# AN EXAMINATION

# OF

# STABILITY AND REUSABILITY

# IN HIGHLY ITERATIVE SOFTWARE

by

**PATRICIA L. RODEN**

## A DISSERTATION

**Submitted in partial fulfillment of the requirements**
**for the degree of Doctor of Philosophy in**
**The Department of Computer Science**
**to**
**The School of Graduate Studies**
**of**
**The University of Alabama in Huntsville**

**HUNTSVILLE, ALABAMA**

**2008**

ProQuest LLC
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106-1346

In presenting this dissertation in partial fulfillment of the requirements for a doctoral degree from The University of Alabama in Huntsville, I agree that the Library of this University shall make it freely available for inspection. I further agree that permission for extensive copying for scholarly purposes may be granted by my advisor or, in his/her absence, by the Chair of the Department or the Dean of the School of Graduate Studies. It is also understood that due recognition shall be given to me and to The University of Alabama in Huntsville in any scholarly use which may be made of any material in this dissertation.

________________________________________ ____________________

(Student Signature) (Date)

## DISSERTATION APPROVAL FORM

Submitted by Patricia L. Roden in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science and accepted on behalf of the Faculty of the School of Graduate Studies by the dissertation committee.

We, the undersigned members of the Graduate Faculty of The University of Alabama in Huntsville, certify that we have advised and/or supervised the candidate on the work described in this dissertation. We further certify that we have reviewed the dissertation manuscript and approve it in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science.

________________________________________ Committee Chair
(Date)

________________________________________

________________________________________

________________________________________

________________________________________

________________________________________

________________________________________ Department Chair

________________________________________ College Dean

________________________________________ Graduate Dean

# ABSTRACT

The School of Graduate Studies

The University of Alabama in Huntsville

Degree Doctor of Philosophy College/Dept. Science/Computer Science

Name of Candidate Patricia L. Roden

Title An Examination of Stability and Reusability in Highly Iterative Software

In this dissertation, we examine the stability and reusability of agilely developed software. When considering stability, we wonder if the highly iterative nature of the agilely-developed software would adversely affect software stability. When considering reusability, from one standpoint we could suppose that the highly iterative nature of an agile process such as the extreme programming paradigm, with less time spent on formal design and the continuous emphasis on choosing the simplest approach to accomplish the task, would result in code which would be less reusable. This seemed particularly true since in the past it has been a truism that developing reusable software required additional work. On the other hand, because of the emphasis on refactoring in highly iterative processes, the resulting code should be more readable, simple, and perhaps therefore more reusable and stable.

First, we investigate the stability of software developed with an agile process utilizing existing stability metrics. The relationship of these stability metrics with the Total Quality Index (TQI) of the QMOOD Quality Model is also studied. Secondly, the well known Chidamber and Kemerer metrics are examined in an effort to develop a model to predict faults over the iterations of the agilely developed projects. Next, we investigate the relationship between faults, refactoring, and reusability in software developed using an agile process. Lastly, the expert reusability evaluations of software developed using a traditional plan-based method are compared to the reusability evaluations for the same applications developed using agile methods.

Our results show that some of the existing object-oriented metrics show potential for stability analysis. Our results also indicate TQI and stability might be used interchangeably in some situations. The intercorrelation of the C&K metrics over our data set made developing fault prediction models difficult and is similar to that experienced by some other researchers in the past. We determine that faults and refactoring are related; however, our results did not show a clear relationship between faults and reusability or refactoring and reusability. Lastly, our results indicate that software developed using a traditional plan-based method is more reusable than software developed using an agile method.

Abstract Approval: Committee Chair ______________________________________
(Date)

Department Chair ______________________________________

Graduate Dean ________________________________________

# ACKNOWLEDGMENTS

There are so many people who must be acknowledged for their contributions to this very long awaited completion of my degree. First, I thank my advisor, Dr. Letha Hughes Etzkorn, for her guidance, inspiration, attention and time devoted to this most unusual degree program. I find her talents amazing, her insights remarkable, and her gracious helpfulness inspiring. I also want to express my sincere gratitude to the members of my committee: Dr. Harry Delugach, Dr. Sampson Gholston, Dr. Wei Li, Dr. Pete Slater and Dr. Mary Ellen Weisskopf. The value of their time, effort, and feedback is immeasurable. I must also thank Dr. Shamsnaz Virani for her help in the collection and organization of the expert evaluations.

My deepest appreciation must be expressed to my family who have supported, encouraged and endured this adventure. To my husband Randy, I thank him for his tolerance of takeout dinners, less than spotless housekeeping and evenings of being abandoned while I traveled to meetings in Huntsville or worked on this dissertation. His patience, encouragement and love served as a strong foundation on which I could lean. To my daughter Miranda, I thank her for the inspiration to complete this degree. Our trips to Huntsville were a special time. We shared the joys and trials of completing our degrees together. I don’t know many mothers who can count their daughter as one of their colleagues as well as a best friend.

Most importantly, I must thank God for blessing me with the health to complete this degree. May He receive the praise for what He continues to do in my life.

# TABLE OF CONTENTS

Chapter

# LIST OF FIGURES

Figure Page

# LIST OF TABLES

# LIST OF ACRONYMS

| | |
|---|---|
| AM | Agile Modeling |
| ANA | Average Number of Ancestors |
| ASD | Adaptive Software Development |
| C&K | Chidamber and Kemerer Metrics |
| CAM | Cohesion Among Methods of Class |
| CBO | Coupling Between Objects |
| CIS | Class Interface Size |
| DAM | Data Access Metric |
| DCC | Direct Class Coupling |
| DIT | Depth of Inheritance Tree |
| DSC | Design Size in Classes |
| DSDM | Dynamic Systems Development Method |
| FDD | Feature-Driven Development |
| FSSEC | Fire Support Software Engineering Center |
| ISD | Internet Speed Development |
| ISO/IEC | International Organization for Standardization/International Electrotechnical Commission |
| LCOM | Lack of Cohesion in Methods |
| LOC | Lines of Code |
| MFA | Measure of Functional Abstraction |
| MOA | Measure of Aggregation |
| NIH | Not Invented Here |

| | |
|---|---|
| NOC | Number of Classes |
| NOH | Number of Hierarchies |
| NOM | Number of Methods |
| NOP | Number of Polymorphic Methods |
| QMOOD | Quality Model for Object-Oriented Design |
| RFC | Response for a Class |
| SDI | System Design Instability Metric |
| SDI$_e$ | System Design Instability Metric with Entropy |
| SPSS | Statistical Package for the Social Sciences |
| TFC | Total Function Calls |
| TQI | Total Quality Index |
| VIF | Variance Inflation Factor |
| WMC | Weighted Method Per Class |
| XP | Extreme Programming |

# CHAPTER 1

## INTRODUCTION

The introduction of agile software development methods to the software engineering landscape has led to questions and controversy concerning their usefulness and appropriateness, as well as the quality of the resulting projects. There are proponents who point to the traits of flexibility, embracing change and continuous testing as strengths [Nerur and Balijepally, 2007]. Others point to the lack of planning and required documentation as negative characteristics of the agile paradigm stating that it seems to be “nothing more than an attempt to legitimize hacker behavior” as compared to the traditional, or plan-based models [Rakitin, 2001]. A third group points out that both the agile and traditional methods have their strengths and weaknesses and for some applications one method may be more suited than the other [Boehm and Turner, 2004].

As the need for timely economically successful software development increases, management decisions become more critical. In order to aid software project managers in making an educated selection between the traditional plan-based method and an agile method for a particular application, research into the application of each method and the resulting product must be investigated. Advantages and disadvantages concerning the characteristics of time needed, quality of the product, reusability, and maintainability must be considered.  Barry Boehm stated that “Both agile and plan-driven methods have a home ground of project characteristics in which each clearly works best and where the other will have difficulties” [Boehm, 2002].

Four major reports on the characteristics of agile methods and summary of the research already completed have been presented [Abrahamsson et al., 2002; Cohen et al., 2004; Erickson et al., 2005; Dybå and Dingsøyr, 2008].  Many of the research studies, however, have limited their view to one or the other method without making comparisons [Kivi et al., 2000; Wood and Kleb, 2003; Müller and Tichy, 2001].  Other articles have been concerned with comparing the practices of the two methods with only anecdotal data [Huo et al., 2004; Williams, 2001].

Several researchers in the discipline have cited the need for significantly more empirical research in the area in order to provide advice and guidance for decision makers concerning which method would be best suited for their needs [Brilliant and Knight, 1999; Layman, 2004; Dingsøyr et al., 2008; Abrahamsson et al., 2003; Dybå and Dingsøyr, 2008].  Empirical research is defined by Brilliant and Knight as “analysis based on the observation of actual practice for the purpose of discovering the unknown or testing a hypothesis.”  Not only is it recommended that the number of empirical studies be increased, but also the quality of these studies be enhanced through improving research methodology [Dingsøyr et al., 2008].  Zannier et al. studied twenty-nine *International Conference on Software Engineering Proceedings* with regard to quantity and quality of empirical studies.  Their results demonstrated an increase in the number of empirical studies through the years but did not demonstrate an increase in the soundness of such studies [Zannier et al., 2006]. For industry to utilize the results of empirical studies, they must be conducted in a manner with high validity to provide a level of trust. The relevance of a study is expanded when it is repeated by other research groups on new populations [Sjøberg et al., 2007]. Increasing the interaction of academia and industry, allocating more resources for empirical research and developing a focus for the areas of empirical research are also needed [Sjøberg et al., 2007].

Research in the area of the agile methods is falling behind the practice of utilizing these methods [Ågerfalk and Fitzgerald, 2006]. The rate of increase in the application of agile methods has outdistanced the increase in agile research. Because the agile methods bring to the software engineering arena a new set of practices and topics such as pair programming, story cards, unit testing and refactoring, the areas for research are increased even more, resulting in a heightened need. Research is needed not only for the management who are making the decision to implement an agile project, but also for the development team to better understand the development process and "the complicated dynamics of agility" [Dingsøyr et al., 2008].

We spent the past several years introducing software engineering classes to the traditional plan-based life cycle models used in the development of software projects. In this context, we became very intrigued by the highly iterative approach used within agile methods such as extreme programming and how the resulting projects compare to the plan-based projects with regard to software qualities such as stability and reusability. When considering stability, we wondered if the very highly iterative nature of agilely-developed software would adversely affect software stability, relative to the stability of software developed using plan-based methods, which typically employs fewer or no iterations. This question has also been raised by others [A-PrimeSoftware, 2008]

When considering reusability, from one standpoint we could suppose that the highly iterative nature of an agile process such as the extreme programming paradigm, with less time spent on formal design and the continuous emphasis on choosing the simplest approach to accomplish the task, would result in code which would be less reusable. This seemed particularly true since in the past it has been a truism that developing reusable software required additional work. On the other hand, because of the emphasis on refactoring in highly iterative processes, the resulting code should be more readable, simple, and perhaps therefore more reusable and stable. In fact, some authors have claimed that developing reusable software within an agile paradigm is quite achievable [Heinecke et al., 2003]. Some authors go further and imply that the intrinsic characteristics of the agile paradigm tend to result in reusable software [Knoernschild, 2006]. In particular, some authors have argued that refactoring, which is an integral part of most agile software processes, particularly improves reusability of software [Moser et al., 2006].

In this dissertation, we investigate the stability of software developed using an agile process using existing stability metrics. As part of our stability analysis, we compare existing stability metrics to some existing object-oriented metrics over several iterations of agilely developed software. Our results show that some of these existing object-oriented metrics show potential for stability analysis. Our primary stability analyses are discussed in Chapter 4. Although stability is often defined in terms of degree of modification of the software [Alshayeb and Li, 2005], it is also sometimes defined in terms of number of faults [Boudnik, 2008; Repenci, 2008], although this might more accurately be termed software reliability. Since stability is sometimes defined in terms of faults, in Chapter 5, we also investigate the utility of the Chidamber and Kemerer (C&K) metrics suite [Chidamber and Kemerer, 1994], probably the best known object oriented software metrics suite, as fault predictors over our data set.

Additionally, in Chapter 6, we investigate the relationship between faults, refactoring, and reusability in software developed using an agile process. Again, since stability is sometimes defined in terms of faults, our analysis of faults versus refactoring represents an additional investigation of stability. Our comparison of faults and refactoring to reusability is part of our investigation of the reusability of agilely developed software. Finally, in Chapter 7, we also compare the reusability of software developed using an agile process to the reusability of software developed using a plan-based process.

The data we used in our empirical analyses was collected in senior software engineering courses taught by two different professors at two different universities across multiple semesters. Our overall research plan and methodology, including an in-depth description of the data analyzed, is provided in Chapter 3.

Background required to understand our work is in Chapter 2. Conclusions and Future Research are in Chapters 8 and 9, respectively.

# CHAPTER 2

# BACKGROUND

The inaugural use of the term “Agile Method” can be credited to the attendees at a meeting in Snowbird, Utah in February 2001. This meeting consisted of a group of seventeen leaders in a field which up to that time had been called “lightweight.” The outcomes of the meeting were a document called the “Agile Manifesto” and a group named the “Agile Alliance.”

The “Manifesto for Agile Software Development” [Agile Manifesto, 2008] states that

“We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

**Individuals and interactions** over processes and tools

**Working software** over comprehensive documentation

**Customer collaboration** over contract negotiation

**Responding to change** over following a plan.

That is, while there is value in the items on the right, we value the items on the left more” [Agile, 2008].

## 2.1 Agile Methods

The agile methods are recognized as possessing such traits as the use of short iterations, early and planned testing, pair programming and feedback loops. These characteristics seem to support the method name of agile which is defined as "moving quickly and easily, active, lively and nimble." According to Highsmith, "Agile methods stress two concepts: the unforgiving honesty of working code and the effectiveness of people working together with goodwill" [Highsmith, 2001]. The use of story cards, standup meetings, and the focus on teamwork are also seen in agile methodology. Unlike the traditional life cycle models which have been given the name of "plan-driven" by Barry Boehm [Boehm, 2002], agile methods welcome change at any time, even late in the development process, encourage a close association with the customer, continually look for ways to keep things simple, and provide a climate where developers can be productive, motivated, and feel a sense of trust in their ability to get the job completed [Paulk, 2002]. Highsmith states that "agile development defines a strategic capability, a capability to create and respond to change, a capability to balance flexibility and structure, a capability to draw creativity and innovation out of a development team, and a capability to lead organizations through turbulence and uncertainty" [Highsmith, 2002].

There are many methods for software development which are classified as agile. These include Adaptive Software Development (ASD), Dynamic Systems Development Method (DSDM), Feature-Driven Development (FDD), Internet-Speed Development (ISD), the Crystal family of methods, Agile Modeling (AM), Scrum, and Extreme Programming (XP). In the article "New Directions on Agile Methods: A Comparative

Analysis" the authors present an evolutionary map of these methods [Abrahamsson, 2004]. The Adaptive Software Development method was developed by Highsmith and Bayer as a method of developing large complex software systems [Abrahamsson, 2003]. Pressman describes the process as a three step life cycle consisting of speculation in which planning is performed and release cycles are defined, collaboration in which requirements are gathered, and learning using focus groups, formal technical reviews, and postmortems [Pressman, 2005]. ASD consists of six essential principles: a mission setting forth a target at which to aim, a set of features representing the value to the customer, iterations where change is viewed positively, set delivery times for each version, and the most difficult tasks are addressed first by embracing risk [Pfleeger and Atlee, 2006].

