the Eigth International Conference on Software Engineering*, pages 386–391, London, UK, August 1985.

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

[32] B. H. Yin and J. W. Winchester. The establishment and use of measures to evaluate the quality of software design. *Software Engineering Notes*, 3:45–52, 1978.

[33] J. K. Navlakha. Measuring the effect of external and internal interface on software development. In *Proceedings of the 20th Annual Hawaii International Conference on System Sciences*, pages 127–136, 1987.

[34] H. A. Jensen and K. Vairavan. An experimental study of software metrics for real-time software. *IEEE Transactions on Software Engineering*, SE-11(2):231–234, February 1985.

[35] J. C. Munson and T. M. Khoshgoftaar. Measurement of data structure complexity. *Journal of Systems and Software*, 12(3):217–225, March 1993.

[36] M. R. Woodward, M. A. Hennell, and D. Hedley. A measure of control flow complexity in program text. *IEEE Transactions on Software Engineering*, SE-5(1):45–50, January 1979.

[37] W. Harrison. An entropy-based measure of software complexity. *IEEE Transactions on Software Engineering*, 18(11):1025–1029, November 1992.

[38] N. F. Schneidewind. Methodology for validating software metrics. *IEEE Transactions on Software Engineering*, 18(5):410–421, May 1992.

[39] N. E. Fenton. *Software Metrics: A Rigorous Approach.* Chapman and Hall, New York, 1991.

[40] J. C. Laprie. For a product-in-a-process approach to software reliability evaluation. In *Proceedings of the Third International Symposium on Software Reliability Engineering*, pages 134–139, Research Triangle Park, North Carolina, October 1992.

[41] D. L. Lanning and T. M. Khoshgoftaar. Canonical modeling of software engineering measures. Technical Report TR-CSE-94-10, The Department of Computer Science and Engineering, Florida Atlantic University, March 1994.

[42] D. L. Lanning and T. M. Khoshgoftaar. Modeling the relationship between source code complexity and maintenance difficulty. To appear in *IEEE Computer*, September 1994.

[43] W. R. Dillon and M. Goldstein. *Multivariate Analysis: Methods and Applications*. Wiley, New York, 1984.

[44] A. Myrvold. Data analysis for software metrics. *Journal of Systems and Software*, 12(3):271–275, July 1990.

[45] P. N. Robillard and D. Coupal. Study on the normality of metric distributions. In *Proceedings of the Annual Oregon Workshop on Software Metrics*, Silver Falls, Oregon, March 1991.

[46] W. J. Krzanowski. Between-Groups comparison of principal components. *Journal of the American Statistical Association*, 74(367):703–707, September 1979.

[47] W. J. Krzanowski. *Principles of Multivariate Analysis: A User's Perspective*. Oxford University Press, New York, 1988.

[48] T. M. Khoshgoftaar and D. L. Lanning. Are the principal components of software complexity data stable across software products? To appear in the *Proceedings of the Second IEEE International Software Metrics Symposium*, London, England, October 1994.

[49] N. F. Schneidewind. Minimizing risk in applying metrics on multiple projects. In *The Proceedings of the Third International Software Reliability Engineering*, pages 173–182, Research Triangle Park, NC, October 1992.

[50] T. M. Khoshgoftaar and D. L. Lanning. On the impact of software product dissimilarity on software quality models. To appear in the *Proceedings of the Fifth International Symposium on Software Reliability Engineering*, Monterey, California, November 1994.

[51] R. H. Myers. *Classical and Modern Regression with Applications.* Duxbury Press, Boston, MA, 1990.

[52] D. H. Kitson and S. M. Masters. An analysis of SEI software process assessment results: 1987-1991. In *Proceedings of the* 15^{th} *International Conference on Software Engineering*, pages 68–77, Baltimore, Maryland, 1993.

[53] S. M. Henry and C. Selig. Predicting source-code complexity at the design stage. *IEEE Software*, 7(2):36–44, March 1990.

[54] T. M. Khoshgoftaar, J. C. Munson, and D. L. Lanning. Alternative approaches for the use of metrics to order programs by complexity. *Journal of Systems and Software*, 24(3):211–221, March 1994.

[55] W. M. Zage and D. M. Zage. Evaluating design metrics on large-scale software. *IEEE Software*, 10(4):75–80, July 1993.

[56] D. L. Lanning and T. M. Khoshgoftaar. Canonical modeling of software complexity and fault correction activity. To appear in the *Proceedings of the 1994 IEEE International Conference on Software Maintenance*, Victoria, British Columbia, Canada, September 1994.

[57] M. Neil and R. Bache. Data linkage maps. *Software Maintenance: Research and Practice*, 5:155–164, 1993.

[58] H. Hotelling. Relations between two sets of variables. *Biometrica*, 28:321–377, 1936.

