USING THE FAULT INDEX SELECTION PROCESS FOR LARGE SOFTWARE SYSTEMS TO GUIDE AND IMPROVE CODE INSPECTION EFFECTIVENESS

THESIS

Presented to the Graduate Council
of Texas State University-San Marcos
in Partial Fulfillment
of the Requirements

for the Degree

Master of SCIENCE

by

Steven C. Oakes, B.S.

San Marcos, Texas
August 2005

ACKNOWLEDGEMENTS

I would like to thank my thesis committee for all of their helpful discussions and guidance during the research leading up to this thesis. I would especially like to thank my advisor, Dr. Greg Hall, both for the motivation for the project itself, and for helping keep it on track. I want to also thank Don Shafer for all of his priceless teachings in the field of Software Engineering. His wisdom and many years of experience helped me to grow professionally in my career at IBM.

I would also like to thank Dr. Kaikhah and Professor Davis for their tremendous patience and assistance in my early computer science education. In addition, I would like to acknowledge the entire faculty of the Computer Science department at Texas State University-San Marcos for their kind support, and for making the classes so much fun. My time at Texas State has been an invaluable and extraordinary journey in learning! Finally, I would like to thank my mentor George Stark. He has guided me through the true reality of the corporate world and has had a significant influence in my career.

This manuscript was submitted on August 20, 2005

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

USING THE FAULT INDEX SELECTION PROCESS FOR LARGE SOFTWARE SYSTEMS TO GUIDE AND IMPROVE CODE INSPECTION EFFECTIVENESS

by

Steven C Oakes, B.S.

Texas State University-San Marcos

August 2005

SUPERVISING PROFESSOR: GREGORY A. HALL

Software inspections are one of the most powerful error detection techniques in software development. Traditionally, experienced software

inspectors read the source code line-by-line, report the errors as they find them, and document comments and suggestions. Selecting code candidates for an inspection is a challenging and difficult task for the inspection teams due to the enormous size of a software system. It is practically impossible to inspect all of the code. Therefore, a surrogate measure to predict the amount of software errors that are in the system is needed and this measure is called the Fault Index. The Fault Index is a single numeric value calculated from statistical analysis of the variability of source code metrics. Utilizing a Fault Index to select code candidates prior to and during the inspection process increases the effectiveness of the inspection. In addition, incorporating the Fault Index to select code candidates is more efficient than inspectors' opinion. Finally, it is up to the inspection team members to inspect and remove errors from those potentially fault-prone components.

CHAPTER I

1.1 Introduction

Inspections are vital in both the success and quality of a delivered software product. This paper will investigate the impact and the potential usage of the Fault Index metric to guide inspections and its effectiveness given time and budget constraints. In addition, this paper will attempt to demonstrate a positive side effect of using the Fault Index. The Fault Index can potentially reduce both the amount of time and costs associated with software inspections and improve the overall quality and reliability of the software. Finding highly complex errors early in the software lifecycle has a positive and significant impact on both development and support costs as well as customer satisfaction.

The amount of time spent during software inspections is highly dependant on pre-inspection reviews and preparation. Typically, the team reviews the code prior to the formal inspection. A formal inspection is considered a strict formal process that has defined rules and guidelines. However, some studies indicate that pre-reviews are not performed at all (Buck, 1981). Many times, the inspectors who attend the inspections have briefly

reviewed the material, did not understand the material, or never reviewed it at all.

In today's software development market, high demands are placed on development organizations to satisfy customers through timely and inexpensive delivery of a high quality product. Many development organizations are competing in a fast-paced technological market and their success greatly depends on delivering high quality software on time, and within budget. Due to time and market demands however, many development organizations are sacrificing quality. This trend is increasing despite significant gains from the efforts of research and development to improve software engineering techniques.

One of the many key process improvement activities recommended by CMMI® is software inspections. Software inspections provide a vehicle in which to detect potential problems earlier in the life cycle. Detecting and ultimately removing injected errors early in the life cycle translates into reduced effort and costs later in the life cycle. In order to perform adequate and efficient software inspections effectively, teams of people must review the code, submit findings, and finally perform the inspection. If the inspected product contains a million lines of code, it is highly probable that the inspection will be ineffective and time consuming. Often, complete inspections may not be practical, in which case, we need the

ability to make informed decisions concerning which portions of the system need inspection, and which may forgo inspection.

An approach to this problem is to identify and collect some static measures from the source code to enhance and use in the inspection process. There are various significant types of static metrics that could be applied to identify high-risk modules that a project might choose from (Khoshgoftaar, Munson, Lanning, 1994). When selecting a static analysis tool, the tool must have the ability to calculate some primitive static metrics. Once this step is performed, a technique called principal component analysis (PCA) is conducted next on the output of the static analysis tool. When the principal component analysis is complete, identification of the Fault Index measure is performed. Finally, when the Fault Index is known, it now gives the inspection teams the ability to uniquely identify and select high-risk code inspection candidates. This process will not only facilitate and improve code coverage during the inspection, but will most likely expose those critical portions of the source code that need thorough inspection. In addition, by utilizing a Fault Index, the test teams will now have the capacity to perform additional types of testing, hence increasing the error discovery rate.

1.2 Research Objective

The goal of this project is to incorporate a Fault Index in the software inspection process. Today, many development teams do not know how to adequately select code candidates to inspect. This causes the teams to randomly select code modules that may or may not have critical potential faults. Code candidates selected based on the Fault Index will result in improved quality of the delivered software product. This is most significant because it can reduce test, support, and lifecycle costs, improve software reliability, reduce time to market, and eventually lead to increased customer satisfaction. To validate these findings, some historical project data will be collected for use as the baseline. Next, the teams will integrate and use the Fault Index and measure its impact on the development process. In parallel, the teams will collect various project data and track the projects through closure and delivery to the customer. This investigation will attempt to prove the effectiveness of using a Fault Index to select code inspection candidates that impacts software reliability, rework, overall costs, and time to market.

CHAPTER II

2.1 Background of Software Inspections

IBM created the software inspection process in 1972, for the dual purposes of improving software quality and increasing programmer productivity (Fagan, 1986). Its accelerating rate of adoption throughout the software development and maintenance industry is an acknowledgment of its effectiveness in meeting its goals (Fagan, 1986). Software inspection is a method of static testing to verify that software meets its requirements. It engages the developers and others in a formal process of investigation that usually detects more defects in the product, and at lower cost, than does machine testing. Users of the method report very significant improvements in quality that are accompanied by lower development costs and greatly reduced maintenance efforts. Software inspections examine the products of each major development activity, such as requirements specification, high-level design, detailed design, and coding (Fagan, 1976). Code inspections are a rigorous and strict form of peer review technique used for identification and removal of coding errors to improve productivity, manageability, and quality (Fagan, 1976; Freedman, Weinberg, 1990). Software reviews, or

inspections, are an important and widely used practice for the development of quality software (Fagan 1976, Gilb 1993, Humphrey 1995, Strauss & Ebenau 1994).

Most code inspections involve a prior stage of individual preparation (Fagan 1976, Gilb 1993, Freedman and Weinberg 1990, Strauss & Ebenau 1994). While multiple reviewers find more defects than an average individual reviewer does, they rarely (if ever) report all the defects lurking in the product (Land et al 1997a & 1997b). Often, inspection meetings fail to report defects detected in preparation by individual reviewers (Votta 1993). Where reviewers have individually searched for defects ahead of the meeting, they find few new ones at the inspection meeting (Porter and Votta 1994, Porter et al 1995). For these reasons, and in view of the resource and schedule costs of holding inspections, some reviewers have questioned the need to hold an inspection meeting (Votta 1993, Johnson & Tjahjono 1998). However, other findings have shown that excellent results have been obtained by both small and large organizations in all aspects of new development as well as in maintenance. There is some evidence that developers who participate in the inspection of their own product actually create fewer defects in future work. Because inspections formalize the development process, productivity and quality enhancing tools can be adopted more easily and rapidly.

2.1.1 Benefits in Defect Reduction, Prevention, and Cost Improvement

With the implementation of the inspection process by IBM, they achieved significant improvements in quality. IBM has nearly doubled the number of lines of code shipped for System/370 software products since 1976, while the number of defects per thousand lines of code has been reduced by *two-thirds.* Feedback from early MVS/XA and VM/SP Release 3 users indicates these products met and, in many cases, exceeded ever-increasing quality expectations (Fenton, 1984).

Observation of a small sample of programmers suggested that early experience gained from inspections caused programmers to reduce the number of defects that were injected in the design and code of programs created later during the same project. Preliminary analysis of a much larger study of data from recent inspections is providing similar results. As improvements are incorporated into everyday practice, it is probable that inspections will help bring further reductions in defect injection and detection rates. Additional reports showing that inspections improve quality *and* reduce costs follow. In all these cases, the cost of inspections is included in the project cost. Typically, all design and code inspection costs amount to 15% of project cost.

IBM and AETNA reported that software inspections found 82 and 93 percent, respectively, of all defects (that would cause malfunction) detected over the life cycle of the products. Other account contracts found over 50 percent of all defects by inspection. In similar reports, The Standard Bank of South Africa and American Express, although unable to use trained inspection moderators (and the former conducted only code inspections), yet both still managed to obtain outstanding results using inspections. The tremendous reduction in corrective maintenance at the Standard Bank of South Africa also brought impressive savings in life cycle costs (Graden, Horsley, and Pingel, 1986).

Naturally, a reduction in maintenance effort allows redirection of programmers to work off the application backlog, which is reputed to contain at least two years of work at most locations. Impressive cost savings and quality improvements have been realized by inspections. For a product of about 20 000 Lines of Code (LOC), R. Larson (Larson, 1975) reported that code inspections resulted in:

- Modification of approximately 30 percent of the functional matrices representing test coverage
- Detection of 176 major defects in the code (i.e., in 176 instances testing would have missed testing a critical function or tested it incorrectly)

- Produced a cost savings of more than 83 percent in programmer time by detecting the major defects by code inspection as opposed to finding them during functional variation testing

There are those who would use inspections whether or not they are cost justified for defect removal because of the nonquantifiable benefits the technique supplies toward improving the service provided to users and toward creating a more professional application development environment (Crossman, 1979).

Experience has shown that inspections have the effect of slightly front-end loading the commitment of people resources in development by adding to requirements and design, while greatly reducing the effort required during testing and for rework of design and code. The result is an overall *net* reduction in development resources and length of schedule. Figure 2.1 is a pictorial description of the familiar "snail" shaped curve of software development resources versus schedule, with and without inspections.

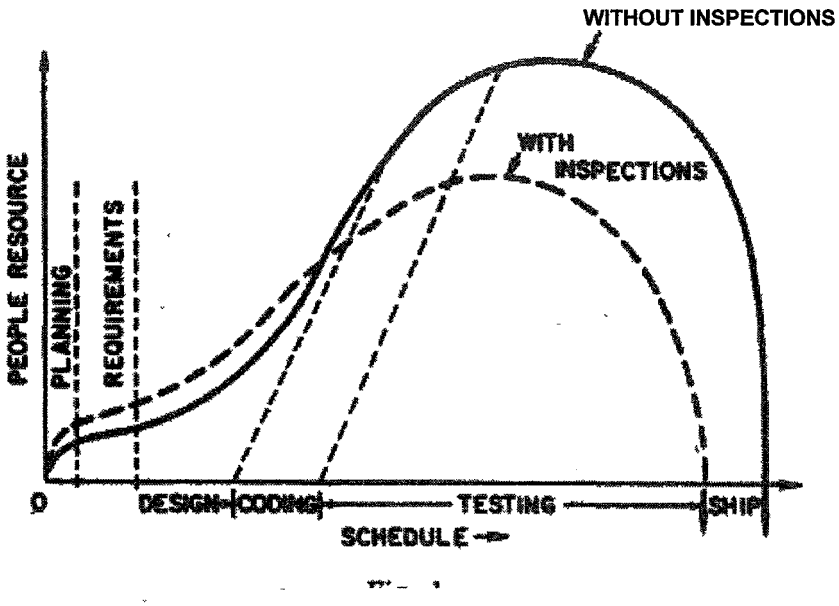

Figure 2.1: Software development Inspection resource curve

Recently, some researchers have argued that pre-inspection meetings may be worthwhile because they are effective at detecting false positives i.e. issues that are not true defects (Sauer et al 1996, Porter & Johnson 1997), and because they detect soft maintenance issues (Porter et al 1997). Experience suggests that, in some organizations, day-to-day pressures preclude effective individual preparation, the result being that an inspection meeting is essential as the only point at which defect detection occurs. While there continue to be good reasons for holding pre-inspection meetings, it is desirable to design the inspection process to be as effective as possible within acceptable cost constraints.

This paper reports the results of an experimental study of utilizing a Fault Index during inspections. The purpose of this research is to empirically demonstrate that using a Fault Index to select fault-prone components improves inspection defect detection performance. Less predictably, this study finds that utilizing a Fault Index to select component code candidates also results in the inspection being more effective at finding new defects not previously discovered by individual effort. The combined positive effects of the Fault Index strongly indicate that it is worth adopting. Further analysis provides a deeper understanding of how code candidate selections are based on principal component analysis findings.

2.1.2 The Software Inspection

The software inspection process is a set of operations occurring in a definite sequence that operates on a given input and converts it to a desired output. In this case, the input is developed source code and the output is discovered defects. Once this process is completed, an exit criterion must be satisfied in order to determine the success of executing this process.

The general purpose of the inspection is to discover and record defects. Defects recorded during the code inspection are identified by component, a brief

description of the defect, and type of error[1]. Currently, preparation for the inspection process is accomplished through the application of unstructured defect detection methods. These methods are composed of defect detection techniques, individual reviewer responsibilities, and a policy for coordinating responsibilities among the review team. Defect detection techniques range in prescriptiveness from intuitive, nonsystematic procedures (such as ad hoc or checklist techniques) to explicit and highly systematic procedures (such as correctness proofs). A reviewer's individual responsibility may be general, to identify as many defects as possible, or specific, to focus on a limited set of issues (such as ensuring appropriate use of hardware interfaces, identifying untestable requirements, or checking conformity to coding standards). Individual responsibilities may or may not be coordinated among the review team members. When they are not coordinated, all reviewers have identical responsibilities. In contrast, the reviewers in coordinated teams have distinct responsibilities. The most frequently used detection methods (ad hoc and checklist) rely on nonsystematic techniques. Reviewer responsibilities are general and identical. Multiple-session inspection approaches normally require reviewers to carry out specific and distinct responsibilities. One reason these approaches are rarely used may be that many practitioners consider it too risky

[1] Errors are classified as Errors, Suggestions, Question and Other

to remove the redundancy of general and identical responsibilities and to focus reviewers on narrow sets of issues that may or may not be present (Porter et al 1997). Clearly, the advantages and disadvantages of alternative defect detection methods need to be understood before new methods can be safely applied.