Dynamic Systems Development Method (DSDM) can be traced to a January 1994 meeting of sixteen rapid application developers in the United Kingdom [Abrahamsson, 2003]. It was published in January 1995 after its creation by a consortium of organizations which take the responsibility of being "keepers" of the method [Pressman, 2005; Larman and Basili, 2003]. DSDM follows a eighty percent rule which states that eighty percent completion of an iteration is enough to move on to the next iteration. The method consists of two activities: feasibility study and business study followed by three iterative cycles: functional model iteration, design-and-build iteration and implementation [Pressman, 2005]. DSDM has been widely adopted in Europe and especially the UK [Beynon-Davies and Williams, 2003].

Feature-Driven Development (FDD) was developed by Jeff De Luca and Peter Coad while working on a large commercial lending application for a Singapore bank in

1997 [Abrahamsson, 2003]. The method consists of five basic steps or activities: Develop an Overall Model, Build a Features List, Plan by Feature, Design by Feature, and Build by Feature. FDD also makes use of six milestones per feature: "domain walkthrough, design, design inspection, code, code inspection, and promote to build" [Pressman, 2005]. Internet-Speed Development (ISD) is a framework which is characterized by quality dependencies, process adjustments, and time drives [Baskerville et al., 2001]. ISD is useful in development situations when a quick release time is critical [Abrahamsson, 2003]. The Crystal family is a collection of methods created by Alistair Cockburn and Jim Highsmith based on the concept that each project needs the ability to maneuver using the most appropriate crystal family method [Pressman, 2005; Abrahamsson, 2003]. The belief that the major influence on the quality of a project is the people involved and their communication through short delivery cycles is another foundation of the Crystal family of methods [Pfleeger and Atlee, 2006].

Agile Modeling (AM) is an approach to modeling which encourages the support of documentation and design needs by creating advanced models. AM is to be used in conjunction with another agile process method [Ambler, 2002]. Agile Modeling is described using a set of core principles which are extremely similar to those of Extreme Programming. There are also a set of supplementary principles as well [Pressman, 2005]. These principles result in a set of core and supplementary practices [Ambler, 2002].

Scrum is an agile method developed by Jeff Sutherland and named after a term used in rugby referring to a group of eight players formed around a ball to move it down the field [Rising and Janoff, 2000]. Scrum was created in 1994 at Object Technology and Schwaber and Beedle are credited with commercializing it [Pressman, 2005]. Scrum uses iterative development making use of “sprints,” one to four week iterations, which deliver the project incrementally following the initial planning. A “backlog” consisting of a list of identified requirements is used to direct the project team’s activity [Rising and Janoff, 2000]. Short, daily fifteen to thirty minute meetings which are called “scrums” are led by a “scrum master” [Pfleeger and Atlee, 2006]. Three questions are asked of each team member at the meetings:

1. What have you completed, relative to the backlog, since the last Scrum meeting?
2. What obstacles got in your way of completing this work?
3. What specific things do you plan to accomplish, relative to the backlog, between now and the next Scrum meeting? [Rising and Janoff, 2000].

Scrum is beneficial in situations where requirements are not easily delineated at the beginning of the project and where there is an expectation of chaos during development [Rising and Janoff, 2000]. In fact, according to Pressman, the most often cited website related to Scrum is www.controlchaos.com [Pressman, 2005].

## 2.2 Extreme Programming

One very well known agile method is Extreme Programming [Beck, 2000]. In a study Dybå and Dingsøyr conducted concerning articles published on agile methods through the year 2005, seventy-six percent were dedicated to Extreme Programming [Dybå and Dingsøyr, 2008]. Extreme Programming (XP) is described as having four values: Communication, Simplicity, Feedback, and Courage and four basic activities: Listening, Designing, Coding, and Testing. The four values give rise to fifteen principles: Rapid Feedback, Assuming Simplicity, Incremental Change, Embracing Change, Quality Work, Teaching Learning, Small Initial Investments, Playing to Win, Concrete Experimentation, Open, Honest Communication, Working with People's Instincts, Accepted Responsibility, Local Adaptation, Traveling Light, and Honest Measurement. The four basic activities, sometimes called the technical backbone of the process, result in twelve recognized practices: the Planning Game, Small Releases, Metaphor, Simple Design, Testing, Refactoring, Pair Programming, Collective Ownership, Continuous Integration, Coding Standards, On-site Customer, and the 40-hr Week [Hislop et al., 2002].

Although Extreme Programming has been on the software development scene for several years and much has been written on the subject, most of the literature concerning XP is lacking in concrete scientific analysis of the evidence of how well it works. One attempt to seek rationale for the Extreme Programming practices was conducted by Kähkönen and Abrahamsson using the 5-A model for knowledge creation, which is a theoretical framework proposed by Nonaka and Takeuchi. Kähkönen and Abrahamsson concluded that the Extreme Programming practices enhanced knowledge creation through "immediate (or frequent) and mutual articulation and appropriation. The practices anticipate only in short intervals and accumulate tacit knowledge in socially shared meaning structures rather than explicit knowledge in documents."[Kähkönen and Abrahamsson, 2003] Kuppuswami reported on the development of a simulation to model the effect of the twelve extreme programming practices which demonstrated a reduction in cost for a software development effort [Kuppuswami et al., 2003].

### 2.2.1 Communication

Communication is not just encouraged in multiple areas of Extreme Programming but is required.  This encompasses the face-to-face interaction between the customer and the programming team, the communication with management, the communication between programming pairs and within each pair [Williams, 2003]. The method of communication differs as well.  First, communication is immediate in informal sessions or in daily stand-up meetings to give a brief progress report. Secondly, instead of the customer describing requirements and the team preparing a lengthy specification document, the Extreme Programming team produces a series of user story cards which are prioritized and with the customer's recommendation implemented in a series of iterations.  Continuous input from the on-site customer will be discussed later.  The lack of a formal specification document leads to one of the major criticisms of the Extreme Programming methodology which is the lack of extensive documentation.

#### 2.2.1.1 Pair Programming

Another practice of the Extreme Programming process is the use of pair programming, two programmers working side by side at one computer in order to accomplish a task.  The old adage "Two heads are better than one" has been supported with the research into the quality of the results generated by pair programming alternatively called "collaborative programming." Although it might first be thought that since two programmers are working at only one computer, only half as much work as if the two were working separately will be accomplished , that is not in fact the case. Some studies give evidence that the two programmers working as a pair produce more code than two solo programmers and the pair programmers produce "better" code [Williams and Kessler, May 2000]. The qualities of the code which would make it "better" include more efficient, tighter, possessing less defects, and having more simplicity [Williams et al., 2000].

Other benefits of pair programming have been documented by Williams and Upchurch [Williams and Upchurch, 2001]. The economics of pair programming is described as being enhanced by savings in code development time, quality assurance, and field support costs. The satisfaction in the working environment was also pointed to as a benefit of pair programming. The process of continually reviewing their code, learning from their pair partner, placing positive pressure on their partner to be productive, focused on the task at hand, and punctual, and the development of communication skills also make pair programming attractive.

Many reasons for the documented success have been proposed. First, while one programmer is "driving" at the keyboard, the other programmer is free to view the project from a more strategic viewpoint, locating defects earlier. The pairs continuously review and refactor their work, trying to simplify it in order to reach the best solution. They bring together two set of eyes to detect errors and two sets of ideas in order to reach an effective, efficient implementation [Williams and Kessler, December 2000].

The practice of pair programming certainly supports the extreme programming value of communication. Beck also points out that some of the other extreme practices would not work without pair programming. Without the use of pair programming, testing might be ignored, refactoring might be delayed and integration might be avoided until too late [Beck, 2000].

#### 2.2.1.2 On-site Customer

The availability of an on-site customer to answer questions, help develop tests, help set priorities, develop user stories, and settle disputes is cited as invaluable to the extreme programming project. The on-site customer has the final say concerning issues such as what must be done (which are placed on user story cards), what order to implement the user stories, and what level of quality is required. This person should be someone who will be intimately involved in the use of the product when it is completed. Although many companies hesitate to release one of their employees to be the on-site customer, after successful delivery of the first few iterations the attitude changes. “The project is *steered* to success by the customer and programmers working in concert”[Jeffries et al., 2001].

### 2.2.2 Simplicity

“Simplicity – the art of maximizing the amount of work not done – is essential” [Principles behind the Agile Manifesto, 2008]. This statement represents one of the core values of extreme programming. Programmers are encouraged to leave the product in the simplest form possible and to do the simplest thing that will work with the smallest practical number of classes and methods. Any duplicate logic or extra complexity is removed as soon as it is detected. This process is called refactoring (and will be further discussed later). This attitude is in opposition to the traditional approach to build in as much flexibility as possible in an effort to be prepared for what tomorrow might bring. The concept of simplicity applies to all areas of the product: communication, design, code and testing.

#### 2.2.2.1 Simple Design

According to Beck, the strategy for design in Extreme Programming is to have the simplest design that runs the designated tests, which are always constructed first. Beck goes on to define what is meant by "simplest." A simplest design results in a system which communicates what is needed, has no duplicate code (sometimes called the Once and Only Once rule), has the fewest possible classes, and the fewest possible methods [Beck, 1999]. A simple design leads to easier communication and because of the short iterations, feedback allows for quick verification of the design.

#### 2.2.2.2 Refactoring

Refactoring is the disciplined process of software evolution through techniques which reduce software complexity, improve software quality, and improve internal structure while leaving the behavior of the code intact. This term was originally introduced by Opdyke to represent restructuring as defined by Chikofsky and Cross in "Reverse Engineering and Design Recovery: A Taxonomy" to be "the transformation from one representation form to another at the same relative abstraction level, while preserving the subject system's external behavior" [Chikosfsky and Cross, 1990].

The preservation of behavior could refer to preserving the resulting output values corresponding to given input values, preserving temporal constraints such as the execution time of certain operations or sequences, or preserving the power consumption and memory constraints for embedded software [Mens and Tourwé, 2004]. The use of preconditions, invariants and postconditions often expressed in predicate calculus is one method of representing this preservation. The presence of documented tests prepared to demonstrate behavior is essential.

The study of refactoring has resulted in classification of refactoring techniques into primitive refactorings which are elementary building blocks which can be used to create composite refactorings. Examples of primitive refactorings include renaming a variable, class, or method, moving a method up to super-class, adding a super-class, moving a method from a super-class down to a sub-class, extracting a method from another method, and replacing a conditional (if or switch) with a polymorphism.

Refactoring is often performed in response to something called "bad smells," undesirable characteristics of code [Fowler, 1999]. For example, one "bad smell" might be a method which is too long. The greater the length of the method, the more difficult it is to understand its purpose. A refactoring might extract another method from the longer method. A long parameter list might be simplified by replacing it with a method in order to pass in enough so that the original method can get to everything it needs. Duplicated code might be removed by extracting a method, pulling up a field, forming a template method or extracting a class. Inconsistent or uncommunicative names may be replaced using the rename refactoring technique. Dead code which includes variables, parameters, methods, code fragments, and classes can be removed by the Delete Code primitive refactoring technique [Fowler, 1999].

Refactoring can be considered as a continual process of quality improvement. Quality attributes which may be considered when refactoring include reusability, robustness, extensibility and performance. Other benefits can improve ease in future modification, ease in spotting a bug when debugging, more comprehensible code, and less error prone code. The benefits of refactoring are not always immediately realized in the short term, but more often seen in the long term.

There are two methods for performing refactoring: manual and automated. Manual refactoring is performed by an individual or pair of programmers and requires the need to rebuild and run tests frequently. If the system containing the refactored code can be rebuilt quickly, and there exist automated regression tests, then manual refactoring may be feasible. If not, automated refactoring might be considered if there exist tools appropriate to the language or the type of code. Automated refactoring is generally quicker, easier, and less tedious, and reduces the chance of bugs being introduced into the project. Automated tools may be integrated into the IDE selected by the development team which will make refactoring even easier and will tend to make the developers view refactoring as just another facet of their development process.

The area of refactoring has been studied from several different viewpoints. One approach has been to consider only one category of refactoring. Another strategy is to study refactoring dedicated to only one language. Early studies of this type involved Smalltalk. Later studies have attempted to develop language independent refactoring techniques [Tichelaar et al., 2000]. Several studies involve developing automated support for refactoring [Kataoka et al., 2001; Maruyama and Shima, 1999; Jeon and Bai, 2002]. Yet other studies attempt to use graph theory to support the process of refactoring [Maruyama and Shima, 1999]. Refactoring using design patterns has also been studied [France et al., 2003]. Grammar based refactoring has also been reported [Kosar et al., 2004]. Still others look at refactoring as applied to different categories of problems like embedded systems, real-time systems, and database management systems [Mens and Tourwé, 2004].

### 2.2.3 Feedback

As one of the common threads which bind the agile methods together as a group, feedback is one of the fifteen underlying principles of Extreme Programming. The small releases, continuous testing and integration, pair programming, and on-site customer are all practices which contribute to this principle. Kähkönen and Abrahamsson concluded in their study of knowledge management theories as applied to extreme programming that several of the XP practices serve to encourage a greater understanding of the project between the customer and developer and within the development team [Kähkönen and Abrahamsson, 2003]. The principle of rapid feedback is substantially supported by letting participants talk to each other instead of the more traditional document driven approach. The physical arrangements of pair programmers and on-site customers also encourage feedback. Scott points out that “without proper control of the feedback this communication could destabilize the team and halt its release of software.” He further points out that it is the combination of all twelve Extreme Programming practices that make sure that the team stays on target and reaches its goal [Scott, 2003].

#### 2.2.3.1 Testing

The emphasis on testing as one of the twelve practices of Extreme Programming results in its description as “test-driven.” It is described as being an incremental design approach which is motivated by passing tests. The cycle of writing a unit test, writing the code, performing the test, and refactoring the code is to occur with great frequency in an Extreme Programming project. Testing actually occurs in the form of both unit tests and functional tests. Unit testing is testing performed on an individual programming component such as a class. Unit tests which are created by the developers and permanently saved for daily regression testing must be passed with a grade of 100% before any more work or functional testing occurs. In fact, Beck states that “If there is a technique at the heart of XP, it is unit testing” [Beck, 1999]. Automated tools exist which support the creation and maintenance of unit tests. Because testing goes on several times a day, these automated tools are almost a necessity. On the other hand, functional testing is concerned with the testing of a group of classes that implements a performance whose description is provided by the user or customer [Smith and Stoecklin, 2001]. These functional tests which correspond to a user story serve in the final validation of the project. The successful passing of functional tests can serve to demonstrate the progress on the project and can build confidence in the work accomplished.

#### 2.2.3.2 Continuous Integration

The practice of integrating as soon as a unit has been tested results in reducing the number and size of conflicts which occur when code is integrated. A programming pair would integrate at least once a day into the code base. This also serves to reduce the cost of integration as supported by an integrated development environment and to provide a repository to aid each programming pair to maintain the direction of their work. It also provides almost immediate feedback to the developers. As with the results of testing in Extreme Programming, the practice of continuous integration can subsequently produce confidence in the project. At all times there is a working subset of the project along with a suite of tests which demonstrate the continued correctness of the subset.

## 2.3 Software Quality

The definition of software quality is elusive, multidimensional, and controversial. Of course, if we are to attempt to measure software quality directly or indirectly, a good definition is necessary. Kitchenham and Pfleeger reported a study they conducted in 1995 in which they solicited the definition of software quality [Kitchenham and Pfleeger, 1996]. The responses were classified according to the five different perspectives of quality set forth by David Garvin:

- the transcendental view which sees quality as being recognizable but unable to be defined,
- the user view which sees quality as the fitness for purpose,
- the manufacturing view which sees quality in the specification conformance,

- the product view which sees quality associated with the product characteristics,
- the value-based view which sees quality relative to the amount of money a customer is willing to pay [Gavin, 1984].