[59] H. Wold. *Research Papers in Statistics: Festschrift for J. Neyman*, chapter Nonlinear Estimation by Iterative Least Squares Procedures, pages 411–444. Wiley, New York, 1966.

[60] A. E. Boardman, B. S. Hui, and H. Wold. The partial least squares–fix point method of estimating interdependent systems with latent variables. *Communications in Statistics—Theory and Methods*, 10(7):613–639, 1981.

[61] W. Harrison and C. Cook. Insights on improving the maintenance process through software measurement. In *Proceedings of the 1993 IEEE International*

Conference on Software Maintenance, pages 37–45, San Diego, CA, November 1990.

[62] T. M. Khoshgoftaar, J. C. Munson, and D. L. Lanning. A comparative study of predictive models for program changes during system testing and maintenance. In *Proceedings of the 1993 IEEE International Conference on Software Maintenance*, pages 72–79, Montreal, Quebec, Canada, September 1993.

[63] K. H. Moller and D. J. Paulish. An empirical investigation of software fault distribution. In *Proceedings of the First IEEE International Software Metrics Symposium*, pages 82–90, Baltimore, MD, May 1993.

[64] G. S. Cherf. An investigation of the maintenance and support characteristics of commercial software. *Software Quality Journal*, 1(3):147–158, September 1992.

[65] G. A. F. Seber. *Multivariate Observations*. John Wiley and Sons, New York, 1984.

[66] T. M. Khoshgoftaar and D. L. Lanning. A study on the stability of principal components of software complexity data. Technical Report TR-CSE-93-57, The Department of Computer Science and Engineering, Florida Atlantic University, October 1993.

[67] G. LeGall, M. F. Adam, H. Derriennic, B. Moreau, and N. Valette. Studies on measuring software. *IEEE Journal of Selected Areas in Communications*, 8(2):234–245, 1990.

[68] F. B. Bastani and S. S. Iyenger. The effect of data structures on the logical complexity of programs. *Communications of the ACM*, 30:250–259, 1987.

[69] R. Chillarege, I. Bhandari, J. Chaar, M. Halliday, D. Moebus, B. Ray, and M. Y. Wong. Orthogonal defect classification—a concept for in-process measurements. *IEEE Transactions on Software Engineering*, 18(11):943–956, November 1992.

[70] M. Sullivan and R. Chillarege. Software defects and their impact on system availability—A study of field failures in operating systems. In 21^{nd} *International Symposium on Fault-Tolerant Computing*, pages 1–8, Montreal, Canada, June 1991.

[71] R. W. Selby and V. R. Basili. Analyzing error-prone system structure. *IEEE Transactions on Software Engineering*, 17(2):141–152, February 1991.

[72] B. T. Compton and C. Withrow. Prediction and control of ada software defects. *Journal of Systems and Software*, 12(3):199–207, July 1990.

[73] B. Kitchenham and L. Pickard. Towards a constructive quality model. *Software Engineering Journal*, pages 114–119, July 1987.

[74] K. S. Lew, T. S. Dillon, and K. E. Forward. Software complexity and its impact on software reliability. *IEEE Transactions on Software Engineering*, 14(11):1645–1655, November 1988.

[75] N. F. Schneidewind. Validating software metrics: Producing quality discriminators. In *Proceedings of the Second International Symposium on Software Reliability Engineering*, pages 225–232, Austin, TX, May 1991.

[76] J. Tian, A. Porter, and M. V. Zelkowitz. An improved classification tree analysis of high cost modules based upon an axiomatic definition of complexity. In *Proceedings of the Third International Symposium on Software Reliability Engineering*, pages 164–172, Research Triangle Park, North Carolina, October 1992.

[77] N. F. Schneidewind. Report on the IEEE standard for a software quality metrics methodology. In *Proceedings of the 1993 IEEE International Conference on Software Maintenance*, pages 104–106, Montreal, Quebec, Canada, September 1993.

VITA

David Lee Lanning was born in Grand Rapids, Michigan on January 6, 1957. He completed his primary and secondary schooling in the Grand Rapids area, graduating from Roger's High School in 1975. Since that time, he has gained experience in many facets of the computer industry, successively maintaining general purpose computer systems employed aboard ballistic missile submarines, enhancing and developing business applications for wholesale distributors, teaching computer programming classes, and developing personal computer based operating system components. During this time he also received several degrees including an A.S. with highest honor in Business Administration from Davenport College, Grand Rapids, Michigan, in 1983, a B.S., Magna Cum Laude, in Mathematics and Computer Science from Grand Valley State University, Allendale, Michigan in 1985, and an M.S. in Computer and Information Science from the Ohio State University in 1988. David is currently employed as a software engineer by the IBM Corporation, and is a Ph.D. candidate studying computer science at Florida Atlantic University, Boca Raton, Florida.