2.2 Steps in the Software Inspection Process

In figure 2.2, each software inspection is itself a lengthy five or six-step process that is carried out by a designated moderator, by the author of the work product being inspected, and by at least one other peer inspector. The process steps are:

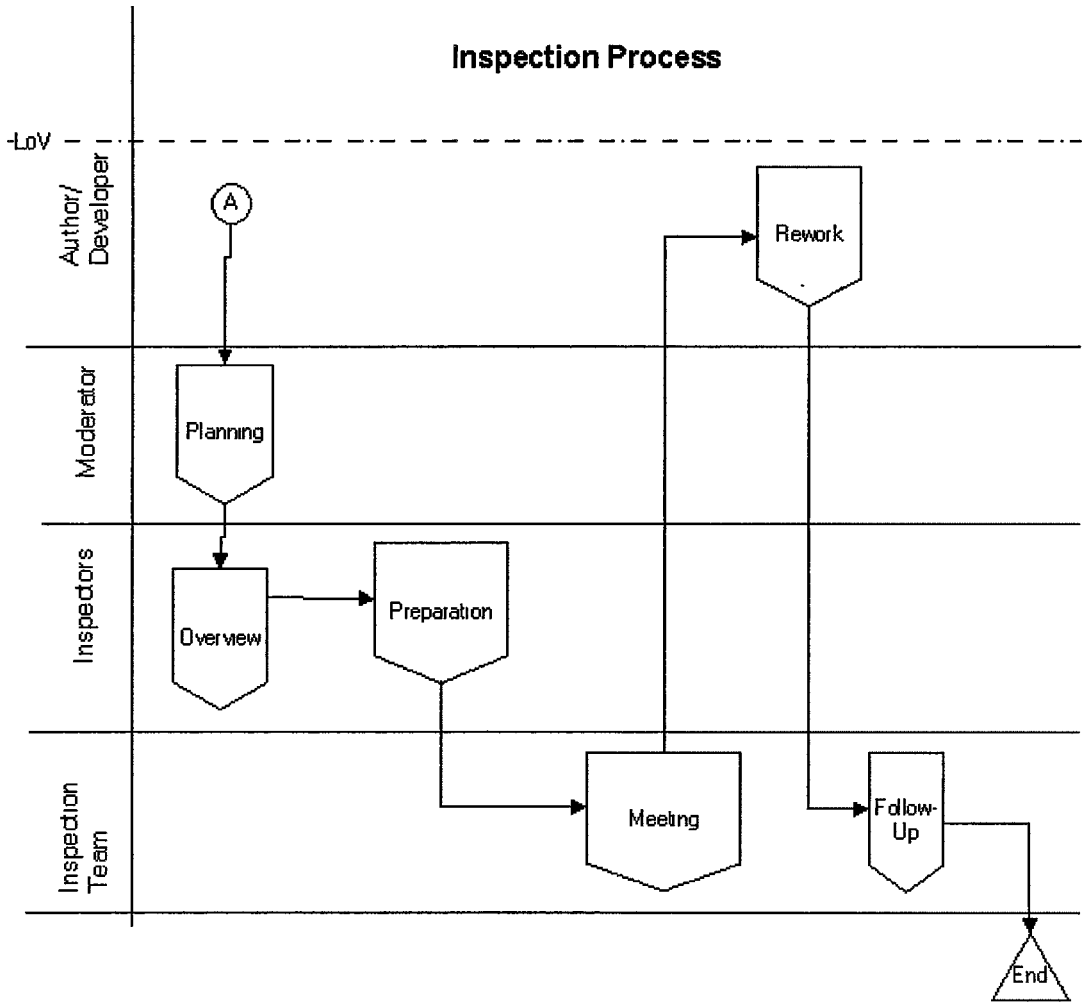

Figure 2.2: Inspection Process

Planning for a code inspection begins when a developer's work product is ready for inspection. The first step is to select a moderator - a peer developer responsible for carrying out the inspection. The moderator typically is selected by the author or a first line manager from among a pool of qualified developers or may be selected by an independent group with overall responsibility for the conduct of inspections. Since the moderator *has* overall responsibility for the

inspection, including the final decision on the work product's disposition at the end of the inspection, it is important that he or she be as objective as possible. One way to ensure this is to specify that the moderator not be a member of a group that has direct production responsibility for the inspected work product. The moderator's first responsibility is to meet with the author and to verify that the work product to be inspected meets the entry criteria for the inspection. When the work product meets these criteria, the next step is to decide whether to hold an overview or not, to select the other inspectors, and to schedule the overview and the meeting.

An Overview is a presentation to provide inspectors with any background information they may need in order to properly inspect the author's work product. Typically, the author gives the overview, and it often covers material pertinent to a number of inspections. Another use of an overview is to provide a tutorial session on a specialized design or implementation technique for the inspected work product. Due to the complexities in calculating the Fault Index, a brief tutorial is recommended. For example, the purpose of using the Fault Index is to focus on those components or modules that are highly complex and have the highest risk of failure (Gupta, Patnaik, Emam, and Goel, 1998).

Preparation for a software inspection is an individual activity. The author prepares by collecting all the material required for this inspection. Once

achieved, the next step would be to prepare a list and recommend which modules should be inspected. The other inspectors prepare by reviewing the code components identified by the output of the Fault Index calculation. The purpose of individual preparation is to develop an understanding of the work product and to note places where this understanding is incomplete, or where the work product appears to have defects. Obvious defects are noted during this step, but detailed analysis and classification of defects is deferred to the inspection meeting.

The next step in the inspection process is the formal meeting. The moderator conducts this meeting. There is an established agenda, which consists of:

- Introduction
- Establishing preparedness
- Reading material and recording defects
- Reviewing defect list
- Determining disposition
- Debriefing

During the introduction, the moderator introduces the inspectors and the examined material and states the purpose of the meeting. Preparedness is established by having each inspector report his or her preparation time. The

moderator sums these times for entry. If the moderator feels that preparation has been insufficient for an effective meeting, he or she may postpone the meeting.

Reading the material and recording defects are the major activities of an inspection meeting. At the seating, one of the inspectors takes the role of reader and paces the group through the material by paraphrasing each "line" of the material aloud for the group. As the reader proceeds through the material, the inspectors (including the reader) interrupt with questions and concerns. Each of these is either handled immediately or tabled. Whenever the group agrees, or the moderator rules, that a defect has been detected, the recorder for that inspection notes the location, description, class, and defect type. The moderator or a fourth inspector, other than the author, can assume the role of recorder.

After the reading and recording of defects, the moderator has the recorder review the defect list to ensure that all defects are recorded and correctly classified. After this, the inspectors determine the disposition of the material: "meets," "rework," or "reinspect." The disposition of "meets" is given when the work product as inspected meets the exit criteria (or needs only trivial corrections) required for the inspection. The disposition of "reinspect" is when the rework will change the work product in a substantial way.

The author performs inspection rework. This phase consists simply of correcting the defects noted in the inspection defect list. The follow-up step is the responsibility of the moderator. It consists of verifying the corrections made during rework. Clearly, this entire process can be time consuming and the net findings of faults in the code can be minimal.

2.2.1 Inspection Effectiveness and Attributes

A key attribute of software inspections is that the process itself collects data for evaluating its own effectiveness. When inspections were first being implemented in IBM by M.E. Fagan, a study was conducted which compared the effectiveness of the inspection process with an existing "walkthrough" approach. The bottom line of this study was that inspections resulted in a 23 percent productivity improvement and a 38 percent quality improvement (Buck, 1981).

Many additional studies conducted since then have documented the effectiveness of inspections (Peele, 1998). It is also important to note that inspections can be implemented on almost any software development project with minimal investment. This is another major benefit of the software inspection process and measured as a return on investment (ROI).

2.2.2 The Return on Investment for Inspections

ROI can measure the economic value of popular approaches to software process improvement. ROI is a simple arithmetic ratio of net benefits to costs, expressed as a percentage. That is, benefits minus costs are the numerator, and total costs are the denominator. ROI measures the magnitude of benefits to costs, benefits returned above costs, profits achieved after expenses, value of an investment, actual benefits, cost savings, and efficiencies obtained. Code inspections provide an ROI of approximately 2,500 percent. The benefit to cost ratio for inspections is 26:1. That means for every dollar spent on inspections yields 26 dollars in return. Therefore, code inspections have been found to provide the most benefit with the least amount of costs (Rico, 2002).

2.2.3 Scheduling for the Inspection

One of the key considerations to decide in planning for inspections is time. As previously mentioned, planning for the entire inspection process is challenging especially when time to market pressures exist. This situation is exacerbated when a large amount of code needs inspecting. Thoroughly examining all of the source code presents a significant challenge and is sometimes infeasible within given time constraints.

The starting point for code inspection implementation effort described in the next section is the incorporation and usage of a Fault Index standard for software code inspections. It provides an effective and efficient methodology for conducting software code inspections for a typical project.

2.2.4 Inspection Guideline Criteria

The implementation methodology first needs to establish and review the basic rules and instructions for the teams to follow. Some of the mandatory instructions for the code inspections are:

1. The inspection process should consist of the steps described previously; the inspection meeting should follow the agenda described previously.
2. The minimum number of participants for an inspection is three: a moderator/recorder, a reader, and the author.
3. An inspection meeting cannot be held unless the participants have individually studied the product prior to the meeting.
4. Each inspection must create a defect list, and defects classified.
5. Collect data on the effort expended and the types of defects found.

For each code inspection, it is mandatory that the project specify:

- Entry criteria

- Exit criteria
- Defect types

Entry criteria are the pro forma conditions that a work product must meet before considered ready for inspection. These generally include the development activity exit criteria that would apply if an inspection were not specified for that activity, but they also include requirements for an effective inspection. For example, the entry criteria for a code inspection would require a clean compile and Fault Index calculation, but would also specify that the inspected material have visible line numbers and pertinent requirements, design, and change information included as part of the inspection package. Exit criteria are the completion conditions for an inspection. Typically, these are the correction of all detected defects and the documentation of any uncorrected defects in the project trouble-tracking system.

As described in the previous section, defect counts are used as process control data. Thus, the precise classification of defects is an essential part of the specification of each inspection. For software inspections, a defect is defined to be non-compliance with the product specification or a documentation standard. Again, individual preparation is a mandatory requirement of the code inspection process. At the inspection meeting, the emphasis is on pooling the individual understanding gained during preparation to maximize defect detection.

Strategies for detecting defects depend on the inspection, the kind of material, and the defect categories. In Fagan's implementation, the inspectors are provided with checklists keyed to the defect types (Fagan, 1976). Since a software inspection is a cooperative process that relies on group synergy, the selection of participants is an important issue. In a code inspection, teams of three or four developers concerned only with the detail design, coding, and unit test are effective (Graden & Horsley, 1986).

CHAPTER III

3.1 Code Inspection and the Fault Index Process

Source code is an elusive entity. It contains complexities both obvious and hidden. It contains attributes that either enhance or detract from its understandability, readability, complexity, maintainability, and reliability. It can be entirely free of defects or contain hazardous, devastating ones. To improve source code quality is to identify and remove detrimental aspects of the source code and enhance the beneficial aspects. Software measurement is one method for identifying and differentiating detrimental and beneficial aspects of source code. Indeed, there are certain complexity metrics that have shown to be distinctly associated with defects (Munson and Khoshgoftaar, 1990a).

The methodology for identifying source code attributes and their relationship to defects was demonstrated in several antecedent studies. The main idea is that we can identify those software attributes that are associated with defects.

The theory and contention of this paper is that the usage of the Fault Index to rank and select high-risk modules used for a code inspection will ultimately improve inspection effectiveness and thus improve code quality, reduce rework, and improve field reliability.

Usually, the author and several other reviewers examine a piece of source code for the purpose of identifying defects. To identify defects the reviewers must understand the code. Code inspections are undoubtedly useful, but humans are fallible and so the reviews and inspections are fundamentally flawed; defects pass code inspections undetected (Votta, 1993). This downstream effect has major consequences for a development organization. As defects go undetected through the product lifecycle, it becomes more costly to find and fix them (Porter et al 1997). Figure 3.1 shows the cost of a defect as it moves through the project lifecycle. The figure contains data collected from over 70 projects beginning in 2004.

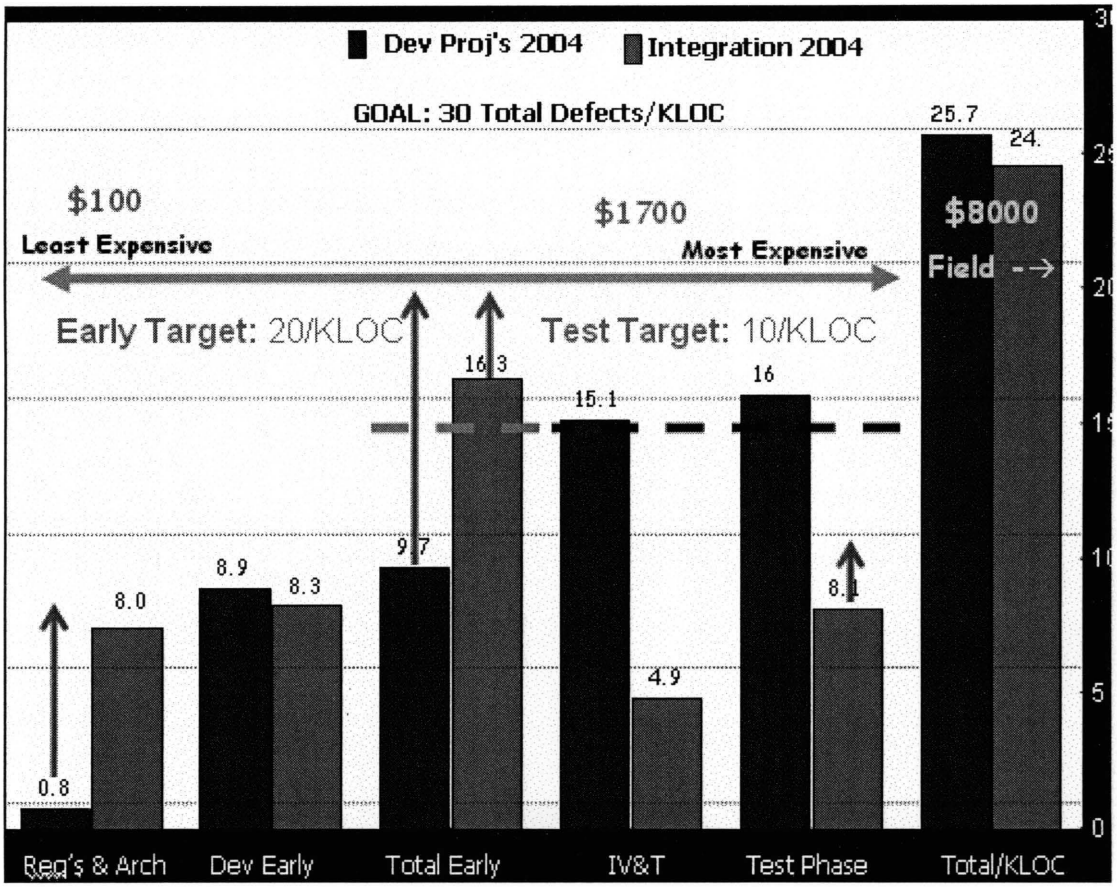

Figure 3.1: Lifecycle Defect Costs

The figure shows how much it costs to find and fix a defect at each process step. The y-axis represents defects/kloc and the x-axis represents each process step. Finding and fixing a defect in the requirements phase costs $100. When a defect escapes requirements and is found and fixed in the test phase, it now costs $1700. When the defect goes undetected and not fixed early in the lifecycle, the cost significantly increases from requirements to the field. Therefore, it is less costly to find defects earlier than later in the field. In addition, customer found defects negatively impacts customer satisfaction. Therefore, enhancing source

code understanding would improve the effectiveness of reviews and inspections thus substantially reduce lifecycle costs and preventing customer dissatisfaction. Subsequently, fewer defects would remain undetected and the overall quality of the source code would improve.

Through a considerable amount of research, statements on inspection effectiveness have been formalized and recognized. It is known that "...more time should be spent on a complex part of the code" (Khoshgoftaar, Munson, Lanning, 1994) and "identify spots where code is most complex, best inspections are only 60 - 80 percent effective." Effectiveness of source code inspections are improved by the identification and rigorous review of the most complex sections of the source code.