The ISO/IEC 9126 software quality standard cites the six attributes of functionality, reliability, usability, efficiency, portability and maintainability relating to software quality. This standard further lists the attributes of analyzability, testability, changeability, and stability as contributing to maintainability [ISO/IEC 9126, 1991].

### 2.3.1 Stability

Elish and Rine point out that "stability is one of the most desirable features of any software design" and state that when a design is not stable the reliability will likely suffer and the maintenance will be more costly [Elish and Rine, 2003]. Stability is touted as a quality factor indicating maturity [Mattsson and Bosch, 1999]. One encounters a variety of definitions of stability in the literature. It is sometimes defined as the ease of software evolution while preserving the software design [Grosser et al., 2003; Elish and Rine, 2005]. Stability has also been described as a characteristic of software in that it can remain architecturally intact through evolutionary changes. Stability may be considered using structural, behavioral, or economical changes. However, structural changes are most often used due to the ease of automation [Tonu et al., 2006]. Software stability may be considered to be an indicator within a software project's life where changes to one class will not likely spread or ripple to other classes in the design. Logical and performance stability are two categories of stability. Logical stability is concerned with the structure of the design while performance stability is related to the behavior of the design [Elish and Rine, 2003].

Stability is often defined in terms of degree of modification of the software [Alshayeb and Li, 2005]. However, other authors have defined stability in terms of faults (though this could more properly be termed reliability rather than stability). For example, Repenci defined software stability ratings for software releases based largely on level and kind of faults [Repenci, 2008].

Bansiya used the extent-of-change metric to measure the stability of a framework structure and applied his work to a study of the Microsoft Foundation Classes (MFC) and Borland Object Windows (OWL) application frameworks [Banisya, 2000]. Grosser et al. made use of case-based reasoning and similarity based approach to predict stability. Case-based reasoning had already been used for modeling correctability [Almeida et al., 1999], reusability [Esteva and Reynolds, 1991] and reliability [Ganesan et al., 2000]. Elish and Rine used the Chidamber and Kemerer (C&K) metrics which will be discussed later in order to determine whether there is a correlation with performance stability [Elish and Rine, 2003].

### 2.3.2 Reusability

The growing cost and demand for software development has resulted in a problem labeled for years as the software crisis. In an effort to address this issue and reduce time and cost, several concepts have been put forth as possible solutions. The development of programming languages like Ada, commissioned by the United States Department of Defense, was an effort to curb programming costs by encouraging development of libraries of reusable packages. Other studies involving software reuse, the automatic detection of reusable code, the intentional creation of reusable code, and the storage and retrieval of reusable code have also been conducted [Dandashi, 2002; Etzkorn and Davis, 1997; Kim, 1997; Kim and Stohr, 1992; Mambella et al.,1995].

Reusability has been touted as a possible "silver bullet" to solve the software crisis although others have stated that there is no silver bullet [Brooks, 1987; Fraser and Manci, 2008]. Like other software qualities it is difficult to quantify and measure. Software Reuse may be defined as "utilization of a software component C within product P, where the original motivation for constructing C was other than for use in P" [Schach and Yang, 1995]. It is viewed as a "means to improve the process of software development and also the quality of the software produced" [Kim and Stohr, 1992]. The "reusability" of a software component is defined "as the extent to which a software component can be used with or without adaptation in a problem solution other than the one for which it was originally developed." Furthermore, "software reuse is the goal while software reusability is necessary in order to achieve this goal" [Kalagiakos, 2003].

Nazareth points out that along with reduced cost and time, other benefits such as improved software quality, greater knowledge sharing, improved maintainability, adoption of standards, and increased productivity of developers can be the results of reuse [Nazareth, 2004]. In an early study of the benefits of reuse, the Fire Support Software Engineering Center FSSEC reported that the benefits of reuse included

14%-68% increase in productivity

20% reduction in customer complaints

25% reduced time to repair

25% reduced schedule

50% reduction in integration time

20%-35% improvement in quality

20% reduction in training costs

400% return on investment [Smith and Sodhi, 1994].

Learning what causes software to be more reusable can enhance our ability to intentionally create software which is reusable. Characteristics which enhance a software component’s reusability include how well the software conforms to standards or style guidelines, the level of supporting information and the degree of testing performed [Poulin, 1994]. Yet other traits enhancing reusability include understandability, portability, generality, and retrievability. Ultimately, software is considered reusable if the effort to reuse the software component is considerably less than the effort to construct a similar functioning component [Prieto-Díaz, 1993].

Kalagiakos addresses several non-technical issues which play a role in the reuse of software. These issues include human factors, namely, the most widely recognized obstacle to reuse called the NIH “Not Invented Here “ syndrome and the change to the roles of people within the organization adopting the practice of reuse, contractual issues such as “Cost-Plus” contracts which actually encourage redevelopment over reuse, and certain legal issues with regard to the software rights [Kalagiakos, 2003].

Reuse may be classified from a number of perspectives. One method involves the actions which led to the reuse of the software. Namely, was it intentionally constructed and planned to be reused or was the reuse an unplanned fortunate accident. Another classification results from the types of applications in which the reuse occurs. Vertical reuse occurs when software is reused within the same area of application or domain. Horizontal reuse occurs when software is reused across different applications. A third method of categorizing reuse has to do with whether the code is reused and unchanged called black-box or "as is" reuse or white-box reuse which involves the changing of code during reuse [Prieto-Díaz, 1993]. Black-box reuse is preferable since it maximizes productivity. Reuse may also be classified as informal or formal. Informal reuse may be management focused occurring from a perceived immediate opportunity or programmer derived reuse of code from his/her personal library. "Formal reuse is a process driven activity that requires common standards, procedures, and practices applied consistently and universally across a given domain" [Smith and Sodhi, 1994]. This definition of formal reuse seems almost to be a hybrid of vertical and planned reuse.

No matter what the classification, most sources agree with the opinion that reuse is a desirable activity in an effort to reduce development cost and create more reliable software. The advent of object-oriented programming as opposed to procedural or imperative programming has been touted as producing reusable code since its beginnings in the 1980's [Meyer, 1996]. Bertrand Meyer stated object technology produces the appropriate level of abstraction in order to enhance reusability. He further states that "reusability in software is inseparable from adaptability" [Meyer, 1996]. The use of instances of a class, the inheritance in object-oriented code which allows a derived class to inherit from its parent class and the use of parametric polymorphism are all forms of reuse. Although object-oriented code possesses these traits which could lend it to reuse, there is an accumulation of object-oriented legacy systems designed without thought of reuse [Etzkorn and Davis, 1997].

There are many research areas involved in the study of reuse and reusability. One area of research involves studying methods for creating reusable code or code with a high potential for reuse. This process is sometimes referred to as “develop for reuse” [Mambella et al., 1995]. Another area of study is concerned with methods for recognizing reusable code. Two issues which need to be addressed when attempting to recognize reusable code have to do with whether the code is useful to the new system being constructed and whether the quality of the code satisfies the requirements of this new system [Etzkorn and Davis, 1997]. Studies in this arena make use of classification of code through such things as comments and identifiers used. A third area of research surrounds the process of developing systems through reusing code called “develop with reuse” [Mambella et al., 1995]. In order to support this development with reuse, there has been a good amount of research in developing repositories of reusable code which in turn leads to study in developing methods of classifying and retrieving code to be reused from those repositories. Examples of methods for classifying and retrieving code include the keyword approach used by Jones and Prieto-Díaz, the full-text approach described by Frakes and Nejmeh and Maarek et al., the faceted approach presented by Prieto-Díaz, the pattern matching through analogy approach used by

Maiden and Sutcliffe, and the use of a semantic encoding method proposed by Kim [Kim, 1997].

## 2.4 Software Metrics

The term “software metrics” has long been associated with the production of numerical results to represent a quality of software [Coppick and Cheatham, 1992]. In fact, the first use of the Lines of Code (LOC) metric can be seen as far back as the 1960’s [Fenton and Neil, 2000]. There is, however, a controversy as to whether these values should be called metrics or measures and several documents present a discussion on this topic [Pressman, 2005; Fenton and Neil, 1999]. In mathematics the term metric usually refers to the concept of distance. Other areas of controversy arise from what should be measured, how it should be measured, and what scale should be used to express the measurement. Several articles have been written to express the opinion that measurement theory should be applied to the area of metrics [Zuse, 1996; Poels and Dedene, 2000; Garcia et al., 2005; Hintz and Montazeri, 1996; Fenton, 1994]. The use of metrics can be applied to all phases in the development process. They can be computed in an effort to measure attributes such as maintainability, understandability, modifiability, complexity, usability, testability, reliability, reusability, and many more [Zuse, 1989]. The computation of these numerical results is performed in an effort to enhance the development process, aid the developer to understand their progress, guide management concerning the project health and schedule and ultimately lead to a higher quality project for the customer [Pfleeger et al., 1997]. Metrics can be calculated as a point value at a given time or can represent a cumulative value over a period of time [Schroeder, 1999].

Software metrics may be categorized according to how they are used. Process metrics, measures of properties of the software development process, are used for planning and support of management activities [Meyer, 1998] [Clark, 2002]. They help predict the status of a project for management and help estimate the effort which will be required. They further help management determine whether a project is on schedule. Product metrics are used for guidelines for improvement or comparison between existing systems [Schroeder, 1999]. They can be used for "addressing risks and problems earlier" [Clark, 2002]. They can also be used to predict faults [Tang, 1999]. Product metrics may be further categorized as external or internal. External product metrics measure qualities which are detectable by users of the software product. Internal product metrics have to do with attributes which are only detectable by the development team [Meyer, 1998]. Schroeder states "You use both product and process metrics, generally in conjunction, to assess the project as a whole" [Schroeder, 1999].

### 2.4.1 Object-Oriented Metrics

A multitude of software metrics have been proposed for object-oriented software. According to Schroeder these metrics can be placed into four categories: System Size, Class or Method Size, Coupling and Inheritance, and Class or Method Internals [Schroeder, 1999]. Examples of system size metrics are Lines of Code (LOC), Total Function Calls (TFC), and Number of Classes (NOC). Class or Method Size metrics include LOC per class or method, number of methods per class, and number of attributes per class. Coupling and Inheritance metrics demonstrate the degree of interdependencies between objects which will have effect on the ability of reuse of the objects. Examples include Class Fan-in, the number of classes depending on a given object, Class Fan-out, the number of classes upon which an object depends, Class Inheritance Level, and Number of Children per Class. The Class and Method Internals metrics support the measurement of quality at the class and method level. These include Number of Global References, Method Complexity, Number of Public Attributes per class, Percent of Commented Methods, and Number of Parameters per method [Schroeder, 1999].

#### 2.4.1.1 Chidamber and Kemerer Metrics Suite (C& K)

One of the first, most analyzed and most referenced suites of object-oriented metrics is the group of metrics proposed by Chidamber and Kemerer. These metrics were proposed with firm theoretical foundations and developed to be useful to software development organizations. The suite consists of six metrics:

- Weighted Methods per Class (WMC). The WMC metric measures the complexity of a class by considering the sum of the complexity of each of the methods. The larger the WMC number indicates a class with high complexity which will be less likely to be reused. Chidamber and Kemerer did not explicitly define complexity but stated it was left as an implementation decision. The complexity metric selected should have an interval scale.
- Depth of Inheritance Tree of a class (DIT). The DIT of a class is the maximum length of the path from the root of the inheritance tree to the given class. A high

DIT value indicates a greater complexity but a greater likelihood that inherited methods will be reused. A greater number of ancestors for a class make its behavior prediction more difficult.

- Number of Children (NOC). The NOC is the number of subclasses directly below a class in the inheritance tree. Moderate values for NOC point to the possibility of reuse through inheritance. An NOC value which is very large might indicate that the parent abstraction is inaccurate or misused. The number of children of a class could be used to indicate its importance to the design.
- Coupling Between Classes (CBO), The CBO measures how much independence a class possesses and hence how likely the class could be reused. A large CBO value would indicate that maintenance might be more difficult due to sensitivity to modifications in other areas and indicate that testing would be more critical.
- Response for a Class (RFC). The RFC indicates the number of methods that can be executed when responding to a message to an object of the class. A large RFC value points once again to an increased complexity of the class which makes for more complicated debugging and testing.
- Lack of Cohesion in Methods (LCOM). The LCOM metric is calculated by subtracting the count of the method pairs having similarities from the pairs of methods having no similarities. The more similarities within pairs of class methods would indicate a more cohesive class. A larger value would indicate a possibility of errors occurring during development [Chidamber and Kemerer, 1994].

### 2.4.1.2 The QMOOD Quality Model

Bansiya and Davis presented a hierarchical model for assessing the quality of object-oriented design [Bansiya and Davis, 2002]. It consists of six quality factors described in Table 2.1.

**Table 2.1: QMOOD Quality Factor Definition [Bansiya and Davis 2002]**

| Quality Factor | Definition |
|---|---|
| **Reusability** | Reflects the presence of object oriented design characteristics that allow a design to be reapplied to a new problem without significant effort. |
| **Flexibility** | Characteristics that allow the incorporation of changes in a design. The ability of a design to be adapted to provide functionality related capabilities. |
| **Understand-ability** | The properties of a design that enable it to be easily learned and comprehended. This directly relates to the complexity of design structure. |
| **Functionality** | The responsibility assigned to the classes of a design, which are made available by classes through their public interfaces. |
| **Extendibility** | Refers to the presence and usage of properties in an existing design that allow for the incorporation of new requirements in the design. |
| **Effectiveness** | This refers to a design's ability to achieve the desired functionality and behavior using object oriented design concepts and techniques. |

The six factors which are described in Table 2.1 are not able to be directly measured. This required Bansiya and Davis to define design properties which could be used to calculate these six quality factors but which could be measured directly. The eleven design properties are presented in following Table 2.2.

**Table 2.2: QMOOD Design Properties [Bansiya and Davis 2002]**

| Design Property | Definition |
|---|---|
| **Design Size (DSC)** | A measure of number of classes used in the design. |
| **Hierarchies (NOH)** | Hierarchies are used to represent different generalization–specialization aspects of the design. Classes in a design which have one or more descendants exhibit this property. |
| **Abstraction (ANA)** | A measure of generalization–specialization aspect of design. Classes in a design which have one or more descendents exhibit this property of abstraction. |
| **Encapsulation (DAM)** | Defined as the enclosing of data and behavior within a single construct. In object oriented designs the property specifically refers to designing classes that prevent access to attribute declarations by defining them to be private, thus protecting the internal representation of the objects. |
| **Coupling (DCC)** | Defines the interdependency of an object with other objects in a design. It is the measure of the number of other objects that would be accessed by an object in order for that object to function correctly. |
| **Cohesion (CAM)** | Accesses the relatedness of methods and attributes in a class. Strong overlap in method parameters and attribute types is an indication of strong cohesion. |
| **Composition (MOA)** | Measures the "part-of," "has," "consists –of," or "part-whole" relationships, which are aggregation relationships in object oriented design. |
| **Inheritance (MFA)** | A measure of the "is-a" relationship between classes. This relationship is related to a level of nesting of classes in an inheritance hierarchy. |
| **Polymorphism (NOP)** | The ability to substitute objects whose interfaces match for one another at runtime. It is a measure of services that are dynamically determined at run-time in an object. |
| **Messaging (CIS)** | A count of the number of public methods that are available as services to other classes. This is the measure of the services that a class provides. |
| **Complexity (NOM)** | A measure of the degree of difficulty in understanding and comprehending the internal and external structure of classes and their relationships. |

Bansiya and Davis related the eleven design properties in Table 2.2 above to the six quality factors in Table 2.1 using the following formulas in the following Table 2.3.