3.1.1 Introduction to the Fault Index

Software complexity is like the weather. Everyone "knows" that today's weather is better or worse than yesterday's. However, if an observer were pressed to quantify the weather, the questioner would receive a list of atmospheric observations such as temperature, wind speed, cloud cover, and precipitation. It is anyone's guess as to how best to build a single index of weather from the weather metrics. Software complexity is defined in IEEE Standard 729-1983 as:

The degree of complication of a system or system component, determined by such factors as the number and intricacy of interfaces, the number and intricacy of conditional branches, the degree of nesting, the types of data structures, and other system characteristics (IEEE, 1983).

By counting these composite metric attributes (and many others), researchers are trying to build a single index of software complexity (Stark and Lacovara, 1985). With a single index available, Java subroutines[2] are placed in complexity order *relative* to one another, giving rise to the name "relative complexity." Recently, relative complexity has gained an alternative name called the Fault Index. The Fault Index can be used prior to and during the coding, testing, and maintenance phases of the software life cycle to locate code "hot spots." The challenge for researchers in software complexity is to define a computational method that agrees with a programmer's intuition of complexity and highly correlates to attributes such as code defects.

With many program complexity metrics available, it is difficult to rank programs by complexity: the different metrics can give different indications. There are two parts to this problem. First, because different metrics can measure the same program attribute, we need a method of evaluating a given program attribute based on the values of all metrics that measure this at-

---

[2] Hereafter, we refer generically to "functions", "modules", and "packages" as the operative level of granularity of the analysis.

tribute. Second, because different metrics can measure distinct program attributes, we need a method of evaluating the overall program complexity based on the preferred values of all program metric attributes. Principal components analysis (PCA) is a statistical method that represents each independent variable metric in proportion to the amount of unique variation contributed by that complexity metric. The next section describes the rationale for using the statistical technique called principal component analysis. Principal component analysis identifies a small number of unobservable factors, which give rise to function complexity. Next, the method for building a single Fault Index based on the principal component analysis is examined. Finally, the results of the combination of the Fault Index and inspection process yield a remarkable improvement to both the inspection process and overall software development lifecycle.

3.2 The Fault Index Metric

Researchers have developed a realistic measure of a single index metric called the Fault Index that combines the measuring capabilities of any number of software metrics (Munson and Khoshgoftaar, 1990a). The method applies principal components analysis, a widely used statistical method that is useful in

identifying uncorrelated sources of variation in multivariate data (Johnson and Wichern, 1992). 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 X 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 a single fault index. The selected principal components explain the structure of software complexity data, and refer to the components as complexity domains (Munson and Khoshgoftaar, 1992).

In order to simplify the structure of software complexity even further than the orthogonal domains produced by the principal components analysis, it would be useful if each of the program modules in a software system could be characterized by a single value representing some cumulative measure of complexity. Furthermore, the measure would serve as a parameter in a function to rank the complexity of software modules. The objective in the selection of such a linear function, g, is that it be related in some manner to software defects either directly or inversely such that $g(x) = ax + b$ where x is the cumulative measure of complexity. The more closely related x is to software defects, the more valuable the function g will be in the anticipation of software defects. Previous research has established that the Fault Index metric has properties that will be useful in this regard. The Fault Index metric is a weighted sum of a set of

uncorrelated attribute domain metrics (Munson and Khoshgoftaar, 1990a; Munson and Khoshgoftaar, 1990b). This Fault Index metric represents each raw metric in proportion to the amount of unique variation contributed by that metric.

To avoid the use of negative complexity values, scale the module Fault Indexes to a new mean and standard deviation. For this study, the Fault Index values scale such that they have a mean of 50 and a standard deviation of 10. Thus, the average Fault Index of the scaled values will be 50. A module with a Fault Index of 60 will be seen to be one standard deviation above the new system average of 50.

The principal value of the Fault Index metrics is in its relationship to software defects. When the Fault Index metric is calculated, a ranking from the highest to lowest value will be performed. If the Fault Index ranking of a program module is large, then the number of defects ultimately found in that module will potentially be large. In this capacity, the Fault Index measure will satisfy the need as a fault substitute. In addition to software defects, the Fault Index metric also has many applications in software project management and software reliability engineering (Khoshgoftaar and Munson, 1992; Munson and Khoshgoftaar, 1990a, 1991b).

3.2.1 Principal Component Analysis

Principal component analysis (PCA) involves a mathematical procedure that transforms a number of (possibly) correlated variables into a (smaller) number of uncorrelated factors called *principal components*. The first principal component accounts for as much of the variability in the data as possible, and each succeeding component accounts for as much of the remaining variability as possible.

Traditionally, principal component analysis is performed on a square symmetric matrix of pure sums of squares and cross products (SSCP), Covariance (scaled sums of squares and cross products), or Correlation (sums of squares and cross products from standardized data). The analysis results for objects of type SSCP and Covariance do not differ, since these objects only differ in a global scaling factor. A correlation object has to be used if the variances of individual variants differ much, or if the units of measurement of the individual variants differ.

Among the large set of static code metrics, each member or metric is observed as a measure of a *distinct* attribute of a program. Some subsets of this set actually measure a single attribute. For example, two common code metrics, executable lines of code and physical lines of code both measure the number of lines of code

in a program but the counts are slightly different. In some languages, this correspondence is one to one. These two static metrics share a *common* measure of program size. This results in a high correlation between the two metrics. This strong correlation identifies that the two static metrics measure the same program attribute and in this study, is undesirable.

For this study, different static complexity metrics can measure the same program attribute. Therefore, we need to identify and remove those static complexity metrics that are strongly correlated. In addition, because different metrics can measure distinct program attributes, a statistical method of evaluating the overall program complexity is needed which is based upon the values of the various program attributes.

This problem entails more than the consideration of just two static metrics. Today, well over a 100 static software complexity measures are defined and there is compelling evidence that at least five contribute to program complexity (Munson and Khoshgoftaar, 1989). All of these measures are some type of observation of attributes of source code text. Many serve different purposes, often someone's notion of the "real" or "essential" complexity of the code. Like some concepts in software engineering, they often have no theoretical basis, but represent an ad hoc view of complexity. There are commercially available tools that parse source code and output a laundry list of static complexity measures.

Many of the static measures are combinations of fundamental observations. We call these derived or independent variable metrics (e.g., Halstead volume, log McCabe Complexity). Once the PCA is complete, the Fault Index will be produced.

3.2.2 The Transformation Matrix

Part of the PCA is producing the standardized transformation matrix. In this particular study, we started with nine original static metrics. The goal of the PCA is to reduce the number of independent variables into a smaller number of domains. The transformation matrix is used to collapse the original matrix of metric values onto a smaller matrix of domain scores. In this case, we collapsed the original nine static metrics into two domains. This reduces the dimensionality of the measurement problem. The transformation matrix then, is an m X d matrix that, when multiplied by the original matrix (n X m), creates an n X d matrix of domain scores. Finally, at the end, we multiply the n X d domain score matrix by the d X 1 vector of eigenvalues for the identified domains. This yields the Fault Index; a single measure of complexity for each of the modules in the system.

3.2.3 A Fault Index Computational Example

The computation of the Fault Index involves a number of steps in the transformation of the raw metric data. The PCA-RCM tool performs all of those calculations. The tool computes standardized scores, or z scores, for each of the raw metrics for each of the modules. This transformation is accomplished by first subtracting the mean value of each metric from each raw metric. This will insure that all metrics have the same mean: 0. Next, it divides these transformed metrics by the standard deviation for that metric, yielding the final z score for the corresponding raw metric. When all of the 4 raw metrics for all modules have been similarly transformed, they will all have the same mean (0) and the same standard deviation (1). The next step in the computational process is the transformation of the standardized metric scores to factor scores or domain scores. This will reduce the total number of measures for each program module from nine scores to one or two scores representing the orthogonal metric domains. The products of the factor scores are summed for each domain to create a domain score, *DS*, for each orthogonal component. The domain score is distributed with a mean of zero and a standard deviation of one.

The final computation is the Fault Index. This measure will consolidate the domain metrics into a single measure, which will also be our fault substitute measure. To compute the module Fault Index, each of the domain scores is

weighted by the eigenvalue of the corresponding domain. These eigenvalues represent the relative proportion of variance that each of the domains explains. The resulting sum of the products of the domain scores and the eigenvalues yields the Fault Index.

This Fault Index value has a mean of zero and a variance equal to the sum of the squared eigenvalues as follows:

$\sigma_p^2 = \sum_i \sigma_i^2$. The Fault Index is then scaled to a more tractable distribution by adjusting the standard deviation of the Fault Index to one and dividing it by the standard deviation, σ_p, for the domains. Finally, the Fault Index is adjusted to a mean of 50 and a standard deviation of 10 in this study. The whole process may be summarized as follows: $p_i = (p_{i'} / \sigma_p)/10 + 50$

3.3 Research Methodology

In the case of this study, a single proxy measure, the Fault Index, is constructed to serve as a code selection measure. This single measure will permit you to rank the program modules from high to low. Once the modules are ranked, those program modules that possess large values will most likely possess a high number of defects. In this case, the Fault Index will serve as a predictor of

defects. If the code executes a significant proportion of time in those modules, the potential exposure to software defects will be great and the software will likely fail. Therefore, utilizing a Fault Index in the inspection process permits the inspection teams to select those code modules that are at high-risk for failure. As a result, the ability to improve the source code quality will significantly increase. In addition, the usage of the Fault Index is considered as an inspection process enhancement. It is a proactive approach to improve source code quality, which is better than a reactive approach of finding and fixing defects in test or in the field.

Because software can be measured in a multitude of ways, the scope of possible software metrics is quite large. There are a large number of static complexity metrics to choose from in the construction of a proxy measure for software faults (Zuse, 1990). Many of these metrics have a high degree of interrelationship among all of the metrics and are more meaningful than others are for a particular project or purpose. Therefore, a chosen subset of static metrics that will be used must:

- Possess a high correlation with defect measures
- Posses a low correlation with other object-oriented metrics

This is important for the subsequent calculation of the Fault Index.

The primary purpose and intent is to implement a standard to enhance the effectiveness of software code inspections by improving the code selection process. Software inspections already had been chosen as a prime candidate for a standard precisely because, as previously mentioned, their effectiveness had been measured and verified. Three ongoing projects were selected to pilot the study of using a Fault Index during code reviews. No sophisticated selection criteria were enforced, but the teams for the pilot evidenced two important attributes. First, the development teams were currently in the middle of a development cycle, however, because the source code was being reviewed after the inception of the pilot (the same languages, authors and coding styles were used), code inspection verification was difficult. The data from this research was collected at various steps in the process. Second, the teams were quite small, which made the initial pilots manageable. Next, three systems selected for the pilot study will briefly be described.

3.3.1 The Software Systems

One of the three teams was developing an On Demand system (System 1) as part of the IBM corporate strategic direction. Five programmers were involved throughout the entire project, and another five programmers (including two

contract programmers) were involved in at least some part. Development included high-level languages, and integrated code in excess of 900 thousand lines of code (KLOC) for the entire project. This highly complex system monitors and reports on the performance of various client software systems. The second system is an Automated Deployment Tool (System 2) that automates the deployment of software in various heterogeneous environments. Three programmers were involved and integrated 50KLOC. The final system is Mixed Address Database (System3) that enables users to create, update, and delete Intranet and Internet account, sub-account and device records. Only two programmers were involved and only produced 10KLOC.

3.3.2 The Selection of Tools

The next task was selecting various tools for use in collecting all of the static software metrics. Three tools were chosen: Together Architecture by Borland, Principal Component Analysis-Relative Complexity Measure (PCA-RCM), Source Code Analysis, and Measurement Program (SCAMP). The Together tool is capable of generating over 30 object-oriented and static complexity metrics for the Java and C++ programming language. Initially, a wide variety of software metrics was collected for each of the systems (Mayer and Hall, 1999). Table 3.1

partially lists the output of some object-oriented and complexity metrics from the Together tool and their definitions are in Appendix B.

Metrics	CBO	CC	DAC	DOIH	FO	HDiff	MNOL	MSOO	NOprnd	NOprtr	NORM	NUOprnd
Package1	31	26	2	1	23	119	7	15	3258	3227	100	317
Package2	3	59	1	1	3	60	2	3	616	518	7	97
Package3	14	65	0	1	8	390	2	10	5811	5855	72	255
Package4	13	33	1	2	11	52	3	4	884	937	34	172
Package5	3	1	0	4	0	8	0	1	36	36	4	8
Package6	11	48	3	1	9	43	2	3	477	423	17	79
Package7	13	89	3	2	9	74	2	22	2037	1713	34	293
Package8	3	9	1	3	2	17	2	3	225	165	11	84
Package9	34	63	3	3	30	126	13	27	2006	2090	77	317
Package10	18	194	1	3	12	94	24	65	3808	4028	52	530
Package11	4	7	0	1	3	13	2	4	166	185	19	65
Package12	12	8	2	1	6	24	2	6	381	400	33	111

Table 3.1: Static object-oriented metrics

First, the source code for each system was loaded into the Together tool. Second, the tool performed all of the static calculations for each of the systems. Third, after the calculations were performed the tool produced a graph, which plotted the static metrics on an axis. The tool provided some standard recommended industry boundary limits for each of the static measures. The tool also permits the user to adjust those limits if needed. Figure 3.2, called a Kiviat graph, displayed the static measures and their boundary limits.

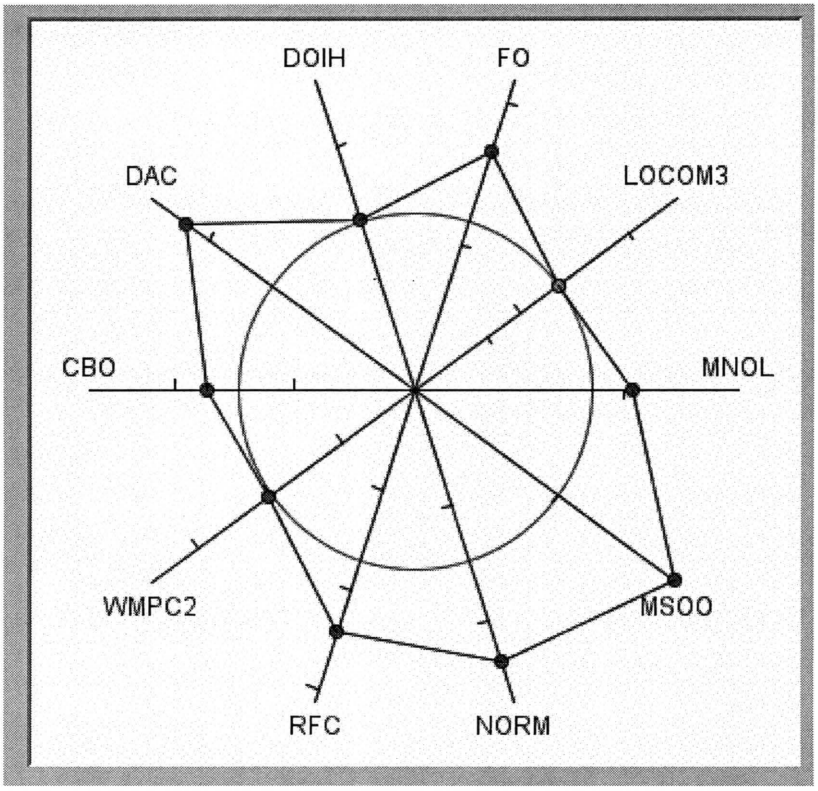

Figure 3.2: Kiviat Graph of Object-oriented Metrics

Each axis (or ray) of the Kiviat graph represents one metric, as labeled in the graph. The measurement scale of each axis is the range of the metric: the minimum value is located in the center, the maximum value at the outer end. The axes are linearly scaled. The line between each metric connects the measurement values of the selected element for each metric on the axes. If the element has many relatively large values, the area enclosed by the thick line will be large. Therefore, the size of the enclosed area serves as an indicator of the criticality of the element.