**Table 2.3: QMOOD Quality Factors and Design Properties Relationships [Bansiya and Davis 2002]**

| Quality Factor | Relationship |
|---|---|
| **Reusability** | -0.25 * Coupling +0.25 * Cohesion +0.5 * Messaging +0.5 * Design Size |
| **Flexibility** | 0.25 * Encapsulation -0.25 * Coupling +0.5 * Composition +0.5 * Polymorphism |
| **Understandability** | -0.33 * Abstraction +0.33 * Encapsulation - .33 * Coupling +0.33 * Cohesion -0.33 * Polymorphism –0.33 * Complexity –0.33 * Design Size |
| **Functionality** | 0.12 * Cohesion +0.22 * Polymorphism +0.22 * Messaging + 0.22 * Design Size +0.22*Hierarchies |
| **Extendibility** | 0.5 * Abstraction –0.5 * Coupling +0.5 * Inheritance +0.5 * Polymorphism |
| **Effectiveness** | 0.2* Abstraction +0.2 * Encapsulation +0.2 * Composition +0.2 * Inheritance +0.2 * Polymorphism |

The six quality factors were then used to find the Total Quality Index (TQI) by taking the sum of reusability, flexibility, understandability, functionality, extendibility, and effectiveness.

### 2.4.2 Stability Metrics

Assessing the stability of software may be accomplished by considering successive versions or iterations and determining the change in certain recognized metrics, such as LOC and the C&K metrics. Bansiya proposed an "extent of change" metric for analyzing structural stability. He outlined four steps in order to accomplish the structural stability assessment:

1. Identify the structural characteristics.
2. Define or select metrics which assess each of the structural characteristics.

3. Collect metrics data for the characteristics from several releases or versions.

4. Analyze the changed characteristics and compute the extent-of-change metric. The extent-of-change metric is computed by first normalizing the first metric's value to one and by dividing each other version's metric value by the previous version's metric value. The normalized metrics are then added to form an "aggregate change." The extent of change is then computed by taking the difference of the aggregate change and the first version's aggregate change [Bansiya, 2000].

The determination of what metric to use (product-related or process-related) has been the subject of several studies. Elish and Rine studied thirteen successive releases of the Apache Ant application. They made use of a suite of stability metrics stating that stability of object-oriented designs is two-dimensional, pointing to the dimensions of size and time. The size-based metrics were organized into class-based metrics which measured the number of stable, added, deleted, and modified classes and relationship-based metrics which examine the change or stability within four different relationships. The generalization relationship deals with the relationship between a superclass and its subclass. The relationship which occurs when one class is a part of another is called an aggregation relationship. The dependency relationship between two classes is seen when one class uses the other as a parameter type or a return type of a method. The final relationship is called the association relationship, which is created when one class uses a method from another class. For each of the four relationships, Elish and Rine propose counting the number of added relationships, the number of deleted relationships and the number of relationships which were stable. They have a straightforward naming convention where N represents Number, S represents Stable, A represents

Added, D represents Deleted, and the relationships are named GR, AR, DR, and SR. So the acronym NAGR represents the number of added generalization relationships. The time-based metrics are used to measure the length of time that a design structure remains stable. The units of days or months are normally used [Elish and Rine, 2005].

### 2.4.2.1 SDI Metric

The System Design Instability Metric (SDI) was proposed by Li et al. in 2000 as a measure of object-oriented software's evolution by measuring the percentages of classes which are affected by change from one iteration to the next. [Li et al., 2000]. The SDI metric makes use of the sum of three values to analyze the system-level design changes from one iteration to the next: the percentage of classes whose names change from one iteration to the next, the percentage of classes which were added, and the percentage of classes which were deleted. The use of the C&K DIT (Depth of Inheritance Tree) along with the number of separate hierarchies was recommended for a large design analysis as well.

Li et al. studied the effectiveness of the SDI metric for indicating the progress of a project and demonstrated that the SDI metric measures different aspects of the development of a project than do the C & K metrics [Li et al., 2000]. They suggested further study on different environments to verify their results.

Since the SDI metric is calculated on information available at design time, it can be considered a design metric.

#### 2.4.2.2 $\text{SDI}_e$ Metric

The $\text{SDI}_e$ metric was proposed by Olague et al. as a revision of the SDI metric, making use of the concept of entropy which was added due to the fact that the dynamic feature of agile development could obscure stability analysis [Olague et al., 2006]. The $\text{SDI}_e$ metric is simpler to calculate than SDI due to the fact that it can be automated whereas the SDI metric requires manual investigation for the count of name changes [Roden et al., 2007]. The original SDI metric also has a hypersensitivity to large single category changes which can be dampened by the use of entropy. The $\text{SDI}_e$ metric is calculated by considering the sum of four values: the number of classes which have been newly created, deleted, changed, and unchanged from the previous iteration [Olague et al., 2006].

Olague et al. performed a theoretical validation of the new metric using the Kitchenham et al. criteria and the Zuse requirements for software measures. Two case studies were utilized to compare the results of the SDI metric with the new $\text{SDI}_e$ metric. The $\text{SDI}_e$ metric was further studied by a comparison to the Chidamber and Kemerer metrics using the pairwise Spearman rank correlation [Olague et al., 2006].

Since the $\text{SDI}_e$ metric is calculated on information that is not normally available at design time; that is, the number of classes that have been changed, it is not typically considered a design metric

### 2.5 Expert Judgment

Expert judgment is collected from individuals recognized as experts due to their knowledge and experience in the field of interest. The use of expert opinion or judgment has been studied from several different perspectives by a diverse set of disciplines which include the cognitive sciences, medicine, meteorology, agriculture, physics, computer science, and engineering [Shanteau, 1992]. Studies have also been conducted which are concerned with the method of eliciting the expert opinion, how to interpret and analyze the expert judgment, and how to improve the use of expert judgment [Keeney and Winterfeldt, 1989; Sandri et al., 1995; Mumpower and Stewart, 1996; Genest and Wagner, 1987].

Expert opinion may be gathered either qualitatively or quantitatively. Quantitative representation of opinion requires the experts to express their judgments as a numerical value which lends itself to an ease of analysis. Experts, however, may prefer to use qualitative opinions expressed as words. There seems to be a "comfortable vagueness of words" which can allow the experts to express their own vagueness about a subject. Numeric values on the other hand seem to force experts to be more precise and demand more consideration in reaching the opinion [Keeney and Winterfeldt, 1989].

Shanteau identified five factors which influence the competence of experts: domain knowledge, psychological traits, cognitive skills, decision strategies, and task characteristics. It was his viewpoint that the performance of the expert was most dependent on the task characteristics. He stated that tasks which involve problem decomposition, static stimuli, more predictability, repetitive tasks, and feedback availability resulted in a higher competence of experts [Shanteau, 1992].

The process of utilizing expert opinion in research generally follows the following steps:

1. The Problem under study is defined.
2. The Experts are selected.
3. The Experts are trained.
4. The Opinions of the Experts are elicited.
5. The Opinions are aggregated.
6. The Decision is made based on the aggregated opinion [Li and Smidts 2003].

Questions regarding the number of experts to select, the method of eliciting the opinion, and the method for aggregating the opinions are sometimes difficult to answer. Several different uses of expert judgment can be found in the literature [Dyba, 2005; Li and Smidts, 2003; Kitchenham et al., 1997; Bansiya and Davis, 2002; Etzkorn et al., 2004; Etzkorn et al., 2001; Bansiya et al., 1999]. These examples demonstrate a wide ranging area for applications of expert judgment.

**2.6 Academic Course Utilization in Software Research**

The use of academic courses to gather data can be seen within a number of documented studies. Laurie Williams has written several articles illustrating the study of pair programming within university courses of different levels [Williams and Kessler, 2000; Williams and Kessler, May 2000; Williams and Kessler., December 2000; Williams and Upchurch, February 2001]. Matthias Müller conducted an experiment which compared pair programming with peer review [Müller, 2005]. Ciolkowski pointed out in his study that although the academic setting may not be truly representative of the industrial setting, the data collected can be useful to help improve current industrial software engineering processes. He stated that there were several benefits. For example, the study can be more cost effective than attempting to study in the industrial setting. Because the students within the graduate level class at the University of Kaiserslautern where the data was collected were exposed to the next approach or technique being studied, he felt they benefited from the study as well and stated that he plans to continue utilizing the students for data collection [Ciolkowski et al., 2004]. Höst et al. reported that the difference between computer science professionals and students is relatively small [Höst et al., 2000]. Many students are working in professional positions as well [Sjøberg et al., 2002].

Jeffrey Carver studied several examples of empirical studies conducted within three countries with the goal of validating a research hypothesis which have used students as subjects and listed several benefits to the students. These included the exposure to state-of-the-art topics, industrial problems, hands-on practice, demonstration that there is a need for evidence using quantitative methods to base improvement in some software process and that they should be prepared to be subjected to assessment throughout their career. He also highlighted the benefits to the researchers making use of student subjects. These include obtaining preliminary evidence to support or disprove a hypothesis, being able to control factors surrounding the study, demonstrating to industry the need to continue the study, and preparing the details of a study before carrying it out in an industrial setting. In addition, Carver pointed out that there are benefits to the instructors of the courses in which the research is being conducted. It could serve to stimulate them into a new manner of teaching, encourage them to incorporate new problems in their courses, stimulate teamwork, enhance the channels of communication between the students and the instructor, aid in maintaining the students' attention, and encourage the development of a critical attitude. In addition to the benefits, Carver also listed some possible costs which might arise when using students as subjects. In his opinion, the benefits clearly outweighed the costs. He concluded with advice for researchers using students to gather data. Making sure the study is clearly integrated with the goals of the course, setting realistic time estimates, motivating the student subjects about the study, and allowing students to give and receive feedback were listed as recommendations for a successful result [Carver et al., 2000; Carter et al., 2006].

Several other studies have also utilized the students in an academic course. Basili, when validating object-oriented design metrics related to software quality, used students in an upper level undergraduate/graduate level course at the University of Maryland [Basili et al., 1996]. Berander reported on the use of fourth year Software Engineering Master's students when studying the process of requirements prioritization [Berander, 2004]. James Noble reported on an experiment to incorporate more agile, iterative approaches within a capstone software project course [Noble et al., 2004].

# CHAPTER 3

# RESEARCH DESCRIPTION

For this research, we performed the following studies:

Study 1. an examination of stability across agile software iterations.

a. One experiment compared existing stability metrics (SDI and $SDI_e$) to each other and compared $SDI_e$ to the C&K metrics.

b. Another experiment compared the stability metrics (SDI and $SDI_e$) to the Bansiya TQI.

c. In another experiment, we attempted to develop fault prediction models across the agile iterations using the C&K metrics suite.

Study 2. an examination of the relationships between refactoring, faults, and reusability across agile software iterations.

Study 3. a comparison of the reusability of software developed using an agile method to software developed using a plan-based method.

a. One experiment examined paired data, that is, the applications that were agilely-developed and the same applications that were developed using a plan-based method were rated for reusability by the same experts.

b. One experiment examined unpaired data, that is, the experts who analyzed the agilely-developed software were not the same experts that analyzed the software developed using a plan-based method.

Studies 1 and 2 were performed on agilely-developed software only, whereas Study 3 compared agilely-developed packages to non-agilely-developed packages.

The software packages examined in this study were collected in senior level software engineering courses taught by two different professors, one at the University of Alabama in Huntsville and the other at the University of North Alabama. A complete description of these packages and how this data was collected is given in Section 3.1 below. Various tools were employed to collect the stability metrics, the C&K metrics, and the Bansiya metrics. These tools are described in Section 3.2 below. The reusability studies required human evaluators to rate classes and packages for reusability. This is described in Section 3.3 below. Detailed descriptions of the steps performed for each study are given in Sections 3.4, 3.5, and 3.6 below.

In this research, we examined 24 packages, and the reusability analysis involved 118 evaluators. We have not encountered any empirical studies of this magnitude in any of our research. The number of projects considered, the number of expert evaluators, the use of more than one university course, more than one professor, more than one semester and the use of both traditional plan-based development and agile development on the same applications all contribute to the uniqueness of this work.

## 3.1 Software Packages Examined in Empirical Studies

The data used within this study began with three different applications assigned as semester assignments in senior level software engineering courses using either a traditional plan-based method or an agile method. These projects were assigned at two different universities, the University of Alabama in Huntsville and the University of North Alabama, by two different professors across multiple semesters. Software teams at each institution received the same problem assignments with the same allocation of time for the assignment. Each agile project was assigned to a programming team consisting of four members and given the same amount of time (eight weeks) and number of iterations to complete. All of the projects were implemented in the programming language C++. There were five traditionally designed or plan-based projects and five agilely developed projects. Table 3.1 lists the class numbers for each of the traditionally developed projects.

**Table 3.1: Traditional (Plan-based) Project Information**

| Project | Number of Classes |
|---|---|
| P1 | 4 |
| P2 | 13 |
| P3 | 7 |
| P4 | 3 |
| P5 | 5 |

Each of the agile projects was delivered in four iterations with the exception of one team which only delivered three iterations. Each iteration was two weeks in duration. The agile project teams also reported fault, refactoring and effort data. Table 3.2 lists the class numbers for each of the five agile projects labeled A thru E.

**Table 3.2: Agile Project Information**

| Project | Number of Classes |
|---|---|
| A1 | 5 |
| A2 | 5 |
| A3 | 6 |
| A4 | 9 |
| B1 | 12 |
| B2 | 19 |
| B3 | 20 |
| B4 | 22 |
| C1 | 1 |
| C2 | 6 |
| C3 | 6 |
| C4 | 7 |
| D1 | 4 |
| D2 | 4 |
| D3 | 8 |
| D4 | 8 |
| E1 | 7 |
| E2 | 18 |
| E3 | 19 |

### 3.1.1 Software Applications Implemented

One application (M) was a database management application for an organization employing providers of services and having members paying membership fees in order to receive services from the providers. The providers were allowed to enter services after validating members. The program supported the editing, addition, suspension, and deletion of members and editing provider information in the database. It also created an electronic fund transfer file and produced weekly reports which were generated automatically at midnight on Friday evening. (This application was implemented for agile Projects A and E and traditional projects P3 and P4.) Another application (V) supported a company maintaining several stores in several regions. A hierarchy of users existed consisting of three levels: store managers, regional managers, and a vice-president. Each successive level had the privileges of the levels beneath it. Monthly target sales, actual sales and whether objectives were met were reported. (This application was implemented for agile Projects B and C and traditional project P5 discussed in later chapters.)

Another application (S) was to implement a scheduler for a computer science department. It utilized a classroom database and a list of valid class times. The input was a list of course requests which gave priority scheduling to graduate courses and it produced a schedule for the semester including a list of course conflicts. (This application was implemented for agile Project D and traditional projects P1 and P2 discussed in later chapters.)

Table 3.3 demonstrates the relationships of the projects.

**Table 3.3: Project Relationships**

| Application | Agile Projects | Traditional Projects |
|---|---|---|
| M | A1-A4<br>E1-E3 | P3<br>P4 |
| V | B1-B4<br>C1-C4 | P5 |
| S | D1-D4 | P1<br>P2 |

## 3.2 Tools Used in Empirical Studies

After they were completed, the projects were analyzed using tools to collect metrics and compare the iterations. We collected the C&K metrics data using IPL's Cantata++ 4.1 analysis tool [Cantata, 2004]. Both system level and class level metrics were collected with the results stored in Excel spreadsheets. We gathered the data regarding the change in the class-level metrics from one iteration to another as they were represented in spreadsheets using the differencing tool Synkronizer® 9.1 XL, which compared two spreadsheets and highlighted changes [Synchronizer, 2006]. We also used the Comparison tool Compare It 3.5 in order to determine code changes [Compare It]. Data was stored in Excel spreadsheets.

## 3.3 Reusability Data Collection Using Human Evaluators

The projects were organized into groups of two or more in two different manners. First, they were grouped primarily by size in order to not overly burden any one evaluator. A second set of groupings was organized in order to have a pair of two projects, one traditionally created and one created using an agile method for the same application to be considered by the same evaluator.

In the first grouping, which we call the "unpaired" grouping, we specifically did not want evaluators of an agile project to be the same evaluators for the non-agile version of the project. The purpose here was to minimize any bias that resulted from the evaluator having already seen another version of the same project. In the second grouping, which we call the "paired" grouping, we gave the agile and non-agile versions of the same project to the same evaluators. The purpose here was to minimize variance resulting from different evaluators with different opinions.

These were presented to evaluators, and the evaluators were asked to fill in a questionnaire related to each class (See Appendix A), a questionnaire for the entire project or package (See Appendix B) and a demographic questionnaire (See Appendix C) which might provide needed data for further studies. The evaluators were presented with the projects, directions, copies of the questionnaires, informed consent statement, and spreadsheet with code numbers for the projects and classes within the projects on CD-ROM, along with hard copies of the directions, questionnaires, informed consent statement, and spreadsheet of code numbers and a self addressed envelope for the return of the signed informed consent form. We also encouraged the evaluators to contact us with any questions they might have. The evaluators were assured of their anonymity. Also, the evaluators were not told which methodology was used in the construction of each project which was recognized by only a project number. Any identifying documentation within the projects was removed, thus the programmers also were anonymous. The evaluators were also asked to provide personal demographic information through another website which was associated with a unique evaluator code. This allowed for the association of all questionnaire responses from one evaluator to be considered as a group while the evaluator remains anonymous.

Our questionnaire that we used in our research has been used in previous research by Etzkorn [Etzkorn, 1997]. Although it was previously used in paper form, we felt that the implementation of the questionnaire as a web-based instrument which collected the responses in a comma separated file would be more user friendly. The unpaired data was collected in this manner. We feel that this made it more user-friendly to both the evaluators and the researcher and encouraged more experts to participate. However, when we administered the questionnaire to the second group of evaluators using the paired plan-based and agile for the same application, we were unable to use the web-based instrument and were forced to have the results delivered in excel files. The consolidation of these files made us appreciate the web-based instrument.

### 3.3.1 Internal Consistency Reliability of the Questionnaire

Internal consistency reliability is a trait to be considered with regard to an evaluation instrument (questionnaire) involving human responses on a number of items. Cronbach's alpha is one of the most commonly used values to estimate such a trait. Reliability of an instrument may be defined to be "the extent to which [measurements] are repeatable and that any random influence which tends to make measurements different from occasion to occasion is a source of measurement error" [Nunnally, 1976]. Cronbach's alpha is a coefficient found as the mean of all split-half reliabilities [Cortina, 1993]. It is a value less than or equal to one and an alpha value of 0.70 or greater is generally desired in order to view an evaluation instrument as reliable.

Part of our questionnaire analyzed the software packages themselves (rather than individual classes), and addressed the areas of hierarchy, size, reusability, flexibility, understandability, functionality, extendibility, and effectiveness, culminating in a final evaluation of total quality. The Cronbach's alpha value was computed for three sets of per-package data: paired, unpaired and the paired and unpaired combined. Table 3.3 below demonstrates our findings. Each of the categories resulted in an alpha value over

0.70, which indicates the per-package questionnaire is acceptable. We found it surprising that the alpha value for the paired data in which each evaluator was given a traditional plan-based project and an agilely developed project for the same application was lower than when the evaluators did not receive the same type of application to evaluate.

**Table 3.4: Cronbach's Alpha for Package Data**

| Evaluation Type | Cronbach's Alpha |
|---|---|
| Paired Data | 0.740 |
| Unpaired Data | 0.883 |
| Combined Data | 0.846 |

Part of our questionnaire analyzed individual classes, and addressed the topics of cohesion, coupling, modularity, interface, documentation, size, complexity, simplicity, encapsulation, composition, inheritance, abstraction, polymorphism culminating in a final evaluation of reusability. The Cronbach's alpha value for the three sets of per-class data—paired, unpaired, and the paired and unpaired combined—are given in Table 3.4. Once again, all of the alpha values are greater than 0.70 from which we can conclude reliability of the per-class questionnaire is acceptable.

Table 3.5: Cronbach's Alpha for Class Data

| Evaluation Type | Cronbach's Alpha |
|---|---|
| Paired Data | 0.820 |
| Unpaired Data | 0.816 |
| Combined Data | 0.812 |

### 3.3.2 Description of Expert Evaluators

Experts were solicited from a pool of colleagues and industry personnel working in computer science, as well as graduate and undergraduate students at the University of Alabama in Huntsville and undergraduate students at the University of North Alabama. They were sent a request for their participation and upon receiving an affirmative reply, they were then mailed a packet containing the CD-ROM, printed material listed earlier, and the self addressed envelope. They were given a one month period in which to complete their evaluation. We expected that the evaluators would take from eight to ten hours of effort in order to complete the task. Our goal was to have at least seven evaluations for each project. We were able to reach or exceed that goal for each one of the projects. Each project was evaluated by at least seven different evaluators and some had as many as fourteen evaluations.

One hundred and eighteen evaluators were used overall. Most of the evaluators (seventy nine) volunteered for the work, but some evaluators (thirty nine) were required to do the work for a course grade. The average number of years of experience of the evaluators in object-oriented programming was 6.2 years. This value was collected from the demographics questionnaire. Evaluators did not distinguish between student experience and industry experience. The overall demographic data of the evaluators is given in Appendix D.

The evaluators were offered the opportunity to receive the results of the research when it is completed. Although the evaluators were asked to make a commitment of time in order to perform the analysis of the projects, it is our belief that the evaluators in turn benefited from the study. They received a review of terminology and also a discussion of what needs to be considered when evaluating software for attributes such as cohesion, coupling and complexity.

This proposed evaluation methodology was approved by the Human Subjects Committee of the University of North Alabama and the IRB Human Subjects Committee of the University of Alabama in Huntsville. See Appendix E.

### 3.3.3 Threats to Validity

We performed our study by considering twenty five projects implemented to solve three different applications. The programmers of the agile projects gathered defect data during software development. With the student projects, we minimized the effects of confounding factors to avoid bias by comparing student projects developed at different universities and conducted by different professors, comparing student projects developed during different semesters, and comparing student projects developed for the same application using the two different development processes. According to Wohlin et al., case studies are useful when performing comparative studies [Wohlin et al., 2000].

We minimized bias by selecting the evaluators in different ways. In the "unpaired" grouping, we specifically did not want evaluators of an agile project to be the same evaluators for the non-agile version of the project. The purpose here was to minimize any bias that resulted from the evaluator having already seen another version of the same project. In the "paired" grouping, we gave the agile and non-agile versions of the same project to the same evaluators. The purpose here was to minimize variance resulting from different evaluators with different opinions.

We considered the four types of threats to validity as presented by Cook and Campbell: external, internal, construct, and conclusion [Cook and Campbell, 1979; Wohlin et al., 2000]. External validity represents the ability to generalize results from our research to industrial practice. We planned to reduce the threat to external validity by making use of different levels and types of expert evaluators. We used senior level computer science students, graduate computer science students, computer science instructors, and industry professionals. We also used student projects developed during different semesters to prevent discussion of one paradigm influencing teams using a different paradigm. Internal validity represents the ability to conclude a causal relationship between the treatment and the effect. We attempted to reduce the threat to internal validity by using both volunteers who received no compensation and students in classes who were assigned the evaluation as an assignment. We also gave the evaluators a reasonable timeframe in which to complete the evaluation. By making sure that no evaluator was aware of the type of development process used to develop the project, we attempted to insure construct validity. Finally, we made use of a large number of projects, classes and evaluators in order to insure conclusion validity also called statistical conclusion validity. This has to do with the ability to draw the correct conclusion from the study [Wohlin et al., 2000].

### 3.3.4 Interrater Reliability

Before considering aggregating the responses of the expert opinions given to the questionnaires, a confirmation that there was some type of agreement among the evaluators was necessary. This was necessary in order for us to consider the mean of the evaluations as being representative of the individual evaluators' values [Cohen et al., 2001]. A result of high interrater agreement would serve to indicate the reliability of the experts' subjective evaluations. James et al. presented the $r_{WG(J)}$ value as a within-group interrater reliability measure for judges scores where J represents the number of parallel items [James et al., 1984]. This value compares the observed within-group variances to an expected variance which would occur with random responses. Cohen et al. stated that the use of $r_{WG(J)}$ has two main advantages. First, because it does not depend on the between-group variances, it would be useful for data where the group mean has restricted range. Secondly, its use provides a measure for each group instead of just one measure for the entire population [Cohen et al., 2001].

The $r_{WG(J)}$ value was calculated for each package and each class within the package. When we considered the data, we determined that approximately ninety-two percent of the class $r_{WG(J)}$ values were in the acceptable range and a majority of the package values were as well. A rule of thumb that a value of .7 is "good" for interrater reliability is sometimes given while others point to .5 or greater as acceptable. acceptable [Cohen et al., 2001]. Since overall the per class $r_{WG(J)}$ values were in acceptable ranges, this meant that we were able to calculate the mean of the expert ratings for each class, and use this mean in statistical and graphical comparisons.

The $r_{WG(J)}$ values for the packages and classes are given in Appendices F and G due to their large size.

## 3.4 Detailed Description of Study 1

In the first part of Study 1, we first calculated the SDI, $SDI_e$, and C&K metrics. Then we compared the graphs of SDI and $SDI_e$, across the iterations of each software package. We then performed a pairwise Spearman's rank correlation of $SDI_e$ to the C&K metrics collected across the iterations. We used Spearman's rather than Pearson's correlation here because the data was not normally distributed, according to the Kolmogorov-Smirnoff test. A complete description of this part of Study 1 is given in Chapter 4 below. It should be noted that stability may be considered using structural (logical) or behavorial (performance) changes. In our discussion of stability, we will consider structural changes.

Next, in the second part of Study 1, we compared the graphs of the SDI and $SDI_e$ metrics to the Bansiya Total Quality Index (TQI). A complete description of this part of Study 1 is given in Chapter 4 below.

Lastly, in the third part of Study 1, we attempted to develop regression models that would predict faults across the agile iterations using the C&K metrics suite. We first performed a collinearity analysis of the C&K metric values. We examined the Variance Inflation Factor (VIF) and the Condition Number. The variance inflation factor (VIF) is the reciprocal of the tolerance which indicates the variance percentage unaccounted for by other independent variables within the regression equation [Olague et al., 2007]. The VIF value may be used as a threshold for determining whether multicollinearity exists [Mansfield and Helms, 1982]. The condition number, the ratio of the square root of the largest eigenvalue to all the others is also an indicator that multicollinearity may be a problem [Olague et al., 2007]. A rule of thumb used for linear regression is a threshold of 10 for VIF which when exceeded calls for further study. Using the VIF and condition number as a rationale for choosing various subsets of the C&K metrics, we then attempted to develop linear regression models using these various subsets of the C&K metrics as the independent values and the fault values as the dependent values. A complete description of this part of Study 1 is given in Chapter 5. below.

### 3.5 Detailed Description of Study 2

In Study 2, we examined the relationships between refactoring, faults, and reusability across agile software iterations. We first graphed each of these values across the iterations. Because trends in the data were difficult to see from these graphs, we normalized the data by subtracting the mean and dividing by the standard deviation for each of the three values. Finally, we performed a Spearman's correlation of faults vs. refactoring, of faults vs. reusability ratings, and of refactoring vs. reusability ratings. We used Spearman's rather than Pearson's correlation here because the data was not normally distributed, according to the Kolmogorov-Smirnoff test. A complete description of Study 2 is given in Chapter 6 below.

### 3.6 Detailed Description of Study 3

In Study 3, we compared the reusability of software developed using an agile method to software developed using a plan-based method. This consisted of two separate experiments, a paired experiment and an unpaired experiment. These experiments were performed on a per-class level. That is, the reusability of the classes was examined, not the (separately rated) reusability of the packages.

The paired experiment consisted of five agilely developed projects and five traditionally developed projects consisting of a total of fifty six classes. The unpaired experiment consisted of five traditionally developed projects and five agilely developed projects as well with a total of ninety five classes.

In the "unpaired" grouping, we specifically did not want evaluators of an agile project to be the same evaluators for the non-agile version of the project. The purpose here was to minimize any bias that resulted from the evaluator having already seen another version of the same project. In the "paired" grouping, we gave the agile and non-agile versions of the same project to the same evaluators. The purpose here was to minimize variance resulting from different evaluators with different opinions.

- For each experiment, the hypothesis to be tested was as follows:
    - $H_0$: There is no significant difference in the reusability of the software developed using highly iterative methods from those developed using traditional plan-based methods.

❑ $H_1$: There is a significant difference in the reusability of the software developed using highly iterative methods from that developed using traditional plan-based methods.

For each class in the unpaired experiment, there were seven expert ratings of reusability. We were able to compute the average of reusability for each class over these seven experts since the $r_{WG(J)}$ interrater reliability values were acceptable per class.

Each class can be considered to be independent of each other class in the software. All the software was developed using the object-oriented paradigm which specifies that encapsulation, or information hiding, is the primary objective when defining each class [Snyder, 1986]. Encapsulation means that interdependencies between classes are kept to an absolute minimum. Thus, the reusability ratings of each class can be considered independent. Since the classes are independent, it was legitimate to compare the mean over the reusability ratings of the agilely-developed classes to the mean over the classes developed using a plan-based method using a t-test: the statistical assumptions behind the t-test were met.

A complete description of Study 3 is given in Chapter 7 below.

# CHAPTER 4

# STABILITY

The stability of software indicates to software professionals whether the software is mature enough for delivery to a customer. Stability metrics attempt to quantify the current software stability level and can be examined to see if software is approaching stability as the development deadlines near.

## 4.1 Validation of the SDI$_e$ Metric

The purpose of one area of our research is the further validation of two existing stability metrics: the SDI metric proposed by Li et al. [Li et al., 2000] and the SDI$_e$ metric proposed by Olague et al. [Olague et al., 2006] as a modification to the earlier SDI metric. The research on stability was accomplished by means of examining four iterations for each of four student projects. These projects were assigned to follow the extreme programming paradigm as closely as possible. Teams programmed in pairs, used story cards, used a simple design, performed planned testing, refactored, performed continuous integration, had small releases, and the instructor for the course served as the on-site customer with availability daily through office hours and by email.