The graph identified which metric exceeded their particular boundary and is considered a prime static metrics candidate. Once that analysis was completed, correlations were performed on those metrics. Based on the results of the correlations, the final metrics set was selected for use in the study.

3.3.3 Correlation of Metrics

One of the key process steps for this study is to identify correlations between the metrics and software defects. The correlation *between* the static metrics must be low. The importance of this low correlation is that we want metrics that are *unique* and will provide meaningful input values for the Fault Index calculation. If the static metrics strongly correlate with one another, this may imply that they are in some way identical. Therefore, the static metrics for this study was based on correlations below 40-50 percent. Conversely, the correlation between static metrics *and* software defects must be strong. Again, the importance of this is that we want to provide meaningful input into the Fault Index calculation in order for it to be a good predictor of software faults. Therefore, the correlations must possess a greater than 70 percent value.

The values of the static metrics produced by the Kiviat graph were analyzed along with their correlations and only a subset of these metrics proved useful for this research.

After performing the static analysis on the source code data from Table 3.1, the only significant static metrics used were based on their correlation, which is denoted by the R-squared value (an example is given in Figure 3.3).

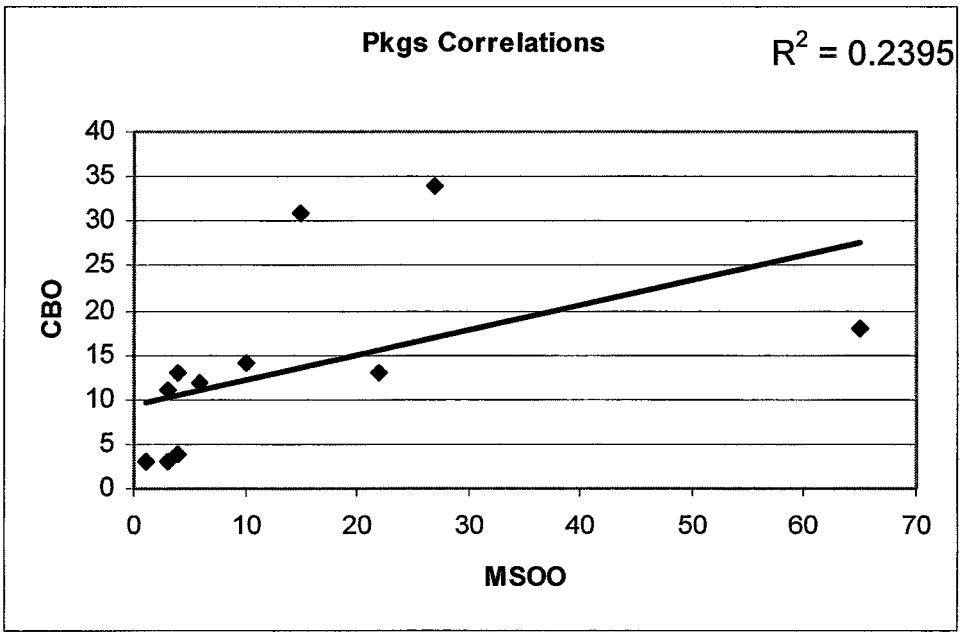

Figure 3.3: Independent variable correlations

As shown in the figure, the two metrics, Coupling Between Objects (CBO) and Maximum Size of Operation (MSOO) are not strongly correlated as denoted by

the 23 percent R-squared value. The weak correlation between the two metrics provides meaningful input into the Fault Index calculation and satisfies the criteria for this study. Therefore, the two metrics are a prime candidate for use as input into the Fault Index calculation. If the two metrics were strongly correlated, it would have meant that they are essentially providing an identical relation into the Fault Index calculation therefore providing no additional value.

In figure 3.4, the same analysis was performed to determine the correlation of each defect count to static metrics.

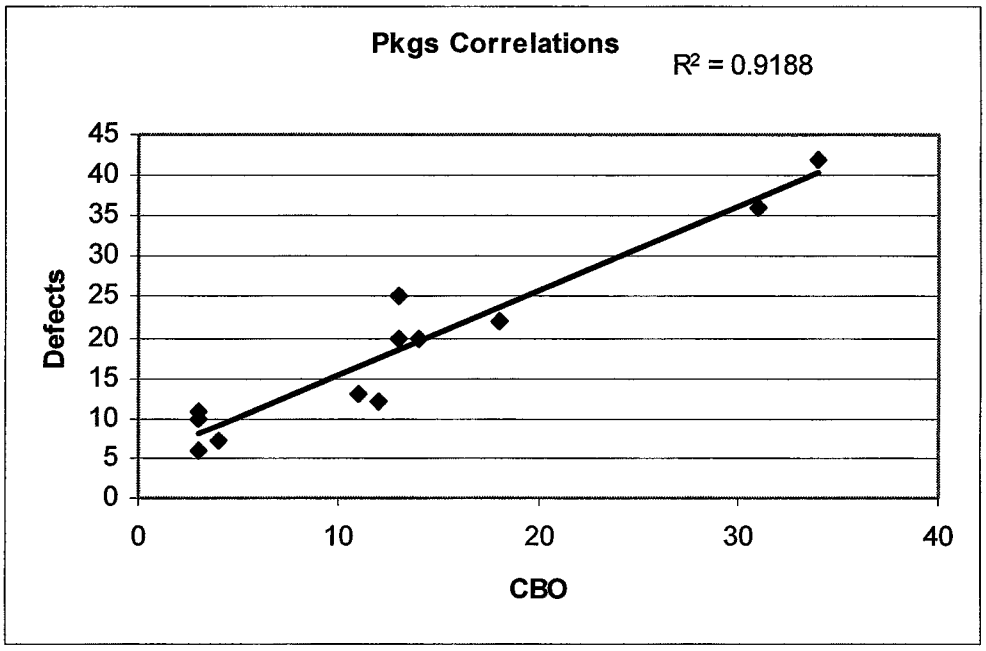

Figure 3.4: Defect to static metrics correlation

As stated earlier, a strong correlation is present, denoted by a 91 percent R-squared value, and shows a positive relation exists between the static metric CBO and defects. This indicates that the metric possesses a strong relationship in determining software defects in the program modules.

Table 3.2 is the final list of the metrics that were selected from the static metric and defect correlations along with their definitions.

RFC	Response for Class; the number of methods, internal and external, available to a class
CBO	Coupling between Objects; counts the number of other classes to which a class is coupled
V(g)	Cyclomatic Complexity (McCabe, 1976)
FO	Fan Out; number of reference types
VOD	Violations of Demeters Law (Chidamber and Kemerer, 1994)
DOIH	Depth of Inheritance; How far down the inheritance hierarchy for a class
LOCOM	Lack of Cohesion Metric;
MNOL	Maximum Number of Levels; depth of (if, for, while) statements
MSOO	Maximum Size of Operation; The number of operations (if, for, while) for a class
WMPC	Weighted Methods Per Class; sum of the complexity for a class
NORM	Number of Remote Methods
DAC	Data Abstraction Coupling

Table 3.2: Selected metrics and definitions

CHAPTER IV

4.1 Output Results from the Principal Component Analysis

In most linear modeling applications concerned with the mapping of software metrics onto software defects, the independent variables, or metrics, from Table 3.1 each are assumed to represent some distinct aspect of variability that is not obviously present in other measures. In software development, the independent variables (in this case the various complexity and object-oriented metrics) strongly correlate with defects and weakly correlate with each other. Such linear models may be subject to changes due to additions or deletions of variables or even discrete changes in metric values. To avoid these problems, principal components analysis (PCA) was used to map the metrics onto orthogonal attribute domains. Each principal component extracted by this procedure may be observed to represent an underlying common attribute domain (Jackson, 1991). Some object-oriented metrics such as *Fan Out* (FO), *Weighted Methods Per Class* (WMPC), and *Response for Class* (RFC) are not strongly correlated with each other. Therefore, these metrics generally prove to associate well with a principal component related to the object-oriented attribute of a program.

We will now examine the outputs of the principal components of the metrics (independent variables) for the three software systems. The first step in the investigative process is to determine how many distinct sources of variation are truly measured by the raw static metrics selected for use in all three systems. Table 4.1 contains the results of the principal components analysis performed on the software packages from system 1.

System 1	**Domain1**	**Domain2**
FO	0.61	0.58
MNOL	0.80	-0.30
MSOO	0.91	-0.04
WMPC	-0.09	0.88
Eigenvalues	1.86	1.20

Table 4.1: Orthogonal attribute domains for system 1

System 1 is comprised of four metrics that are based on the correlation selection and was reduced to two orthogonal attribute domains. The table contains the values of the varimax rotation of the factor pattern. To reiterate, one of the primary purposes of principal components analysis is to reduce the dimensionality of the attribute problem. The data shows that the metrics *MSOO,*

MNOL, and *FO* are distinctly associated with the first principal component labeled Domain l. In addition, each domain of all three systems, has an eigenvalue greater than one. This is important because it represents the relative contribution of its associated domain to the total variance explained by all of the domains.

Upon application of principal components analysis, only two distinct attribute domains were produced for systems 1 and 3 and only one for system 2. Tables 4.2 and 4.3 show the orthogonal attribute domains for systems 2 and 3 for their respective metrics set.

The domain metrics give considerable information about the programs. Consider the domain metrics for system 1. The large positive value of the Maximum Size of Operation (MSOO)[3] indicates that the influence of this program's MSOO complexity is strongly positive compared with this influence in other programs. The slight negative value of the weighted methods per class domain metric indicates that the influence of this program's complexity is average compared with this influence in other programs.

System 2 and 3 both have large positive domain values for VOD, RFC, and FO metrics in Domain 1. However, system 3 has lower and negative domain

[3] The number of operations (if, for, while) for a class domain metric.

results in Domain 2. The increase in the difference between these values reflects the wider gap between the program's complexities. The more negative value of the domain metric reflects the decrease in the program's complexity.

System 2	Domain1
FO	0.87
RFC	0.90
VOD	0.92
Eigenvalue	2.42

Table 4.2: Orthogonal attribute domain for system 2

System 3	Domain1	Domain2
FO	0.78	-0.27
MSOO	0.85	-0.31
NORM	0.94	-0.04
RFC	0.87	0.31
VOD	0.84	-0.38
V(g)	0.58	0.73
DOIH	0.43	0.31
Eigenvalues	4.22	1.04

Table 4.3: Orthogonal attribute domain for system 3

By transforming the static metric attributes for each of the system modules into orthogonal domain metrics, consequently standardized all of the measurements

in order to achieve relevant comparisons among the measurements. To this end, each of the system's results were baselined on their correlated static code measurements. To achieve this baseline measurement, means and standard deviations were computed for their respective set of selected static metrics. Further, a transformation matrix was computed for the mapping between the static metrics to the one or two orthogonal domain metrics for this set of program modules. The transformation matrices were obtained from the principal components analysis and are shown in Tables 4.4, 4.5, and 4.6 respectively.

System 1	Domain1	Domain2
FO	0.33	0.48
MNOL	0.43	-0.25
MSOO	0.49	-0.03
WMPC	-0.05	0.73

Table 4.4: Transformation matrix for system 1

The standardized metrics were then transformed to their domain metric equivalents. All subsequent domain metrics were obtained using the original baselined transformation matrix. The domain metrics are normalized such that they have a mean of 0 and a standard deviation of 1. In this manner, all of the

domain metrics were established to be comparable. For example, a domain metric of + .36 indicates that the associated module is .36 standard deviations above the average module value of 0.

System 2	Domain1
FO	0.36
RFC	0.37
VOD	0.38

Table 4.5: Transformation matrix for system 2

System 3	Domain1	Domain2
FO	0.19	-0.26
MSOO	0.20	-0.30
NORM	0.22	-0.04
RFC	0.21	0.29
VOD	0.20	-0.36
V(g)	0.14	0.70
DOIH	0.10	0.30

Table 4.6: Transformation matrix for system 3

4.2 Output from the Fault Index calculation of the Three Software Systems

Once the principal component analysis was complete, the final step was the calculation of The Fault Index for the three systems. The values of the Fault Index (FI) for the three systems are shown in Tables 4.7, 4.8, and 4.9 respectively.

System 1 Packages	DOMAIN1	DOMAIN2	FI	Defects
Package13	2.48	0.03	71	45
Package1	0.87	1.59	66	35
Package3	0.27	1.16	59	30
Package4	-0.34	1.42	55	25
Package14	1.17	-1.35	52	12
Package11	0.51	-0.53	51	12
Package2	0.17	-0.44	49	21
Package9	-0.45	0.03	46	23
Package12	-0.53	0.09	46	6
Package7	-1.12	0.75	45	4
Package6	-1.09	0.69	45	5
Package5	-0.18	-1.04	43	10
Package8	-0.54	-1.18	39	9
Package10	-1.22	-1.21	33	5

Table 4.7: Fault Index for system 1

In its capacity as a fault substitute, the program packages shown in Table 4.7 are ordered from the most complex (highest FI) to the least complex (smallest FI). Previous validation studies support the conclusion that the packages that possess a high Fault Index are the modules that have the greatest potential for defects (Khoshgoftaar and Munson, 1990; Munson and Khoshgoftaar, 1992b).

However, not all of the large Fault Index packages were identified as being the ones with the most defects reported. Possible explanations for this may include that the package was not tested thoroughly to expose those parts of the module where the faults existed. Some of the paths through the code were executed and others were not. Finally, the possibility of random chance is considered as well. However, in all three cases, the highest Fault Index did produce the most defects.

System 2 Packages	DOMAIN1	FI	Defects
Package9	1.71	67	42
Package1	1.53	65	36
Package10	0.93	59	22
Package7	0.86	59	20
Package3	0.19	52	20
Package4	-0.23	48	25
Package6	-0.50	45	13
Package12	-0.55	44	12
Package2	-0.74	43	11
Package8	-0.88	41	10
Package11	-1.11	39	7
Package5	-1.22	38	6

Table 4.8: Fault Index for system 2

It is interesting to consider programs from the high and low extremes and from the middle of the Fault Index scale. System 3 has the highest observed Fault Index of all three systems and this system had the highest eigenvalue.

System 2 has the lowest Fault Index. It is important to note the values of the Fault Index for all three systems and their weighted metrics failed to rank the elements in the order suggested by the metrics contributing to the weighting.

The Fault Index metric ranking of these systems is partially consistent with the order implied by the raw static metrics. Still, this comparison of the Fault Index metric and the traditional object-oriented metrics does not validate either in the sense offered by (Schneidewind, 1992).

System 3 Packages	DOMAIN1	DOMAIN2	FI	Defects
Package5	2.47	-0.77	72	25
Package6	1.09	0.37	61	12
Package7	0.49	1.51	58	14
Package4	-0.01	2.37	56	5
Package1	0.40	-1.17	51	15
Package12	-0.16	-0.39	48	10
Package9	-0.36	-0.10	46	13
Package11	-0.28	-0.47	46	11
Package3	-0.31	-0.66	45	2
Package8	-1.05	-0.13	40	5
Package10	-1.09	-0.13	39	2
Package2	-1.20	-0.44	37	4

Table 4.9: Fault Index for system 3

It can be observed from the values in the tables that there is a great deal of variability in the complexity of these program packages. To develop an

understanding of this failure potential, a means of characterizing the functional behavior of a system in terms of how it distributes its activity across a set of modules needs developing. This requires knowing what the program does, its functionalities, and how these different functionalities allocate time to the modules that comprise the system.