The data required to calculate the stability metrics is shown in Table 4.1 in the following format:

- Summary table containing for each iteration:
    - a = number of classes with class name change
    - b = number of newly added classes
    - c = number of deleted classes
    - m = total number of classes in the previous iteration.
    - The SDI values for each iteration with SDI = (a + b + c) / m * 100
    - $C_1$ = number of classes added.
    - $C_2$ = number of classes deleted.
    - $C_3$ = number of classes with metric changes.
    - $C_4$ = number of classes unchanged.
    - N = total number of classes in this iteration.
    - The $SDI_e$ values for each of the four iterations, calculated using

      $SDI_e = -((C_1/N) \log_2(C_1/N) + (C_2/N) \log_2(C_2/N) + (C_3/N) \log_2(C_3/N) + (C_4/N) \log_2(C_4/N))$

A manual inspection of the source code for each of the packages was used in order to determine whether there were any classes whose names were changed. The inspection of the code to determine whether classes were changed was performed using the class level metrics calculated with the Cantata ++4.1 metrics tool and the Synkronizer® XL comparison tool which compares Excel spreadsheets. We performed a Spearman's rank correlation analysis between the $SDI_e$ metric and the sum of all the classes in the project for each of the six C&K metrics using the SPSS® 13.0 statistical analytic tool. The correlation between the SDI$_e$ metric and the average of each of the C&K metrics was also computed. Spearman's rank correlation may be used to represent the strength of the relationship between two variables resulting in values ranging from -1 to 1. A value near one indicates a strong positive relationship, a value near zero indicates a weak relationship and a value near -1 represents a strong negative correlation. The variables being studied are not required to be normally distributed [Sheskin 2004].

**Table 4.1: Stability Data, SDI, and SDI$_e$**

| Version | a: Name changed | b: Added | c: Deleted | m: Total classes | **SDI** | $C_1$: Classes Added | $C_2$: Classes Deleted | $C_3$:Classes Changed | $C_4$:Classes Unchanged | N: Total Classes | **SDI$_e$** |
|---|---|---|---|---|---|---|---|---|---|---|---|
| A1 | 0 | 5 | 0 | 5 | 0 | 5 | 0 | 0 | 0 | 5 | 0 |
| A2 | 0 | 0 | 0 | 5 | 0 | 0 | 0 | 4 | 1 | 5 | 0.722 |
| A3 | 0 | 1 | 0 | 6 | 20 | 1 | 0 | 0 | 5 | 6 | 0.650 |
| A4 | 0 | 3 | 0 | 9 | 50 | 3 | 0 | 5 | 1 | 9 | 1.352 |
| | | | | | | | | | | | |
| B1 | 0 | 12 | 0 | 12 | 0 | 12 | 0 | 0 | 0 | 12 | 0 |
| B2 | 0 | 8 | 1 | 19 | 75 | 8 | 1 | 11 | 0 | 19 | 1.219 |
| B3 | 0 | 1 | 0 | 20 | 5 | 1 | 0 | 17 | 2 | 20 | 0.748 |
| B4 | 1 | 3 | 1 | 22 | 25 | 4 | 2 | 18 | 1 | 22 | 1.242 |
| | | | | | | | | | | | |
| C1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 |
| C2 | 0 | 5 | 0 | 6 | 500 | 5 | 0 | 1 | 0 | 6 | 0.650 |
| C3 | 0 | 1 | 0 | 7 | 16.667 | 1 | 0 | 3 | 3 | 7 | 1.449 |
| C4 | 0 | 0 | 0 | 7 | 0 | 0 | 0 | 5 | 2 | 7 | 0.863 |
| | | | | | | | | | | | |
| D1 | 0 | 4 | 0 | 4 | 0 | 4 | 0 | 0 | 0 | 4 | 0 |
| D2 | 0 | 4 | 4 | 4 | 200 | 4 | 4 | 0 | 0 | 4 | 1 |
| D3 | 0 | 4 | 0 | 8 | 100 | 4 | 0 | 4 | 0 | 8 | 1 |
| D4 | 0 | 0 | 0 | 8 | 0 | 0 | 0 | 4 | 4 | 8 | 1 |
| | | | | | | | | | | | |

The comparison graphs for SDI and $SDI_e$, and the pairwise Spearman's rank correlations between the C&K metrics and $SDI_e$ are shown in

- Project A: Figures 4.1, 4.2, Table 4.2
- Project B: Figures 4.3, 4.4, Table 4.3
- Project C: Figures 4.5, 4.6, Table 4.4
- Project D: Figures 4.7, 4.8, Table 4.5

Tables 4.2, 4.3, 4.4, and 4.5 contain analysis of both the sum and the average of the C&K metrics across the classes of the project.

### 4.1.1 Project A Stability Analysis

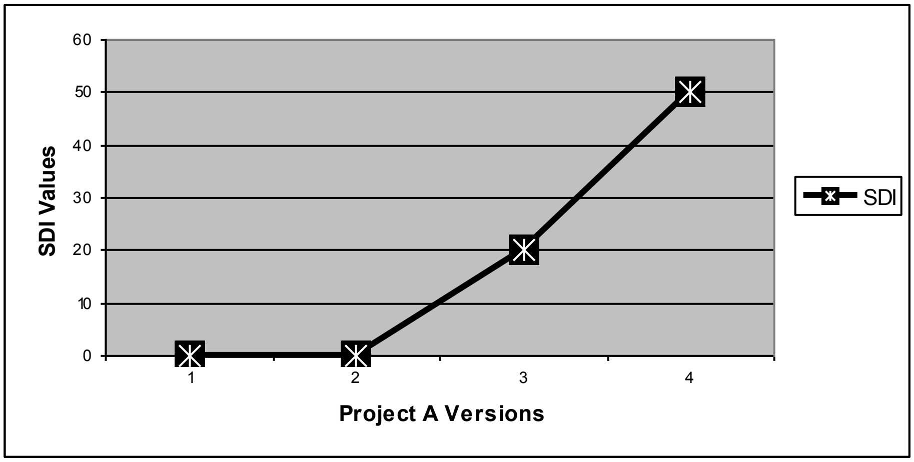


**Figure 4.1: Project A SDI Results**

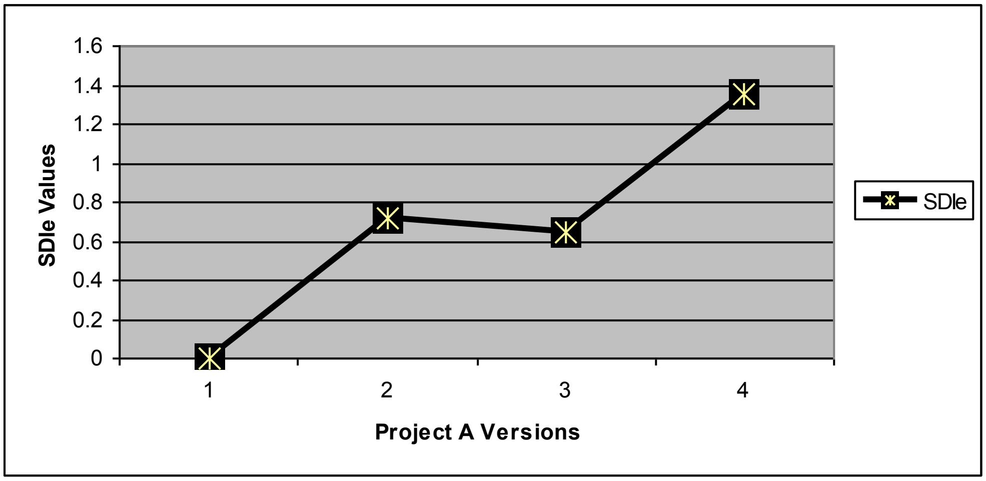


**Figure 4.2: Project A SDI$_e$ Results**

**Table 4.2: Pairwise Spearman's Rank Correlation between C&K metrics and the SDI$_e$ metric for Project A**

| | LCOM | DIT | CBO | NOC | RFC | WMC |
|---|---|---|---|---|---|---|
| Sum of C&K | | | | | | |
| Correlation | .949 | .775 | .949 | . | .949 | .949 |
| ρ-value | .051 | .225 | .051 | . | .051 | .949 |
| Significant at α = 0.05? | NO | NO | NO | | NO | NO |
| Significant at α = 0.10? | YES | NO | YES | | YES | NO |
| Avg. of C&K | | | | | | |
| Correlation | .316 | . | .316 | . | .949 | .316 |
| ρ-value | .684 | . | .684 | . | .051 | .684 |
| Significant at α = 0.05? | NO | | NO | | NO | NO |
| Significant at α = 0.10? | NO | | NO | | YES | NO |

Project A implemented a database management application for an organization employing providers of services and having members paying membership fees in order to receive services from the providers. The providers were allowed to enter services after validating members. The program supported the editing, addition, suspension, and deletion of members and editing provider information in the database. It also created an electronic fund transfer file and produced weekly reports which were generated automatically at midnight on Friday evening.

Entries within the pairwise Spearman's rank correlation tables which are empty indicate the value of the given system-level metric is extremely small, near zero. Zero values for the NOC sum and average are a result of the lack of inheritance for a majority of the classes. The discrepancy between the DIT and the NOC numbers may be traced to the use of Microsoft Foundation Class library classes for which Cantata++ calculated a DIT value of one even though there were no children classes of locally defined classes within the project. Project A's results in Table 4.2 above are inconclusive overall due to lack of significance. Although none of the correlations were significant at $\alpha = 0.05$, four of the correlations, Sum of LCOM, Sum of CBO, Sum of RFC, and Average of RFC were significant at $\alpha = 0.10$. In the Olague et al. paper, only the Average of C&K LCOM was significant at either level $\alpha = 0.05$ or $\alpha = 0.10$ [Olague et al., 2006]. None of the correlations was significant in the Li et al. study [Li et al., 2000]. Due to lack of significance, it is not clear whether SDI$_e$ is measuring the same thing as any of the C&K metrics or not (although the significance at $\alpha = 0.10$ of some of the results does lead one to wonder if there is some overlap of the SDI$_e$ metric with Sum of LCOM, Sum of CBO, Sum of RFC, and Average of RFC) [Roden et al., November 2007].

The fact that the changes from the first iteration to the second iteration were only changes to existing classes is reflected in the different shapes of Figures 4.1 and

4.2 above. No new classes were added and none of the class names were changed. Thus the SDI metric would have a value of zero but the $SDI_e$ metric would be a positive value. Similarly, between the second and third iterations, there was only one class added, causing the SDI metric to increase while the $SDI_e$ metric decreased. Due to the fact that there were both added classes and changed classes from Iteration 3 to Iteration 4, both metrics increased. It should be pointed out that for each of the graphs generated for the SDI metric, the value for the first iteration will always be zero. Similarly, the beginning value in the graph for the $SDI_e$ metric will always be zero [Roden et al., November 2007].

### 4.1.2 Project B Stability Analysis

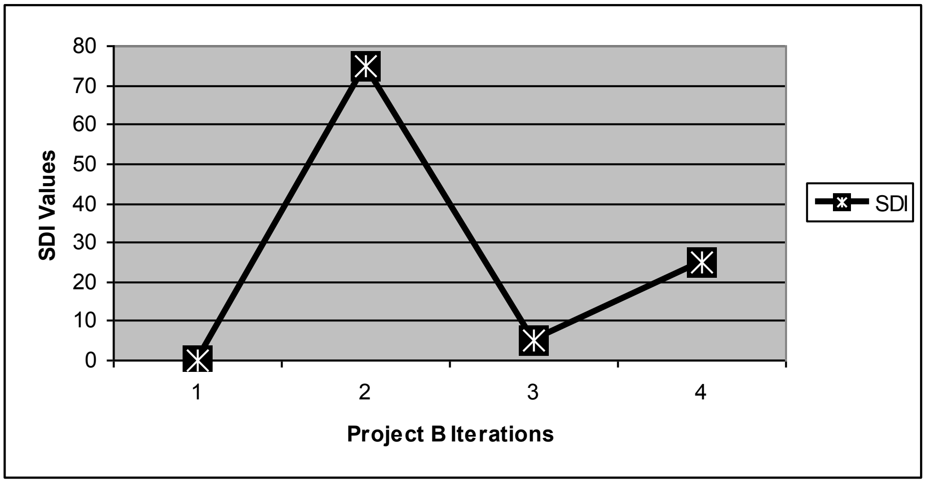


**Figure 4.3: Project B SDI Results**

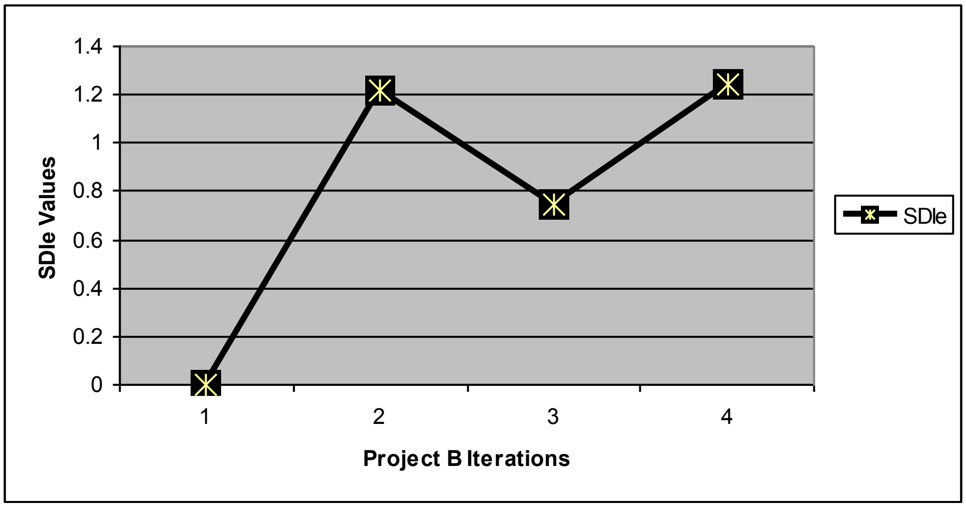


**Figure 4.4: Project B SDI$_e$ Results**

**Table 4.3: Pairwise Spearman's Rank Correlation between C&K metrics and the SDI$_e$ metric for Project B**

| | LCOM | DIT | CBO | NOC | RFC | WMC |
|---|---|---|---|---|---|---|
| Sum of C&K | | | | | | |
| Correlation | .200 | .400 | .400 | . | .800 | .600 |
| ρ-value | .800 | .600 | .600 | . | .200 | .400 |
| Significant at $\alpha = 0.05$? | NO | NO | NO | | NO | NO |
| Significant at $\alpha = 0.10$? | NO | NO | NO | | NO | NO |
| Avg. of C&K | | | | | | |
| Correlation | -.400 | -.20 | .400 | . | .400 | .600 |
| ρ-value | .600 | .800 | .600 | . | .600 | .400 |
| Significant at $\alpha = 0.05$? | NO | NO | NO | | NO | NO |
| Significant at $\alpha = 0.10$? | NO | NO | NO | | NO | NO |

Project B implemented an application which supported a company maintaining several stores in several regions. A hierarchy of users existed consisting of three levels:

store managers, regional managers, and a vice-president. Each successive level had the privileges of the levels beneath it. Monthly target sales, actual sales and whether objectives were met were reported. The correlation outlined in Table 4.3 above again demonstrates results similar to those generated by Olague et al., that is, that no correlations were significant [Olague et al., 2006]. Thus, again the results are inconclusive.

The different shapes of the graphs in Figures 4.3 and 4.4 can be attributed to the fact that the differences in first three iterations were greatly influenced by changes in already existing classes which greatly affected the calculated value of the $SDI_e$ metric but were not used in the calculations for the SDI metric. This programming team implemented the most classes in the first iteration and placed a greater emphasis on refactoring in the later iterations [Roden et al., November 2007].

### 4.1.3 Project C Stability Analysis

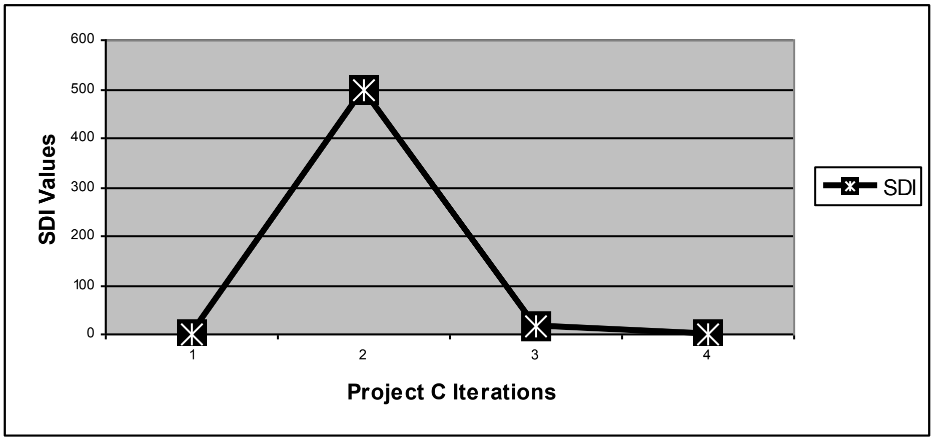


**Figure 4.5: Project C SDI Results**

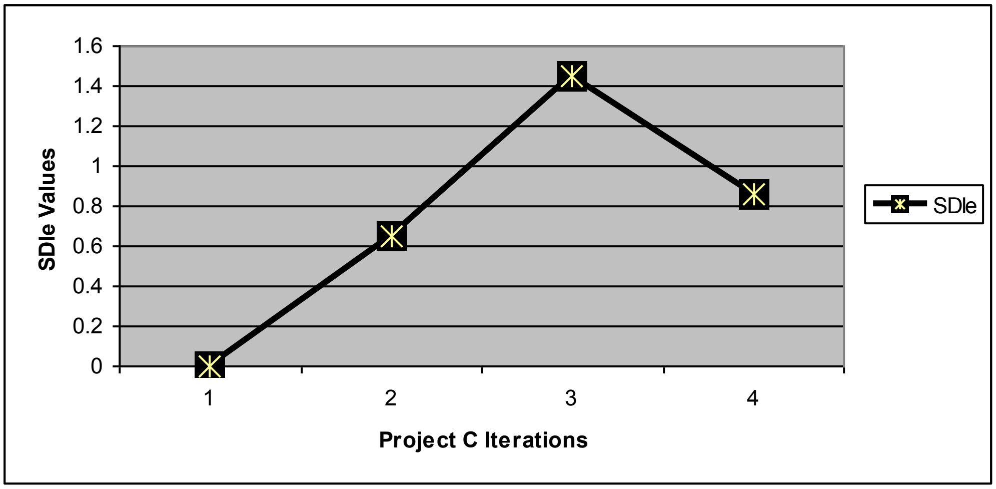


**Figure 4.6: Project C SDI$_e$ Results**

**Table 4.4: Pairwise Spearman's Rank Correlation between C&K metrics and the SDI$_e$ metric for Project C**

| | LCOM | DIT | CBO | NOC | RFC | WMC |
|---|---|---|---|---|---|---|
| Sum of C&K | | | | | | |
| Correlation | .738 | -258 | .738 | -.258 | .800 | .400 |
| ρ-value | .262 | .742 | .262 | .742 | .200 | .600 |
| Significant at α= 0.05? | NO | NO | NO | NO | NO | NO |
| Significant at α = 0.10? | NO | NO | NO | NO | NO | NO |
| Avg. of C&K | | | | | | |
| Correlation | -.400 | .258 | .600 | -.258 | .000 | -.400 |
| ρ-value | .600 | .742 | .400 | .742 | 1.00 | .600 |
| Significant at α = 0.05? | NO | NO | NO | NO | NO | NO |
| Significant at α = 0.10? | NO | NO | NO | NO | NO | NO |

Once again, the Spearman's rank correlation results in Table 4.4 above compare to the results from Olague et al. [Olague et al., 2006]. The shapes of the graphs in Figures 4.5 and 4.6 once again can be attributed to the fact that the $SDI_e$ metric calculation is affected by changes to existing classes while the SDI metric does not make use of the measurement of number of classes changed.

Projects B and C, which had the same requirements, analyzed above were assigned as student software engineering projects at the two different institutions by two different professors. The teams were given the same problem description and the same amount of time to complete the project. It should be noted that the graphs of the SDI metric for Projects B and C have differing shapes. Likewise, the graphs for the $SDI_e$ metrics differ as well. Further investigation of the development process revealed two major factors which could account for these differences. First, the two teams chose to implement different story cards at different iterations. Second, the number, size, and complexity of the classes in the two projects were vastly different. One team chose to distribute their new class development later into the iteration process while the other team chose to instead perform more refactoring in the last iteration [Roden et al., November 2007].

### 4.1.4 Project D Stability Analysis

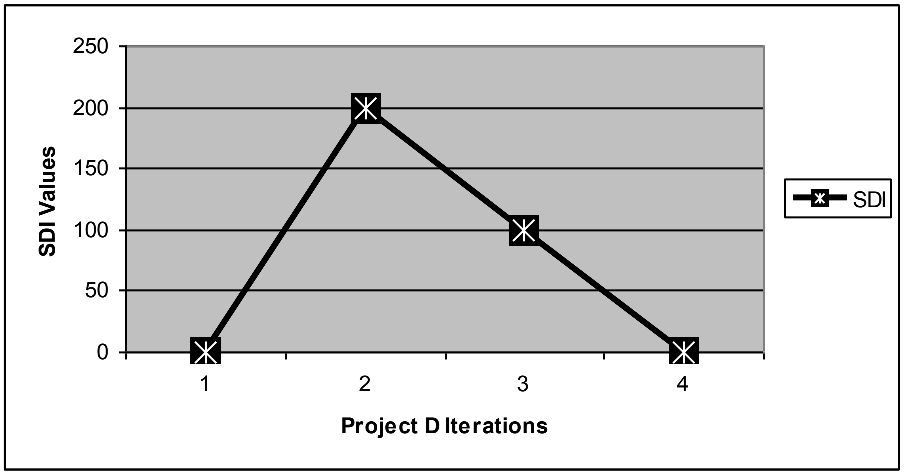


Figure 4.7: Project D SDI results

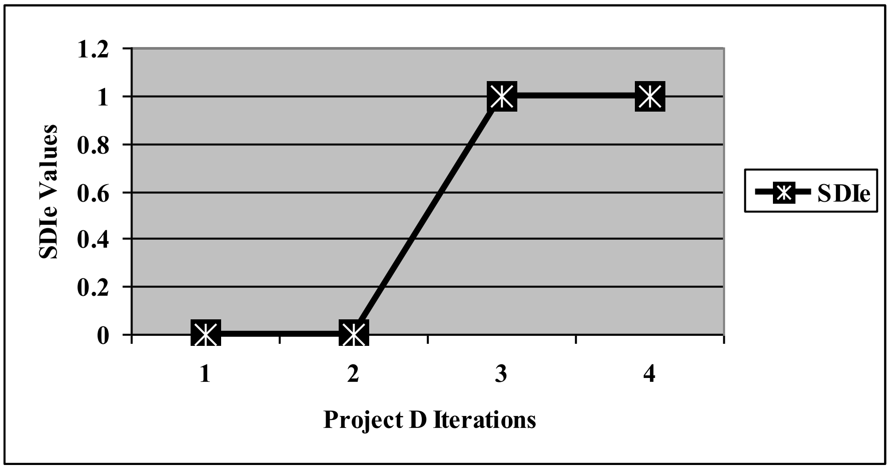


Figure 4.8: Project D $SDI_e$ Results

**Table 4.5: Pairwise Spearman's Rank Correlation between C&K metrics and the $SDI_e$ metric for Project D**

| | LCOM | DIT | CBO | NOC | RFC | WMC |
|---|---|---|---|---|---|---|
| Sum of C&K | | | | | | |
| Correlation | .943 | .577 | .943 | .577 | .943 | .943 |
| ρ-value | .057 | .423 | .057 | .423 | .057 | .057 |
| Significant at $\alpha$= 0.05? | NO | NO | NO | NO | NO | NO |
| Significant at $\alpha$ = 0.10? | YES | NO | YES | NO | YES | YES |
| Avg. of C&K | | | | | | |
| Correlation | .000 | .000 | .943 | .000 | .000 | .000 |
| ρ-value | 1.00 | 1.00 | .057 | 1.00 | 1.00 | 1.00 |
| Significant at $\alpha$ = 0.05? | NO | NO | NO | NO | NO | NO |
| Significant at $\alpha$ = 0.10? | NO | NO | YES | NO | NO | NO |

The purpose of Project D was to implement a scheduler for a computer science department. It utilized a classroom database and a list of valid class times. The input was a list of course requests which gave priority scheduling to graduate courses and it produced a schedule for the semester including a list of course conflicts.

This team apparently had some misunderstandings with regard to the feature of extreme programming which delivers small iterations with increasing and/or improved functionality. This team removed the entire functionality of the first iteration from the project's first iteration and introduced new functionality in the second iteration. For the third iteration the original functionality was reintroduced and refined for the fourth iteration. Due to this misunderstanding, the $SDI_e$ metric computes two consecutive zero values for the first two iterations because the same number of classes was added as was deleted in the second iteration. The change in the $SDI_e$ metric reflects that the functionality was restored between the second and third iterations. In the same manner the SDI metric demonstrates the largest change between the second and third iterations.

The results in Table 4.5 above for Project D are inconclusive overall due to lack of significance: none of the correlations were significant at $\alpha = 0.05$, although four of the correlations, Sum of LCOM, Sum of CBO, Sum of RFC, and Average of CBO were significant at $\alpha = 0.10$. Due to lack of significance, it is not clear whether SDI$_e$ is measuring the same thing as any of the C&K metrics or not (although the significance at $\alpha = 0.10$ of some of the results does lead one to wonder if there is some overlap with Sum of LCOM, Sum of CBO, Sum of RFC, and Average of CBO). It is interesting that three out of the four metrics were significant in Table 4.2, project A as well: Sum of LCOM, Sum of CBO, and Sum of RFC. However, in Project A it was Average of RFC instead of Average of CBO that was significant. Again, this leads one to wonder if there is some overlap of the SDI$_e$ metric with Sum of LCOM, Sum of CBO, Sum of RFC, and possibly either Average of RFC or Average of CBO. More study on larger data sets is required [Roden et al., November 2007 ].

### 4.2 Comparison of Stability and TQI

The second area of our stability research is concerned with determining if there is a relationship between the Total Quality Index of Bansiya and Davis and the stability metrics SDI and SDI$_e$ [Bansiya and Davis, 2002; Li et al., 2000; Olague et al., 2006]. This study arose from a serendipitous event in which two researchers using the same data for two separate research projects had each prepared a graph and shared the results in a meeting. One was studying the total quality index while the other was separately studying the stability metrics. It was an accidental comparison which caused this researcher to realize there could possibly be a close relationship between these two concepts.

This study used five undergraduate software engineering projects implemented using the extreme programming paradigm. Four of the projects contained four iterations each. These were the same four projects discussed in Section 4.1. An additional project, Project E had only three iterations due to a failure at delivery of one of the iterations. The following Table 4.6 indicates the computed values for the SDI and $SDI_e$ metrics, the values of the six QMOOD quality values, the Total Quality Index TQI, and the normalized values of SDI, $SDI_e$, and TQI. The values of SDI, $SDI_e$, and TQI were normalized by subtracting the respective mean and then dividing by the standard deviation through the aid of the Minitab tool. This was accomplished to facilitate simultaneous graphing of the concepts on one graph.

**Table 4.6: Stability and Quality Values**

| Package | SDI | SDIe | Reusability | Flexibility | Understandability | Functionality | Extendibility | Effectiveness | TQI | Normalized SDI | Normalized SDIe | Normalized TQI |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| A1 | 0.000 | 0.000 | 1.000 | 1.000 | -0.990 | 1.000 | 1.000 | 1.000 | 4.010 | -0.741 | -1.232 | -0.498 |
| A2 | 0.000 | 0.722 | 0.792 | -0.022 | 0.802 | 0.617 | -0.091 | 0.009 | 2.107 | -0.741 | 0.074 | -1.152 |
| A3 | 20.000 | 0.650 | 0.925 | 2.006 | -0.169 | 1.127 | 2.443 | 2.038 | 8.369 | 0.106 | -0.056 | 1.002 |
| A4 | 50.000 | 1.352 | 0.853 | 1.818 | -0.377 | 1.050 | 2.145 | 1.849 | 7.338 | 1.375 | 1.214 | 0.648 |
| | | | | | | | | | | | | |
| B1 | 0.000 | 0.000 | 1.000 | 1.000 | -0.990 | 1.000 | 1.000 | 1.000 | 4.010 | -0.770 | -1.380 | -1.160 |
| B2 | 75.000 | 1.219 | 4.156 | 16.687 | -12.612 | 5.639 | 11.606 | 9.340 | 34.818 | 1.424 | 0.718 | 0.080 |
| B3 | 5.263 | 0.748 | 3.499 | 14.361 | -10.461 | 4.610 | 8.350 | 7.773 | 28.133 | -0.616 | -0.094 | -0.189 |
| B4 | 25.000 | 1.242 | 4.759 | 1.568 | -0.163 | 5.532 | 37.049 | 15.624 | 64.369 | -0.038 | 0.756 | 1.270 |
| | | | | | | | | | | | | |
| C1 | 0.000 | 0.000 | 1.000 | 1.000 | -0.990 | 1.000 | 1.000 | 1.000 | 4.010 | -0.522 | -1.238 | -0.927 |
| C2 | 500.000 | 0.650 | 3.163 | 0.792 | -2.564 | 1.954 | 1.000 | 0.833 | 5.178 | 1.499 | -0.151 | 0.352 |
| C3 | 16.667 | 1.449 | 3.837 | 0.821 | -2.772 | 2.255 | 1.000 | 0.857 | 5.998 | -0.455 | 1.184 | 1.250 |
| C4 | 0.000 | 0.863 | 1.570 | 1.000 | -1.577 | 1.247 | 1.000 | 1.000 | 4.240 | -0.522 | 0.205 | -0.675 |
| | | | | | | | | | | | | |
| D1 | 0.000 | 0.000 | 1.000 | 1.000 | -0.990 | 1.000 | 1.000 | 1.000 | 4.010 | -0.783 | -1.500 | -0.059 |
| D2 | 200.000 | 1.000 | 0.876 | 2.007 | -1.319 | 0.823 | 1.818 | 1.727 | 5.932 | 1.306 | 0.500 | 1.353 |
| D3 | 100.000 | 1.000 | 0.534 | 0.523 | 0.520 | 0.575 | 0.778 | 0.832 | 3.761 | 0.261 | 0.500 | -0.241 |
| D4 | 0.000 | 1.000 | 0.705 | 0.266 | 0.427 | 0.611 | 0.254 | 0.393 | 2.656 | -0.783 | 0.500 | -1.053 |
| | | | | | | | | | | | | |
| E1 | 0.000 | 0.000 | 1.000 | 1.000 | -0.990 | 1.000 | 1.000 | 1.000 | 4.010 | -0.654 | -0.852 | -1.149 |
| E2 | 63.636 | 0.964 | 2.032 | 0.822 | -1.402 | 1.402 | 0.913 | 0.898 | 4.665 | 1.151 | 1.101 | 0.673 |
| E3 | 5.556 | 0.297 | 2.078 | 0.837 | -1.552 | 1.432 | 0.897 | 0.902 | 4.594 | -0.497 | -0.249 | 0.476 |
| | | | | | | | | | | | | |

First we graphed the set of six quality factors against $\text{SDI}_e$ for each of the five projects and were unable to see any common trends consistent in the majority of the projects. We next graphed each quality factor against $\text{SDI}_e$ for each package across the iterations. Once again, although we saw some relationship between different quality factors for different packages, no common trend emerged. We next graphed the normalized values of the SDI, SDI$_e$, and the TQI for each set of iterations for each project. We were able to detect some interesting relationships from these graphs.