4.3 Results

The results from this study are depicted in the following four figures of the three selected systems. Figures 4.1, 4.2, and 4.3 clearly demonstrate that there is a strong correlation of defects associated with utilizing the Fault Index metric. Figure 4.4, is showing the combined results of all three of the programs into one graph to depict how various types of projects and programming styles do not influence the correlation of the Fault Index metric.

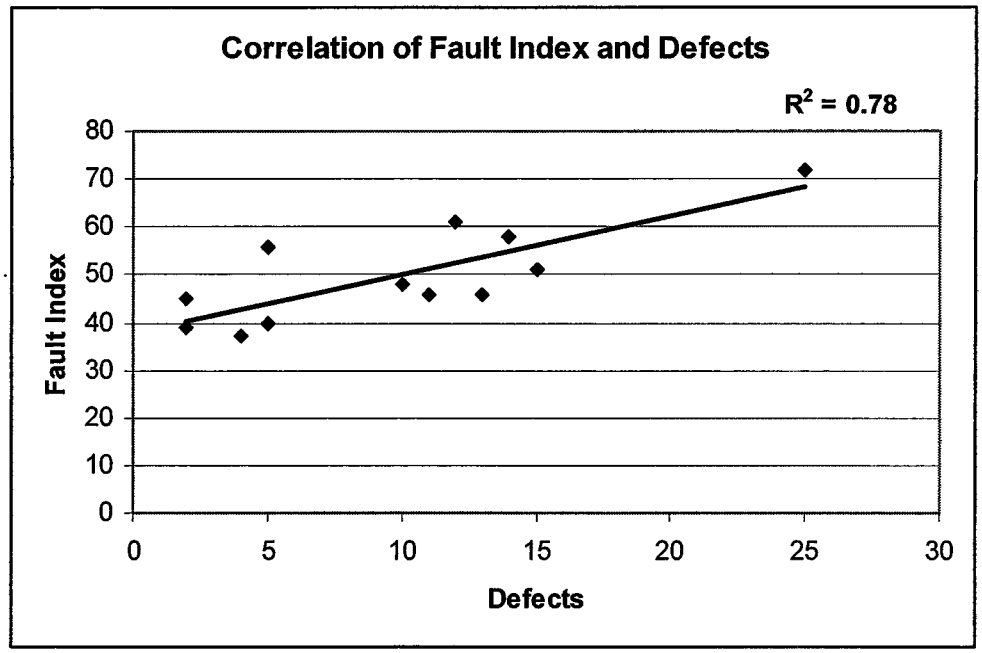

Figure 4.1: Fault Index distribution for system1

This study identified a strong correlation between the Fault Index metric and the software faults. System 1 had a 78 percent positive correlation of its Fault Index and the defects discovered. It was determined that a correlation greater than 60 percent was sufficient to claim success. It is also concluded that using the Fault Index as a surrogate proxy measure can fairly predict and aide in the selection of code candidates for the inspection process.

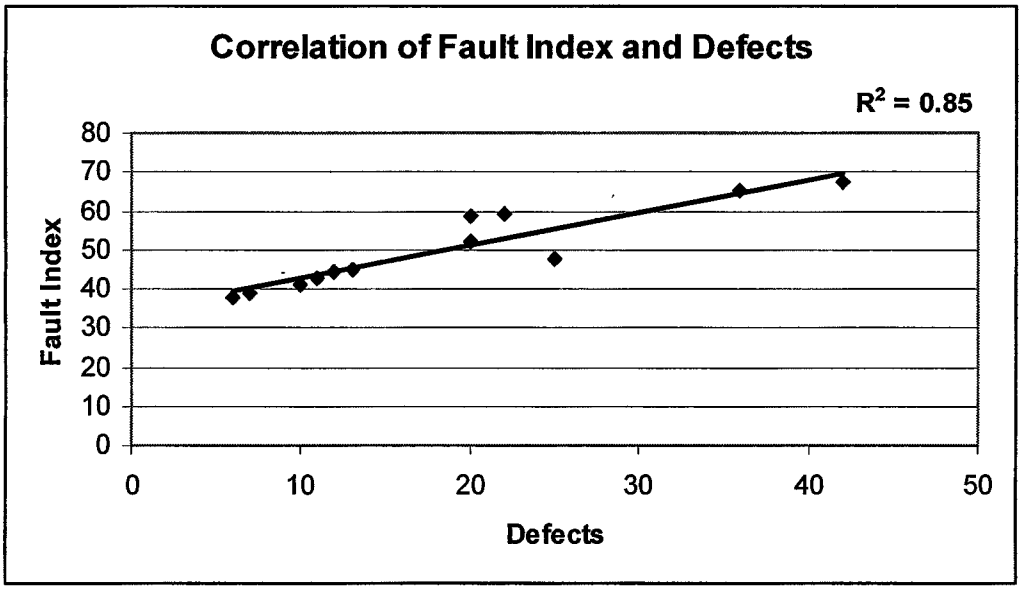

Figure 4.2: Fault Index distribution for system2

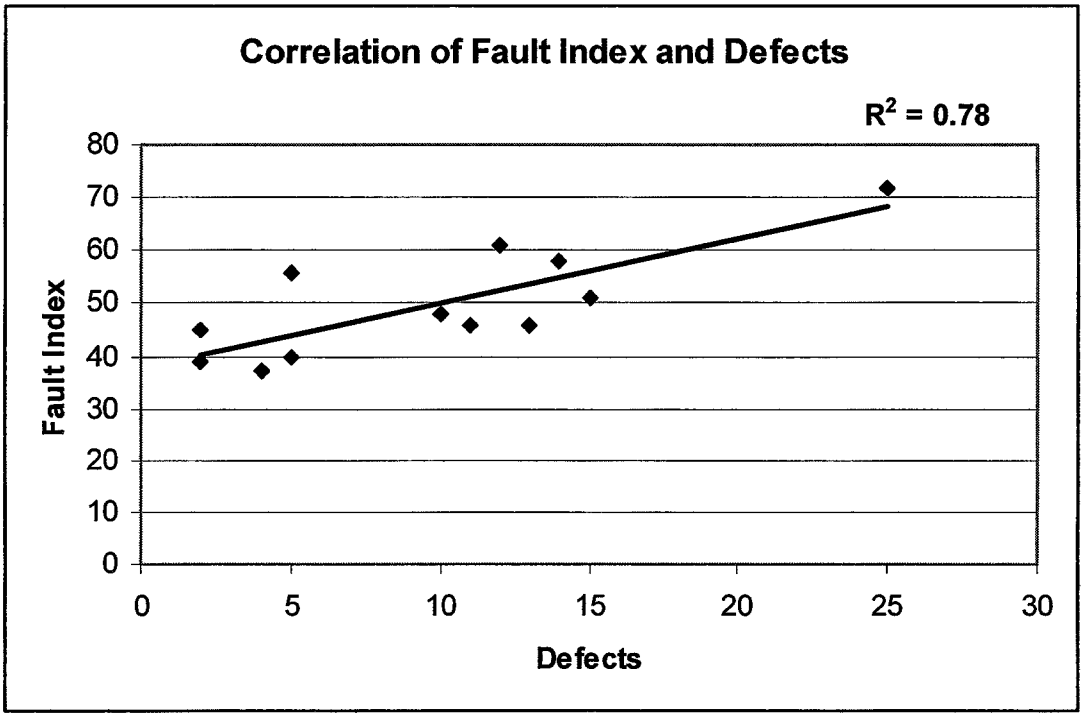

Figure 4.3: Fault Index distribution for system3

It is clear that the complexity of programs is multidimensional. There are many distinct and uncorrelated complexity domains. Different metrics can measure different program attributes. At the same time, different metrics can measure the same program attribute. With many metrics available for a group of programs, it becomes difficult to rank the programs by complexity: the different metrics can give different indications. In Figure 4.4, all of the systems data was combined to provide an overall Fault Index to defect correlation. This is meant to substantiate and prove the validity of the hypothesis. The correlation of the Fault Index and defects of all three systems combined was determined to be 65 percent.

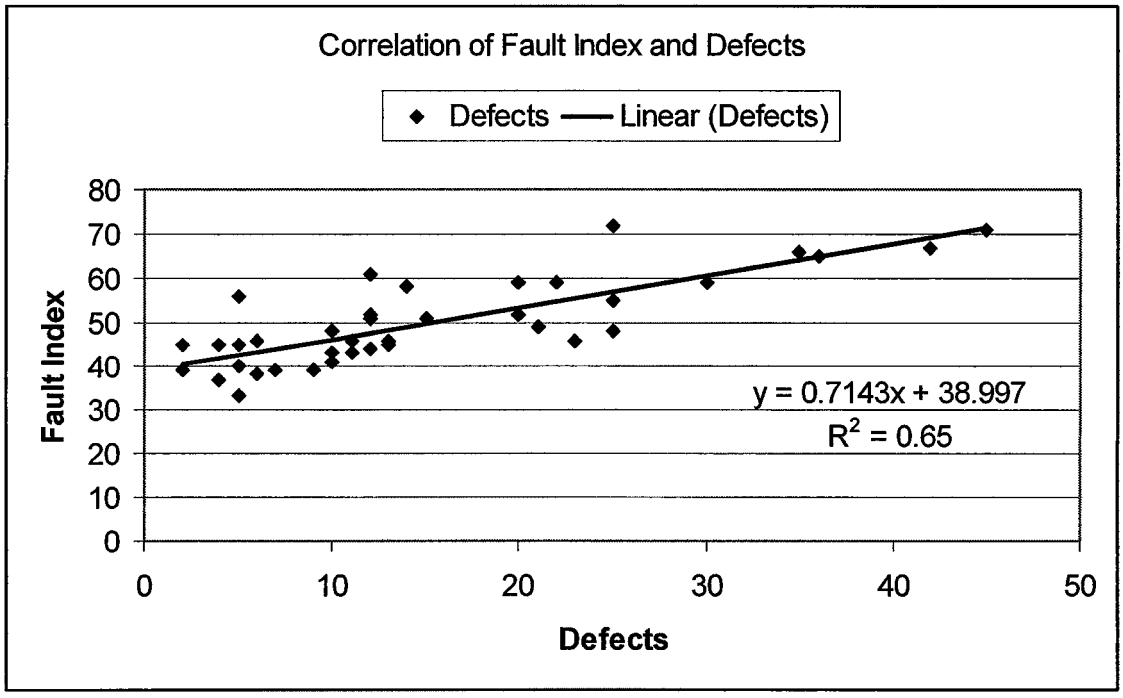

Figure 4.4: Fault Index distribution for all systems

A number of characteristics are important in evaluating this method. First, the derived metric will have little use if not validated (Schneidewind, 1992). Studies of several systems have established relationships between complexity and program faults. Second, a method of combining metrics should be stable. In general, stability is difficult to judge: we need a combination of metrics because it is not easy to rank programs by considering multiple metrics. For a given program pair, if all constituent metrics are higher for one member of the pair, one would expect that the combined metric for this member will also be higher. Data presented by other researchers concluded that the Fault Index metric consistently ranked the programs of each pair into the expected order (Ramamurthy and Melton, 1988). The Fault Index metric exhibits more stability than the single traditional object-oriented metrics. Another study has also established the stability of the Fault Index metric (Munson and Khoshgoftaar. 1990b).

Third, the combining method should be extendable to many independent metrics. More than one hundred software complexity metrics exist. A minimal set of measures needed to capture the most important attributes of programs could consist of more than 44 software complexity measures (Zuse. 1991). The Fault Index metric can use any number of metrics with no change in the basic methodology. Thus, the Fault Index metric is more extendable.

Fourth, because we seek to reduce the difficulty in comparing program complexity, the combining metrics should be effective at reducing the number of total metrics and must be considered. The Fault Index metric reduces the problem of ranking programs using n metrics to a single metric.

Finally, the combining method should be widely applicable. The Fault Index metric can be computed for any group of programs for which complexity metrics can be collected. As presented, static metric synthesis severely restricts the domain of measurable programs. Thus, the combining method restricts itself to a subset of the potential applications in software system development. Furthermore, because programs that use multiple entry, multiple exits, and GOTO constructs are not measurable by the combining method, the method is not applicable in research aimed at testing hypotheses regarding the effects of these constructs on software reliability. Thus, the Fault Index metric is more widely applicable.

4.3.1 Inspection Team Results

Teams involved with the systems that were inspected in this study were asked to provide feedback. Some of the observations are listed here:

- When asked about their general experiences with using software metrics, positive themes were echoed.
- Software metrics were useful by identifying code segments that could be improved.
- Software metrics were useful for identifying good code.
- Overall source code quality was improved.
- The amount of rework and test costs were reduced
- Support staffing levels were reduced

However, another theme was echoed that was not as positive. Even though some source code that could be improved was isolated by using the Fault Index software metric, sufficient resources to make those improvements were not available. This was due in part to support of other programs with major defects that needed to be fixed. Management commitment and some small amount of resources are required to take full advantage of a code review using this software metrics program.

4.3.2 Baseline Results using the Fault Index

Table 4.10 presents *all* of the Fault Index program results. These results were compared with the non-Fault Index program baseline. Clearly, the results of the study using the Fault Index proved valuable. The Fault Index provided significant cost savings and improved quality. Total project effort decreased due to less rework of defects and inspection review rates significantly increased from 40 lines of code per hour to 120. This proves that inspection effectiveness had improved. However, the amount of time spent in inspections increased significantly, which is a good thing. This contributed directly to the reduction in overall effort and rework. In addition, a higher number of defects were found during the inspections thus reducing the number to be discovered by development and test. The average number of defects discovered during inspections increased from 100 to 500. The net result is that fewer defects will be released into the field and consequently not found by the customer. The average number of potential defects shipped to the customer went from 742 to 200.

Metrics	No Fault Index	Fault Index	Units and descriptions
Fault Index Investment	0	12,300	hrs
Labor Rate	$70	$70	$
Size (Lines of Code)	50000	50000	LOC
Total Effort (hrs)	79000	22000	hrs @ SEI Level 3 AIM 85
Inspection Review Rate	44	120	LOC/HR
Avg Injection Rate	4%	3%	Industry Avg 10%
Total Defects	2000	1500	Defects Injected throughout the Lifecycle of the project
Inspection Hours	200	500	Inspection Removal hours Spent
Inspection Hours/Defect	0.7	2	Actual Data from Inspection Tool
Inspection Found Defects	100	500	Defects removed prior to Dev Test
Residual Inspection Defects	1,900	1000	Defects remaining to be found by Dev & Test
Development Removal	385	500	
Remaining to be Found (Test)	1,515	500	
Total Early Defect/KLOC	9.7	20	
Test Efficiency	42%	60%	15 Defects/KLOC to 9 Defects/KLOC
Test Defects	773	300	Defects Found by Test
Test Hours/Defect	13	6	
Test Hours (Finding)	10,357	1800	Estimated based on History
Hours to Fix Defects	7,727	3,000	Avg of 10 hrs to Fix defect if you were to fix All
Costs to Fix	$540,855	$210,000	Costs to Fix defects in Test
Defects Shipped	742	200	Remaining to be found
30% are defects (Sev 1 & 2's)	223	60	
Costs of Support Rework	$1,781,640	$480,000	$8000 / Customer Reported Defect
Potential Latent Defects Remaining	520	140	

Table 4.10: Fault Index baseline comparisons

4.4 Other Significant Results

Other results were calculated as part of the study that aimed at demonstrating that using the Fault Index was valuable and affected the bottom line. In Table 4.11, development productivity improved by 72 percent and consequently the amount of rework improved by 27 percent. This had a positive impact on improved time to market such that the product delivered was earlier than expected. The combination of early delivery and higher quality directly affected the customer satisfaction rating, increasing it by 5 percent.

Measurement	Rating	% Improvement
% Savings from Rework by:		27%
Improved Customer SAT	87 to 94	5%
Productivity Increase by:	11 to 19	72%
Reduced Cycle Time by:	13 to 8 months	60%

Table 4.11: Percentage improvements to management

4.4.1 Amount of Time Required to Review Code

One expected established measurement is the inspection review rate. It stated that team members could spend less time reviewing certain sections of source code denoted as not requiring an intensive review (i.e. a lower Fault Index), and

instead would be able to spend more time reviewing other critical sections. The team anticipated an overall drop in total time required to review code, because the number of 'good' code sections outnumbered the number of 'bad' ones.

Data was routinely collected for the code inspections, which included both the number of person-hours spent reviewing source code and the number of reviewers at a particular code review. Using this data, a measurement for each code review (those that used the Fault Index and those that did not) was established. This measurement, called *review speed* (rs) was established to show the number of seconds each reviewer spent, on average, reviewing each line of code. The formula for rs is stated as:

$rs = (t / p) / LOC$

Where:

t is the total time spent reviewing the code.

p is the total number of source code reviewers.

LOC is the number of Lines Of Code reviewed.

The average rs at code reviews not using software metrics was measured at 9.34 seconds per LOC. The average rs at code reviews that were augmented with the Fault Index data were measured at 3.46 seconds per LOC. The results of the measurements indicate that each code reviewer spent on average, 63 percent less time reviewing code using the Fault Index than not using it. Using the Fault

Index during the review to concentrate scrutiny may account for part of the improvement, but also part of this improvement may reflect the fact that source code is being improved with the help of the Fault Index prior to reviews, so it may be easier to understand during the review. No one disputed the possibility that other factors may be involved in the large difference in reviewing time, but it would appear that the Fault Index had a positive effect upon the amount of time spent reviewing source code.

4.4.2 Effectiveness of Source Code Reviews

When the Fault Index measurement was evaluated during the pilot, each team member described how source code review effectiveness was changed. Of those team members who reviewed the code using the Fault Index, all but one felt that the Fault Index provided a small, positive improvement in their overall code review effectiveness. In addition, the Fault Index education was cited as a major factor in improving this effectiveness.

As shown in Table 4.10, defects found per inspection hour went from .7 defects to 2. This was another major improvement on how effective code inspections were, compared to what had been previously achieved.

When asked whether too little, too much or just the right amount of data using the Fault Index metric information was distributed with each of the reviews, half of the team thought that too much material was distributed, and the other half thought that the amount of material was just right. Not one member thought that too little information was distributed. It is recommended that future usage of the Fault Index metric program prior to an inspection should start slowly, using only a few software metrics, and working up to more software metrics over time.

4.4.3 Source Code Quality

Table 4.12 lists some additional data that collected and measured during this study. Early indications show that using the Fault Index has provided significant improvements in many areas. A fault per inspection hours was measured and it represented how many critical defects were removed prior to development and test. A 185 percent improvement was noted. In addition, test had reported a significant reduction in the number of test found defects. The number of test found defects was reduced by 33 percent. Prior to using the Fault Index, an average of 15 defects/KLOC were typically discovered by test and now that number has declined to 9 defects/KLOC.

Early defect detection numbers were noted as rising from 9 defects/KLOC to 20 defects/KLOC. This has two positive impacts to the organization. First, this improvement reduces costly rework and provides a huge cost savings. Second, delivery time to the customer is reduced resulting in a faster time to market. Table 4.12 clearly shows that the cost of finding and fixing defects earlier in the lifecycle is significantly less than finding and fixing them later. The mean time between failures (MTBF) increased from 8 hours to 20. That is a considerable 150 percent improvement in quality. Because of that, each member truly felt that the source code quality dramatically improved. Overall, defect removal efficiency reported by all process areas improved dramatically. Finally, the most essential critical element of the code quality analysis is the perceived quality in the field by the customer.

Currently, customer reported defects prior to using the Fault Index was at an all time high. After the implementation of the Fault Index, the maintenance team reported a dramatic decrease in the number of problem reports within the first month of delivery. That is an astounding 76 percent improvement with field defect quality going from 5 defects/KLOC to 1.2 defects/KLOC.

Results	Fault Index	No Fault Index	% Improvement
Defects/KLOC(Test)	9	15	40%
Faults/Inspection Hour	2	0.7	185%
Inspection Rate (LOC/hr)	120	44	173%
MTBF	20	8	150%
Field Defects/KLOC	1.2	5	76%
Defect Efficiency (%):			
Pre-Test	12%	5%	140%
Test	65%	50%	30%
Overall	95%	45%	111%

Table 4.12: Percentage improvement yields

4.4.4 Quality Issues Identified

Figure 4.6 is a Pareto chart created to plot some of the quality issues identified during the inspection. Normally, these issues are not discovered due to time constraints when reviewing the source code. Exposing these issues may not have been possible had we not used the Fault Index. The Together tool has a built in mechanism that searches, ranks and reports quality issues. Table 4.13 defines each quality issue and ranks them by High, Medium, and Low.

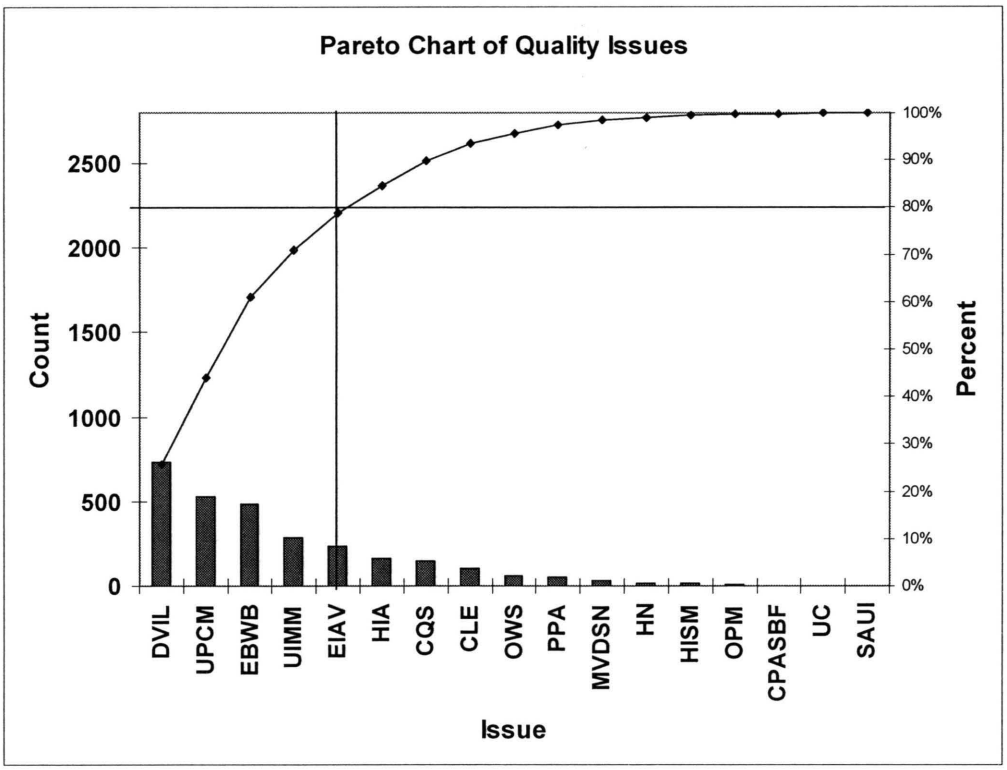

Figure 4.5: Pareto chart of discovered quality issues

As shown in the figure, over 2000 quality issues were reported. The most notable was Declaring Variables Inside Loops (DVIL). Most of the issues identified were corrected as time permitted. Again, standard thresholds were used and when those modules exceeded that threshold, it was reported. Next, the issues were counted and placed in a table. Table 4.14 shows each system's issues ranked and normalized by KLOC. System 2 had 819 quality issues identified followed by system 1 with 536. Combining this data with the Fault Index, a scatter plot graph was created. Figure 4.7 shows the Fault Index plotted with the quality issues.

Once this data was plotted, we were clearly able to identify code modules that were at high risk to fail.

CLE	Complex Loop Expression (L)
CPASBF	Constant Private Attributes not Final (H)
CQS	Command/Query separation (H)
DVIL	Declaring Variables inside Loops (H)
EBWB	Enclosing body within block (L)
EIAV	Explicitly Initialize all variables (L)
HIA	Hiding Inherited Attributes (H)
HISM	Hiding Inherited Static Methods (H)
HN	Hiding Names (H)
MVDSN	Multiple Visible Declarations with the same name (H)
OPM	Overriding Private Methods (H)
OWS	Overriding System Function (H)
PPA	Public and Private Attributes Mixed (M)
SAUI	Static Attribute Used for Initialize (H)
UC	Unnecessary Casts (H)
UIMM	Unnecessary Interface Members Modified (H)
UPCM	Unused Private Class Member (H)

Table 4.13: Quality issues list

System	High	Med	Low
System1	536	21	21
System2	819	14	134
System3	112	17	111

Table 4.14: Quality issues per thousand lines of code

In the upper right region, those circled data points are code modules that are at the highest risk for failure. Those components are some of the first ones to be selected for an inspection due to their high-risk fault-prone potential.

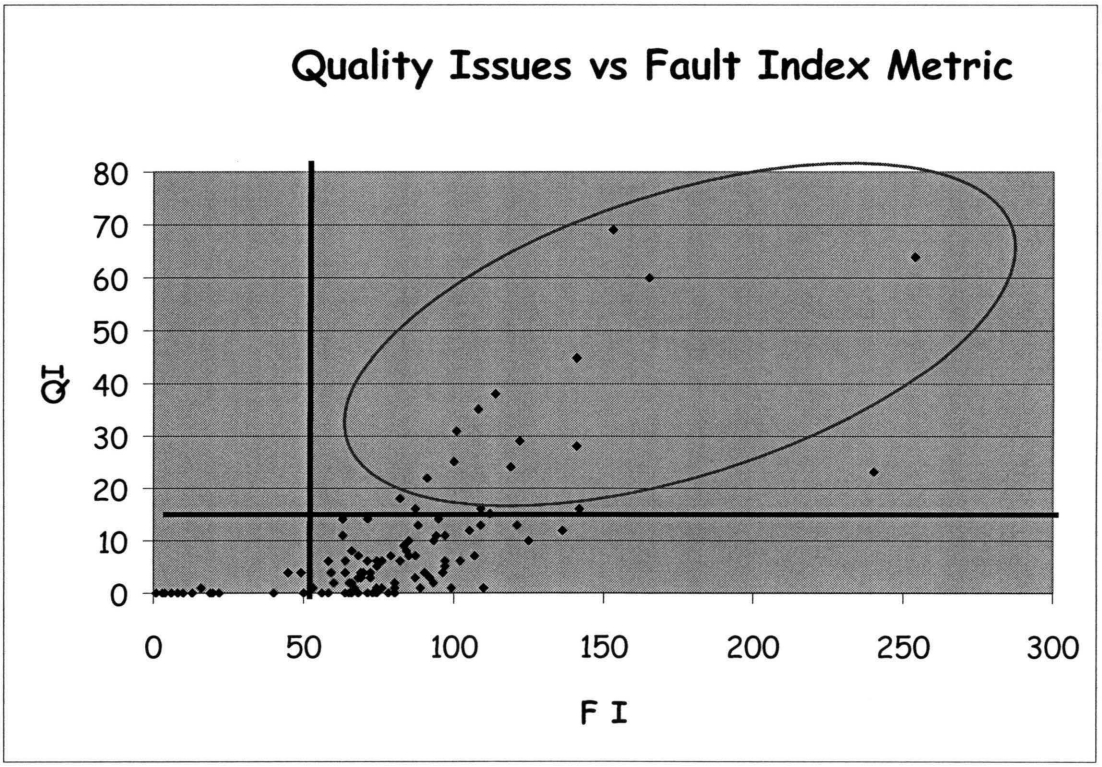

Figure 4.6: Quality issues correlated with the Fault Index

4.5 Summary

The consensus and overall reaction of the team was optimistic. They concluded that by using the Fault Index to rank and identify high-risk class modules, positively enhanced code reviews, and improved the source code. The teams experienced a positive *return-on-investment*, so to speak, during the pilot.

Given the experience that the teams gained during the first Fault Index metrics pilot, the following steps to adopting a code inspection and using it has been created. Interested development teams could use these steps either as a rigorous or flexible guide.

1. Obtain management support. One of the more poignant lessons from the pilot is that even though source code modification to improve code quality was specified for using the Fault Index metric for code reviews, human and time resources were not readily available to perform this work. Some amount of resources should be granted for taking advantage of the information learned during the code reviews.
2. Decide what metrics are available for you to use in order to construct the Fault Index. This study recommends using the Fault Index as a primary key metric to select code candidates for an inspection. As you grow and learn using this metric, expand to other metrics. Remember, anything that measures code is considered a software metric. Something as simple as

Lines of Code (LOC) or Cyclomatic complexity V (g) can be extremely useful to some degree in program understanding. Some metrics are applicable only to a particular programming language or family of programming languages. Some commonly useful software metrics include, but are not limited to:

- SSI: the number of Shipped Source Instructions
- CSI: the number of Changed Source Instructions
- LOC: the number of Lines of Code
- Comment density: the ratio of comments to code
- Nesting level: the number of nested constructs
- Software science: various measures from source volume to difficulty to estimated time to complete
- Logical complexity: the number (and sometimes also the density) of decision statements

3. Decide the method and tools to collect and present metrics. Once you have selected the metrics to review the source code, the next step is to find or create a tool that automatically collects the information for you. Manual collection of software metrics is possible, but tends to be both tedious and error-prone.

4. Determine exceptional metric values, if appropriate. Teams that adopt a software metrics program may wish to accept information pertaining to what a 'good' or a 'bad' metric value is for each of the metrics collected. See Appendix A for more information.
5. Determine the scope of the code reviews and which weakly correlated software metrics will be used. In the beginning, you may opt not to review all of your code content using the Fault Index. For instance, you may opt to not review non-source code, like messages or help text, using software metrics. The advantage to limiting the scope of the initial program would be to ease into it slowly.
6. Hold source code reviews. Whatever normal code review process you currently have in place today can still be used. Simply distribute the Fault Index metrics information with the source code. The Fault Index metrics information should only enhance and not replace your existing process. Source code authors should review the code before the review. Source code reviewers should be instructed to scan the Fault Index information and locate those sections of the source code that need meticulous review and those that need only be skimmed.
7. Hold periodic checkpoint meetings. Two or three software metric checkpoint meetings should be held during the development cycle to

ensure that all team members have the ability to effectively utilize the software metrics you chose to use. Additionally, any concerns or problems that may arise in your program of adopting the Fault Index for code reviews could be made known and action plans could be created.

Enhancing source code reviews using the Fault Index is an inspiring concept. The pilot study explored both the benefits and drawbacks and reached a positive conclusion relating to its efficacy. At the end of the first development cycle, the team discovered a number of positive benefits, and pitfalls that could be avoided.

The pilot team discovered that the Fault Index metric improved code review effectiveness, reduced the amount of time required to review the code, and substantially improved source code quality. Finally, both education and resource are cited as being top requirements to an effective Fault Index metrics program. The development team plans to extend the pilot into the next development cycle of their product. Education and resources will both be provided to further improve and enhance the effectiveness of code reviews.

APPENDIX A

A.1 Statistical Analysis of Source Code

Software metrics typically produce 'raw' numbers that are difficult to interpret. A successful software metrics program must satisfy two requirements to alleviate this difficulty:

- Provide education on how to interpret the numbers. For any particular software metric, is a larger or smaller number better? The cyclomatic complexity metric is related to source code complexity, but both small and large numbers can be argued to be less advantageous for average identifier length. A good software metric tool will give information on how to interpret the numbers. Experimentation will provide some education and experience on interpreting software metrics.
- Provide information on what is a 'good' software metric value and what is a 'bad' one. Providing education would enable a person to compare two software metric values, but what is also needed is to know from a set of software metric values which represent code that needs further attention (a 'bad' software metric value,) and which do not. Some software metrics

come 'packaged,' so to speak, with recommended bounds. E.g., sections of source code should not exhibit a cyclomatic complexity with numbers greater than ten (McCabe, 1976). However, other software metrics do not come prepackaged with recommended boundaries, and even the recommended boundaries may not be applicable to all projects, programming languages or people.

The following addresses techniques that may be employed to satisfy the second requirement. The first requirement may be satisfied through experimentation, expert instruction, or through examination of the original software metric proposal documents.

Three methods for determining exceptional software metric values are presented; one formal and two informal. The formal method is provided for reference only. Only the two informal methods were employed during the pilot.

Formal Method

The formal method is commonly used in theory and should be somewhat familiar- statistical standard deviation. Using this method, existing source code is measured and standard deviations are produced from the results. The advantage to this approach is that very accurate bounds between 'good' and 'bad' software metric values can be obtained. The disadvantages are that source code must exist to measure, and that standard deviation can be difficult to compute.

When establishing software metric bounds, ensure that you are initially measuring source code that is at least approximately similar to that which is to be later reviewed. Typically, different bounds must be established for each new project, programming language, or even personnel involved.

For code reviews, identifying the standard deviations of the code metrics is important and should be noted. After computing the standard deviation values, decide what value of σ is to be acceptable or unacceptable. A value of $\sigma = 1$ would yield approximately one-third of the measured source code with 'bad' values. A value of $\sigma = 2$ would yield approximately five percent of the measured source code with 'bad' values. A value of $\sigma = 3$ would place the 'bad' software metric boundary well below one percent. A value mid-way between $\sigma = 2$ and $\sigma = 3$ should be acceptable.

This value should then be published with the software metrics themselves for code reviews. Reviewers would be then be able to notice 'bad' software metric values and concentrate attention on the corresponding sections of source code.

As an example of formal standard deviation computation, consider the following software metric results for *software science difficulty*:

0.7	8.3	8.5	3.0	3.9
23.8	55.9	34.8	4.3	27.6
7.1	18.0	9.6	33.2	8.1
19.9	28.5	14.0	37.2	6.9
12.7	24.2	15.9	20.1	17.3

Table A.1 Software Science standard deviation results

In table A.1, the computed value for σ=12.89, with a mean $\bar{x} = 17.74$. For the purposes of this example, let us assume a standard deviation boundary at 2.5σ. Therefore, the software science difficulty boundary would be the sum of the mean and the product of 2.5 and σ. Thus, the software science difficulty upper bound is set at 49.97. Given the input data set, only one measurement is exceptional: 55.9. During future code reviews, any source code segment exhibiting a software science difficulty value above 49.97 should be scrutinized more carefully.

Optionally, you may wish to pursue the opposite scenario in your software metrics program. You could establish a series of bounds of 'fantastic' software metric values. Given the example data set, let us assume that any source code with a software science difficulty measure more than one standard

deviation less than the mean might need only cursory examination during reviews. With this example, a 'fantastic' bound of 4.85 could be established, and four of the example software science difficulty measurements would qualify.

Informal Methods

Two informal methods of establishing software metric bounds were employed during this pilot. Neither is steeped in scientific proof, nor are they statistically verifiable. However, they did seem to work fairly well and were both very easy to use.

Informal Method #1 (Eyeball)

Using this method, software metrics measurements are collected as in the above formal method. Instead of computing a standard deviation, the measurements are sorted in ascending order. These sorted measurements are then examined by eyeball. Either the measurements can be graphed using any of a variety of graphing program (such as by using a spreadsheet program), or just the raw numbers can be examined. Almost invariably, the measurements collected for the pilot exhibited the following conditions:

A definite break occurred between very acceptable numbers, and less acceptable numbers. A definite break occurred between the less acceptable numbers and the unacceptable numbers. The measurements generally followed the form as graphed in figure A.1.

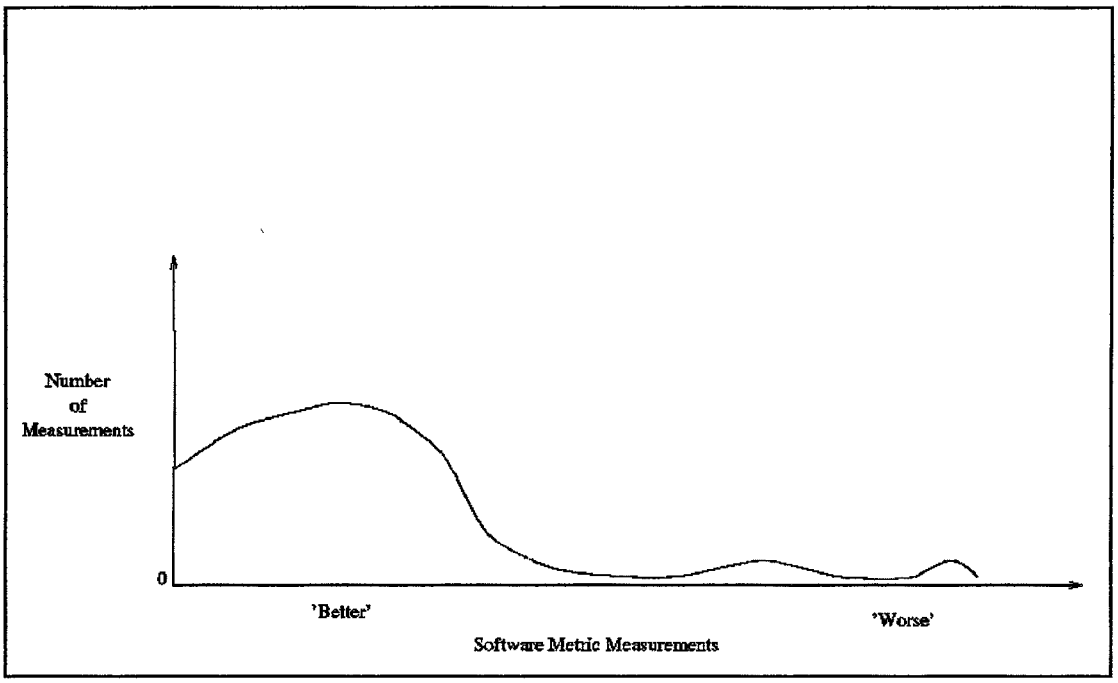

Figure A.1: Graphical Form of Pilot Software Metric Measurements

This is the basic form of the software metric measurements seen during the pilot. The left-most part of the graph is normal. The first peak represents 'grey' measurements, and the second represents 'black' measurements. Breakpoints to start the grey and black areas are established somewhere in the valleys. On infrequent occasions, the 'grey' area was not readily discernible and only a 'black' measurement was established.

We established a grey and a black area for each software metric. As indicated on the graph, the grey area indicated measurements that represented source code that should be examined more closely during code reviews. The black area indicated measurements that represented source code that not only should be

examined more closely during code reviews, but should also be the target of a serious attempt at source code modification with the explicit purpose of improving source code quality. The remaining area (white) indicated measurements that the team felt to be acceptable. Source codes with measurements at the end of the graph furthest from the grey and black areas were candidates for skimming. Once computed, the grey and black boundary measurements were published so that code reviewers could access them while reviewing the software metrics and source code.

The advantages behind the first eyeball method is that it is much easier to compute than by using standard deviation, and it is fairly reliable. The disadvantage stems from a lack of accuracy. For the purposes of the pilot, this disadvantage did not seem overly severe, however.

As an example, assume the same software science difficulty measurements provided in Formal Method, which sorted are:

.7	3.0	3.9	4.3	6.9
7.1	8.1	8.3	8.5	9.6
12.7	14.0	15.9	17.3	18.0
19.9	20.1	23.8	24.2	27.6
28.5	33.2	34.8	37.2	55.9

Table A.2 Informal method standard deviation results

In table A.2, either by eyeball or by graphing, note where the possible breakpoints are. Very generally, it seems that the breakpoints exist at about 30, and again at about 40. A general graph is provided in figure A.2. Thus, we would say that software science difficulty measurements between 30 and 40 represent the grey area, and measurements greater than 40 represent the black area. Given the provided software science difficulty measurements, three are grey and one is black.

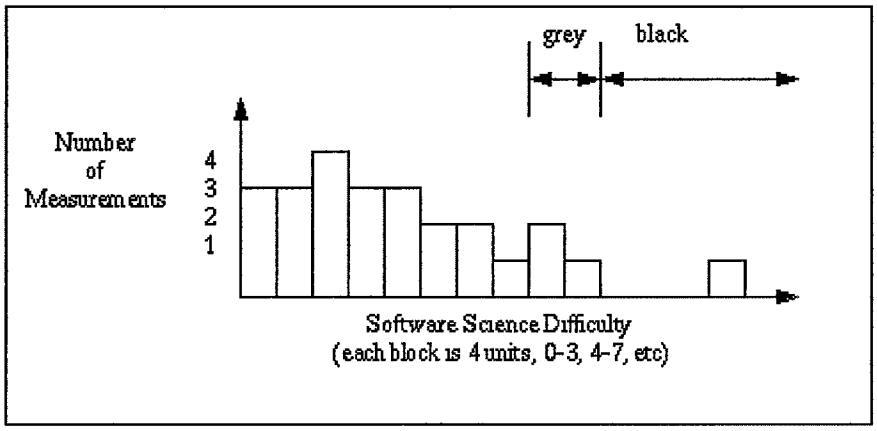

Figure A.2: Graph of Example Data

The example software science difficulty is graphed. Notice the 'bump' in the curve. This bump signals the start of the grey area. The other bump at about 55 is obvious, and belongs within the black area. With more data points, the grey and black boundary points oftentimes become more pronounced.

Informal Method #2 (Other Eyeball)

Using this method, software metrics measurements are not collected as in the other two methods. Rather, the list of software metric measurements provided for the code review is quickly scanned, and any apparently exceptional metrics are noted. If one were given the example software science difficulty measurement, the measurement of 55.9 should stand out. Perhaps the measurements in the 30's would also stand out. This method is very informal, but implies some rather significant advantages and disadvantages.

One advantage is that software metric measurement selection occurs very quickly. Instead of spending a great deal of time computing standard deviation, or sorting and perhaps graphing data, the results are immediately obtainable. Another advantage is that no historic information need exist. Before, existing and similar source code is measured to establish bounds. If no source code exists that closely matches the new code, this informal method would still allow some degree of assurance that sections of source code with exceptional software metric measurements would be viewed more closely.

The disadvantages are that the evaluation criteria are very inaccurate. What may be an exceptional software metric measurement for one set of reviewed source code may not be exceptional for another, even though both sets are written in the same programming language, by the same programmer, or

perform similar or complementary tasks. For instance, if lines of code were used as the software metric, one code review could contain small utility functions of 10 LOC or less each, and one main function of 100 LOC. Another code review could contain several sorting functions of 80-120 LOC each. The exceptional function of 100 LOC in the first review would not even be noticed in the second. This disadvantage would be minimized with reviewer experience, however.

APPENDIX B

B.1 Metrics Definitions

SSI: Shipped Source Instructions: the number of Shipped Source Instructions that we produced and delivered to the end user.

Nesting level: Counts the maximum depth of if, for and while branches in the bodies of methods. Logical units with a high number of nested levels might need implementation simplification and process improvements, because groups that contain more than seven pieces of information are increasingly harder for people to understand in problem solving.

Response for Class: RFC is the number of methods, internal and external, available to a class (Chidamber and Kemerer, 1994). This measure is calculated as 'Number of Local/Internal Methods' + 'Number of Remote/External Methods'. The enumeration of methods belonging to a class and its parent, as well as those called on other classes, indicates a degree of complexity for the class. RFC differs from CBO; Coupling Between Objects, in that a class need not utilize a large number of remote objects in order to have a high RFC value. Consider a class with a large number of exposed methods that is utilized by a client class

extensively. In this case, the client class RFC value would be high while its CBO could be one. Distinguishing CBO as a coupling measure and RFC as a complexity measure is important. Many software metrics have relationships and/or similarities to other metrics. As RFC is directly related to complexity, the ability to test, debug and maintain a class increase with an increase in RFC. In the calculation of RFC, inherited methods count, but overridden methods do not. This makes sense, as only one method of a particular signature is available to an object the class. In addition, only one level of depth is counted for remote method invocations.

Coupling Between Objects or CBO: Object-oriented design necessitates interaction between objects. Excessive interaction between an object of one class and many objects of other classes may be detrimental to the modularity, maintenance and testing of a system. Coupling Between Objects counts the number of other classes to which a class is coupled, save inherited classes, java.lang.* classes and primitive types (Lee, Liang, Wang, 1993). A decrease in the modularity of a class can be expected with high values of CBO. However, some objects necessarily have a high degree of coupling with other objects. In the case of factories, controllers, and some modern user interface frameworks, one should expect a higher value for CBO.

An important note about the implementation of the CBO metric is that it only counts each type used in a class once. Multiple usages of a class as referenced, thrown as an exception, or passed as a formal parameter are only counted at their first occurrence. As mentioned, parent classes do not count, as is the case for implemented interfaces. There is an implicit high degree of coupling in these cases.

Cyclomatic Complexity: This measure represents the cognitive complexity of the class. It counts the number of possible paths through an algorithm by counting the number of distinct regions on a flow graph, meaning the number of *if*, *for* and *while* statements in the operation's body. Case labels for switch statements are counted if the *Case as branch* box is checked.

A strict definition of Fault Index looks at a program's control flow graph as a measure of its complexity:

$$CC = L - N + 2P$$

where L is the number of links in the control flow graph, N is the number of nodes in the control flow graph, and P is the number of disconnected parts in the control flow graph.

For example, consider a method that consists of an *if* statement:

```
if (x>0) {
   x++;

} else {

   x--;

}
```

$CC = L - N + 2P = 4 - 4 + 2*1 = 2$

A less formal definition is:

$$CC = D + 1$$

where D is the number of binary decisions in the control flow graph, if it has only one entry and exit. In other words, the number of *if*, *for* and *while* statements and number of logical *and* and *or* operators.

For the example above:

$CC = D + 1 = 1 + 1 = 2$

Software Science Difficulty: This set of metrics is based upon the work of Maurice Halstead who proposed that source code complexity is fundamentally based upon the usage of the operators and operands of a programming language. Using the total and unique numbers of operators and operands, Halstead offers several metrics, which provide measurements of source code for its size, difficulty, average time for development, and other factors.

Length: The length of a piece of source code is the sum of total operators and operands. This software science metric is directly related to the size of the piece of source code it is measuring.

Volume: The volume of a piece of source code, like its length, is directly related to the size of the piece of source code it is measuring. Unlike length, however, the volume is also directly related to the uniqueness of the components of the source code.

Difficulty: The difficulty of a piece of source code is proposed to be a measure of its complexity. The more complex a piece of source code, the greater its difficulty measure.

Effort: The effort of a piece of source code is proposed to be the relative amount of work that would be required to produce the piece of source code. It is simply the product of the source code's volume and difficulty

Time: The average amount of time required to develop a piece of source code. Time is directly related to Effort. The above number represents the number of HOURS required to develop the source files by a person of average skill given average working conditions. This number includes time to develop code, but not time to comment it.

It is calculated as ('Number of Unique Operators' / 2) * ('Number of Operands' / 'Number of Unique Operands').

Logical Complexity- based upon logical attributes of the input source both. Both the absolute number of logical decisions and their frequency are measured. This metric is based upon the proposal that the more decisions a source file contains, and the greater the density of those decisions, the more complex it is.

Call Tree- This statistic lists, for each module of the measured source files, the number of called modules, their names, and the source code locations of their calls.

VOD - Violations of Demeters Law: Law of Demeter (Lieberherr and Holland, 1989). The definition of this metric is based on the minimization form of the Law of Demeter. Based on the concepts defined there, and remembering that the minimization form of Demeters Law requires that the number of acquaintance classes should be kept low, we define the VOD metric.

Definition 1 (Client) Method *M* is a client of method *f* attached to class *C*, if inside *M* message *f* is sent to an object of class *C*, or to *C*. If *f* is specialized in one or more subclasses, then *M* is only a client of *f* attached to the highest class in the hierarchy. Method *M* is a client of some method attached to *C*.

Definition 2 (Supplier) If *M* is a client of class *C* then *C* is a supplier to *M*. In other words, a supplier class to a method is a class whose methods are called in the method.

Definition 3 (Acquaintance Class) A class *C1* is an acquaintance class of method *M* attached to class *C2*, if *C1* is a supplier to *M* and *C1* is not one of the following:

the same as *C2*;

a class used in the declaration of an argument of *M*

a class used in the declaration of an instance variable of *C2*

Definition 4 (Preferred-acquaintance Class) A preferred-acquaintance class of method *M* is either:

A class of objects created directly in *M*, or

A class used in the declaration of a global variable used in *M*.

Direct creation means that a given object is created via operator *new*.

Definition 5 (Preferred-supplier class) Class *B* is called a preferred-supplier to method *M* (attached to class *C*) if *B* is a supplier to *M* and one of the following conditions holds:

B is used in the declaration of an instance variable of *C* ,

B is used in the declaration of an argument of *M*, including *C* and its super classes.

B is a preferred acquaintance class of *M*.

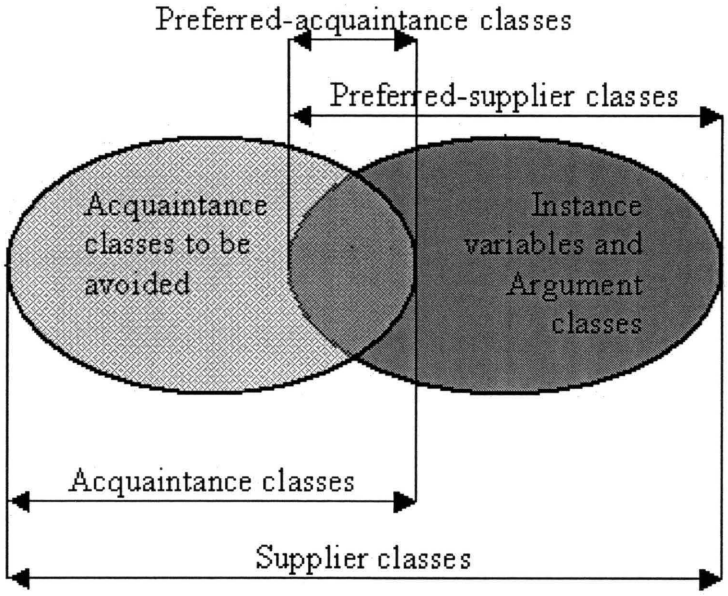

Figure B.2: The relation among the different types of supplier classes

The class form of Demeters Law has two versions: a strict version and a minimization version. The strict form of the law states that every supplier class of a method must be a preferred supplier.

The minimization form is more permissive than the first version and requires only minimizing the number of acquaintance classes of each method.

Observations:

The motivation behind the Law of Demeter is to ensure that the software is as modular as possible. The Law effectively reduces the occurrences of certain nested message sends and simplifies the methods.

The definition of the Law makes a difference between the classes associated with the declaration of the method and the classes used in the body of the method, i.e. the classes associated with its implementation. The former includes the class where the method is attached, its super classes, the classes used in the declarations of the instance variables and the classes used to declare the arguments of the method. In some sense, there are 'automatic' consequences of the method declaration. They can be easily derived from the code and shown by a browser. All other supplier classes to the methods are introduced in the body of the function that means these couples were created at the time of concretely implementing the method. They can only be determined by a careful reading of the implementation.

Definition 6 (VOD Metric) being given a class *C* and *A* the set of all its acquaintance classes.

VOD(C) = |A|

Informally, VOD is the number of acquaintance classes of a given class.

Keeping the VOD value for a class low offers a number of benefits, enumerated below:

Coupling control- A project with a low VOD value is the sign of minimal "use" coupling between abstractions. That means that a reduced number of methods can be invoked. This makes the methods more reusable.

Structure hiding- Reducing VOD represents in fact the reducing of the direct retrieval of subparts of the "part-of" hierarchy. In other words, public members should be used in a restricted way.

Localization of information- A low VOD value also means that the class information is localized. This reduces the programming complexity.

Inter-Module Complexity: The Henry-Kafura inter-module complexity metric was designed to measure the connectivity of a module within a source file (Henry and Kafura, 1981). This metric has been found to correlate well with maintenance effort. The number of input and output parameters is reflected in this metric. Basically, the fewer parameters a module has and the fewer parameters that are used in calls to other modules, the less connected it is with other modules, and the less likely errors are to propagate throughout source code. The inter-module complexity is directly related to the interconnections a module has with other modules. Look for modules with exceptionally high inter-module complexity measures. The complexity of these modules can be reduced by removing some of the interconnections (parameters). Parameters can also be reduced in a module by splitting it up into two or more smaller modules, each using fewer parameters, and calling fewer modules.

REFERENCES

Ackerman, A.F., Fowler, P.J, and Ebenau, R.G. (1983, Sept 26-30). Software Inspections and the Industrial Production of Software. Proceedings of the Symposium on Software Verification and Validation, Darmstadt, Germany

Basili, V., Perricone, B. (1984). Software Errors and Complexity: An Empirical Investigation. Communications ACM, (Vol. 27), 42-52.

Buck, F.O. (1981). Indicators of Quality Inspections. IBM Technical Report TR21.802, Systems Communications Division, Kingston, NY.

Chidamber, S., and Kemerer, C. (1994). A Metrics Suite for Object Oriented Design. IEEE Transactions on Software Engineering, (Vol. 20) Number 6, 476-493

Crossman, T.D. (1979). Some experiences in the use of inspection teams in application development. Process Application Development Symposium.

Fagan, M. E., (1976). Design and Code Inspections to Reduce Errors in Program Development IBM System Journal, (Vol. 15, No. 3)

Fagan, M. E., (1986). Advances in Software Inspections. IEEE Transactions on Software Engineering, (Vol. SE-12, No. 7)

Fenton, L.H. (1984). Response to the SHARE software service task force report. IBM Corp., Kingston, NY

Freedman, D. P., Weinberg, G. M. (1982). Handbook of Walkthroughs, Inspections, and Technical Reviews: Evaluating programs, projects, and products (3rd ed.). Boston: Little, Brown and Co.

Gilb, T., Graham, D. (1993). Software Inspection, Addison Wesley.

Glass, R. (2002). Sorting Out Software Complexity. Communications ACM, (Vol. 45), 19-21

Graden, M. E., Horsley, P. S., and Pingel, T. C., (1986). *The Effects of Software Inspections on a major Telecommunications-project.* AT&T Technical Journal, 65(3):32-40.

Gupta, V., Patnaik, A., Emam, K., Goel, N. (1994). A System for Controlling Software Inspections

Halstead, M.H. (1977). *Elements of Software Science,* Elsevier North-Holland, New York

Henry, S. and Kafura, D. (1981). Software Structure Metrics based on Information Flow. IEEE Trans. on Soft Eng. 7, (pp. 509-518).

Humphrey, W. S. (1995). A discipline for software engineering Addison-Wesley Publishing Company

IBM, (1977). Inspections in Application Development-Introduction and Implementation Guidelines. IBM Technical Publications GC20-2000, July

IEEE (1983). Glossary of Software Engineering Terminology. ANSI/IEEE Std 729-1983, New York, NY.

Jackson, J. E. (1991). *A User's Guide to Principal Components.* John Wiley and Sons, Inc., New York, New York.

Johnson, R.A., and Wichern, D.W. (1992). *Applied Multwariate Statistical Analysis.* Prentice-Hall. Englewood Cliffs.

Johnson, P. M. and Tjahjono, D. (1998). Does every inspection *really* need a meeting? Empirical Software Engineering: An International Journal, 3, 9-35.

Khoshgoftaar, T., Munson, J., Lanning, D. (1994). Alternative Approaches for the use of Metrics to Order Programs by Complexity. Journal of Systems and Software, (Vol. 24).
Lake, A., Cook, C. (1994). Use of Factor Analysis to Develop {OOP} Software Complexity Metrics. Technical Report

Land, L. P. W., Sauer, C., Jeffery, R. (1997a). Validating the defect detection performance advantage of group designs for software reviews: Report of a laboratory experiment using program code, Proceedings of the 6th European Software Engineering Conference held jointly with the 5th ACM SIGSOFT Symposium on the Foundations of Software Engineering, Zurich, Switzerland, 22-25 September, Lecture Notes in Computer Science 1301, (Eds.) Goos, G., Leeuwen J.V., (pp. 294-309), Springer-Verlag.

Land, L. P. W., Jeffery, R., Sauer, C. (1997b). Validating the defect detection performance advantage of group designs for software reviews: Report of a replicated experiment, 1997 Australian Software Engineering Conference, September 29 - October 2, Sydney, Australian, 17-26, IEEE Computer Society.

Larson, R.R. (1975). Test plan and test case inspection specification. IBM Corp., Technical Report (TR 21.585)

Lee, Y., Liang, B., Wang, F. (1993). Some Complexity Metrics for Object Oriented Programs Based on Information Flow. *Proceedings: CompEuro,* (pp. 302-310), March

Lieberherr, K., Holland, I. (1989) Assuring Good Style for Object Oriented Programs, *IEEE Software,* (pp 38-48), Sept

Marinescu, I.R. (1998). An Object Oriented Metrics Suite on Coupling. Universitatea "Politehnica" Timisoara, Facultatea de Automatica si Calculatoare, Departamentul de Calculatoare si Inginerie Software

Mayer, T., Hall, T. (1999). A Critical Analysis of Current Object Oriented Metrics. Software Quality Journal, 8, 97-110

McCabe, T. (1976). A Complexity Measure. IEEE Trans. on Soft. Eng., (pp.308-320).

Munson, J. C., and Khoshgoftaar, T. M. (1989). The dimensionality of program complexity, in *Proceedings of the Eleventh International Conference on Software Engineering,* (pp. 245-253).

Munson, J. C., and Khoshgoftaar, T. M. (1990a). Applications of a relative complexity metric for software project management. Journal of Systems and Software, (Vol. 12), 283-291.

Munson, J. C., and Khoshgoftaar, T. M. (1990b). The relative software complexity metric: A validation study in *Proceeding of Software Engineering 90,* (pp. 89-102).

Munson, J. C., and Khoshgoftaar, T. M. (1991a). Some primitive control flow metrics, in *Proceedings of the Annual Oregon Workshop on Software Metrics.* Silver Falls, Oregon, March

Munson, J. C., and Khoshgoftaar, T. M. (1991b).The use of software complexity metrics in software reliability modeling, in *Proceedings of the International Symposium on Software Reliability Engineering,* (pp. 2-11).

Munson, J.C. (1992). Measuring dynamic program complexity. Technical Report

Munson, J. C., and Khoshgoftaar, T. M. (1992). The Detection of Fault-Prone Programs. *IEEE Trans. Software Eng.* 18. (pp. 423-433).

Peele, R. (1998). Code Inspections at First Union Corporation. Proceedings of COMPSAC, (pp. 445-446).

Porter, A. A., Votta, L. G. (1994). An experiment to assess different detection methods for software requirements inspections, Proceedings of the 16th International Conference on Software Engineering, Sorrento, Italy, 103-112, May.

Porter, A. A., Votta, L. G.; Basili, V. R. (1995). Comparing detection methods for software requirements inspections: A replicated experiment, IEEE Transactions on Software Engineering, 21(6), 563-575, June.

Porter, A. A., Johnson, P. M. (1997). Assessing software review meetings: Results of a comparative analysis of two experimental studies, IEEE Transactions on Software Engineering, 23(3), 129-145, 31 March.

Porter, A. A., Siy, H. P., Toman, C. A., Votta, L. G. (1997). An experiment to assess the cost benefits of code inspections in large-scale software development, IEEE Transactions on Software Engineering, 23(6), 329-346, June.

Ramamurthy, B. and Melton, A. (1988). A Synthesis of Software Science Measures and the Cyclomatic Number. *IEEE Trans. Software Eng.* 14. 1116-1121

Rico, D. F. (2002) Software Process Improvement: Modeling Return on Investment (ROI) Pittsburgh: National SEPG Conf., SEI

Sauer, C., Jeffery, R., Land, L. P. W., Yetton, P. (1996). A behaviorally motivated program for empirical research into software development technical reviews, Technical Report 96/5, Centre for Advanced Empirical Software Research, School of Information Systems, University of New South Wales, Sydney 2052. Also, to be published in the IEEE Transactions on Software Engineering, 1999.

Schneidewind, N.F., (1992). Methodology for Validating Software Metrics. *IEEE Trans. Software Eng.* 18. 410-421

Stark, G., Lacovara, R. (1991). On the Calculation of Relative Complexity Measurement. MITRE Corp. GeoControl Systems, Houston

Strauss, S. H., Ebenau, R. G. (1994). Software inspection process, McGraw–Hill. Votta L. G., Jr. (1993). Does every inspection need a meeting? Proceedings of the ACM SIGSOFT 1993 Symposium on Foundations of Software Engineering, 18(5) of ACM Software Engineering Notes, 107-113, December

Ward, W.T. (1989). Software Defect Prevention Using McCabe's Complexity Metric. Hewlett-Packard Journal, 64-69

Zage, W.M., and Zage, D.M. (1993). Evaluating Design Metrics on Large-Scale Software. *IEEE Software* 75-80.

Zuse, H. (1990). *Software Complexity: Measures and Methods.* Walter de Gruyer & Co., New York, New York

VITA

Steven Christopher Oakes was born in Laurel, Maryland, on August 3, 1963, the son of Robert William Oakes and Marina Louise Oakes. He completed his work at Laurel High School, Laurel, Maryland, in 1981. He entered the University of Texas at Austin in August 1989. He received the degree of Bachelor of Business Administration in Finance in June 1991. A year after graduating from the University of Texas, he entered Texas State University-San Marcos. During the next several years, he was employed as a Nurse while taking Pre-medical and Computer Science courses. He received the degree of Bachelor of Science from Texas State in June 2001. He began to work for IBM upon graduating in 2001. During the next several years while employed at IBM, he entered the Master's program for Software Engineering. In August 2005, he received the degree of Master's of Science from Texas State University-San Marcos.

Permanent Address: 1521 Farm Dale

Allen, Texas 75002

This thesis was typed by Steven Christopher Oakes.