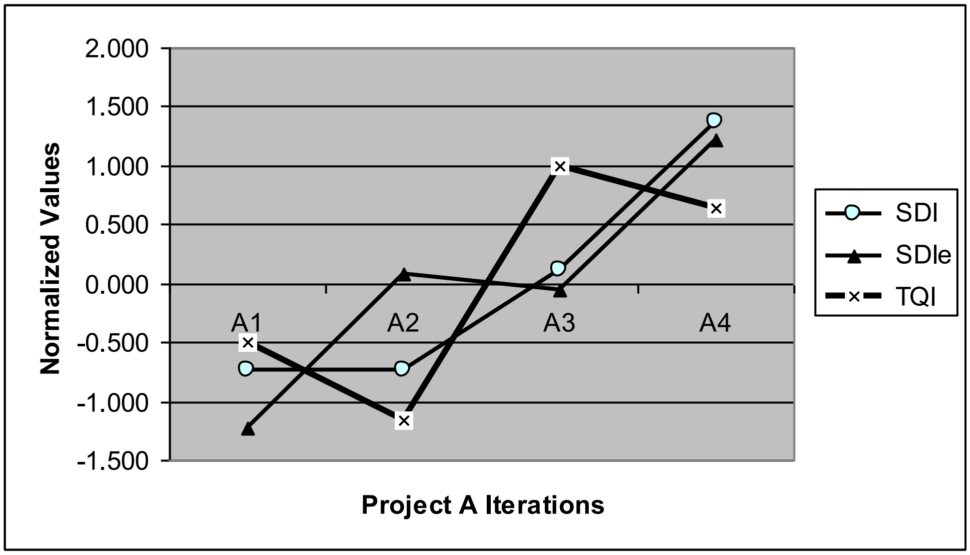


**Figure 4.9: Project A – TQI vs. Stability**

From a comparison of the graphs of the stability metrics and the TQI value for Project A in Figure 4.9 above, we were unable to determine that the TQI value could be used to demonstrate stability across the iterations. When we performed a qualitative examination of the development process, it was discovered that the students did not deliver any new classes at iteration number two but instead spent their time improving their existing code instead of adding new classes. Also, between iterations two and three, only one class was added and four out of the five classes already existing were refined. At the final iteration, three new classes were added in an apparent mad dash for the finish line. The uneven division of work across the iterations and the seeming rush at the end of the project should have resulted in the TQI value going down as it did.

Although the SDI metric graph seemed to follow the TQI graph during the first three iterations, the rush at the end apparently affected the final iteration differently.

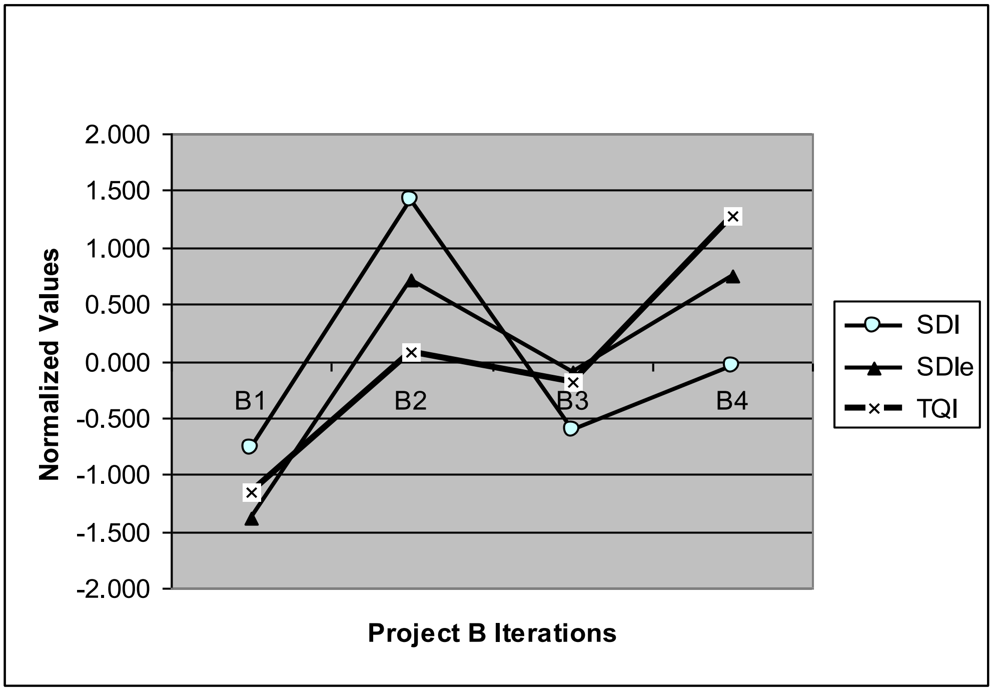


**Figure 4.10: Project B – TQI vs. Stability**

Figure 4.10 above for Project B shows that both the stability metrics graphs follow the graph of TQI. We should point out that, from a qualitative inspection, this student project did adhere to the extreme programming paradigm in a much closer fashion than did the previous Project A.

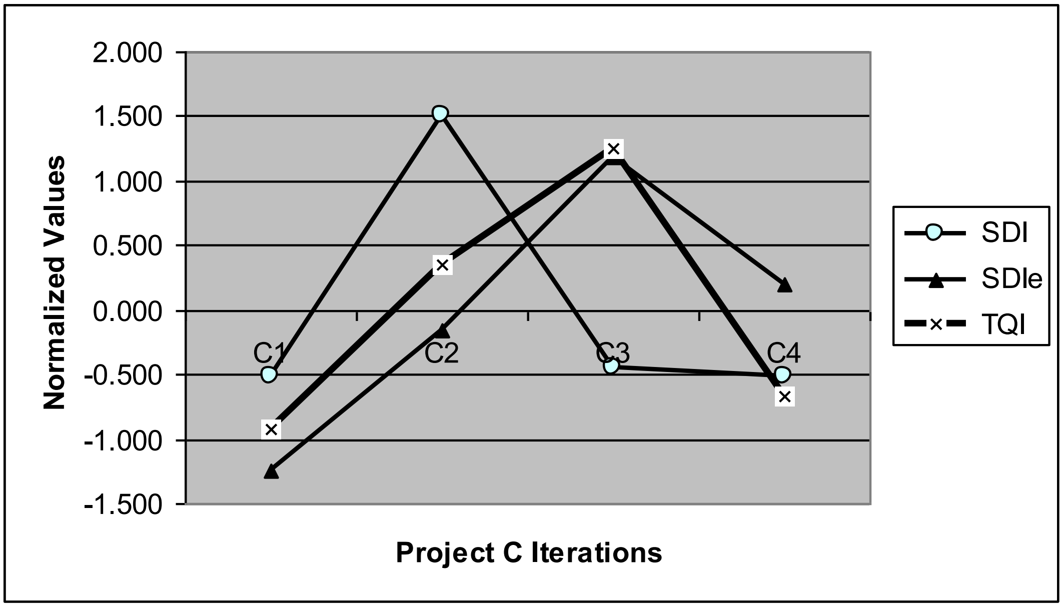


**Figure 4.11: Project C – TQI vs. Stability**

A comparison of the graphs for Project C in Figure 4.11 above demonstrates that the graph of TQI closely follows the $SDI_e$ metric graph. From another qualitative examination, this programming team also followed the extreme programming model and delivered a successful project.

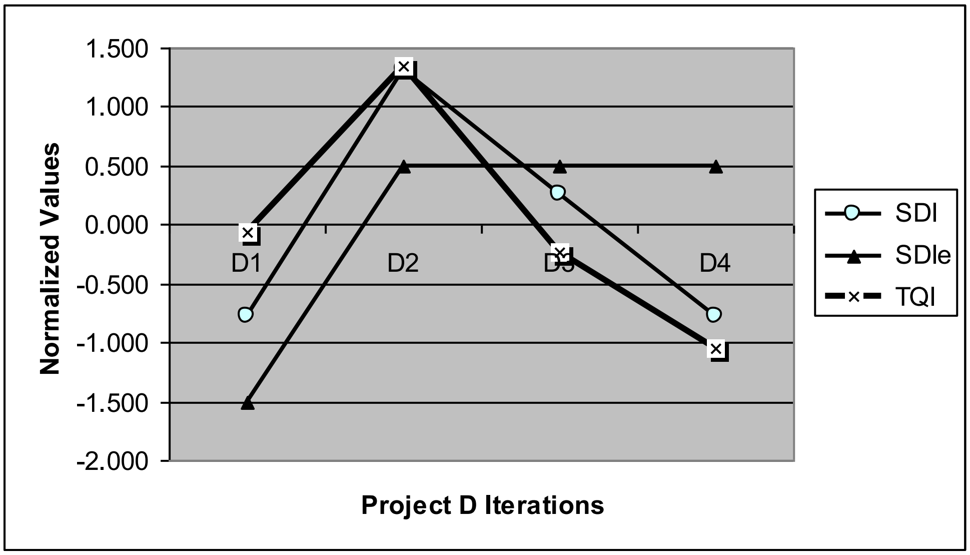


**Figure 4.12: Project D – TQI vs. Stability**

As were all the other projects described previously, Project D was also assigned to be implemented using extreme programming. Upon delivery, it was discovered that apparently the students on the programming team failed to grasp the concept in which increasing functionality is delivered at each iteration. After the first iteration the team removed all the classes and delivered a totally different set of classes. On the third iteration the original functionality from the first iteration was reintroduced. This caused some very interesting values for $SDI_e$. We might suppose that possibly this led to the fact that the $SDI_e$ metric did not closely follow the TQI graph. The SDI metric defined in Li et al. does in fact closely approximate the TQI graph [Li et al., 2000].

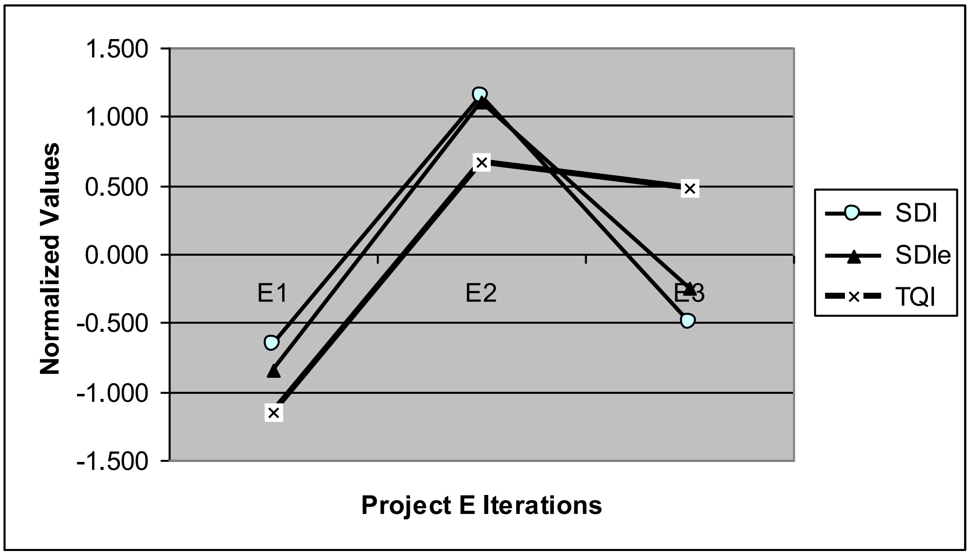


**Figure 4.13: Project E – TQI vs. Stability**

This fifth software engineering project, Project E, implements the same application as did Project A. Because the project team had some difficulty with their second iteration, there are only three iterations. The iteration listed as E2 above was in fact delivered when the other programming teams were delivering their third iteration. This might explain the steep increase in the graphs of the stability metrics or the fact that the graphs of Project A and Project E are so different. We again notice the striking similarities in the graphs of both the stability metrics with the graph of TQI.

Table 4.7 demonstrates the correlation of the stability metrics, SDI and $SDI_e$, with the Bansiya's Total Quality Index, TQI. There is a correlation between each of the stability metrics and TQI with a somewhat stronger correlation with SDI.

Table 4.7: Spearman's Correlation of Stability Metrics and TQI

| | SDI | SDI$_e$ |
|---|---|---|
| Correlation | .618 | .474 |
| P-value | .002 | .022 |
| Significant at $\alpha = 0.05$? | Yes | Yes |
| Significant at $\alpha = 0.01$? | Yes | No |

## 4.3 Discussion

We investigated the new SDI$_e$ metric as proposed by Olague et al. through the analysis of five highly iterative projects. The data presented in this study serve to further demonstrate the SDI$_e$ metric is a measure which can be used to characterize system stability.

Probably the most important result of this study is that the classes-changed category is a very important category to be used in measuring stability. Because the SDI$_e$ metric does not use names changed, based on our experience in this project, it is certainly easier to automate than the original SDI metric, since determining whether a class name has been changed requires a human investigation of the source code. It is true that names changed could be a measure of stability that is missed by the SDI$_e$ metric. It should be noted that while the SDI metric may be determined during design and hence could serve as a measure of design stability, the SDI$_e$ metric requires metrics be collected from the source code.

The statistical analysis of all five projects was not conclusive due to the lack of significant results; so it is unclear overall whether the SDI$_e$ metric measures something different to what is measured by the Chidamber and Kemerer metrics [Chidamber and

Kemerer, 1994]. However, based on the results one wonders in particular whether the Sum of LCOM, Sum of CBO, and Sum of RFC metrics might have some overlap with SDI$_e$. Additional study is needed. We did discover that there are exceptional circumstances which would give a misleading value for the SDI$_e$ metric as demonstrated within Project D. (In this case, not correctly following the given project development model was the cause.)

In SDI$_e$ the complete Cantata metrics set, as provided by the commercial Cantata++ tool, was used to determine whether or not a class changed. In our study, this category of "classes changed" turned out to be important. However, some metrics in the Cantata set are not as useful as some other metrics for determining the changes in a class. For example, the ACC_EXTMETH (accessible external methods) for a class might change due to method changes in other classes while the actual code for the class is unchanged. Other metrics such as the ACC_USERFUNC (accessible non-member functions) and the ACC_SYSFUNC (accessible non-member functions declared in system header file) can also demonstrate changed values without the actual code of the class changing. Another point of difference arises from the use of library classes and how their use is reflected in the inheritance metrics. Also, using changes in the arbitrary set of metrics provided by a particular commercial metrics tool is far too connected with a proprietary tool to be repeatable; the company might change the particular set of metrics without warning, and thus metrics analysis using SDI$_e$ might not be repeatable on future metrics tool releases. Thus, a better SDI$_e$ would be measured using a particular set of metrics, with clearly defined definitions. This would reduce dependence on a single tool.  Exactly which set of metrics is best requires further study.

A different method to measure classes changed would involve examining actual changes in executable code.  This is work intensive, but might have greater accuracy. This is similar to work previously performed by Alshayeb and Li [Alshayeb and Li, 2005; Alshayeb and Li, 2003].  This is possible future research.

# CHAPTER 5

# THE RELATIONSHIP OF METRICS AND FAULTS

Although stability is used to indicate maturity and is usually calculated as the number of modifications to the source code, some authors [Boudnik, 2008: Repenci, 2008] define stability in terms of fault density. The argument would be, the fewer the faults, the more stable the software (although fault density is more commonly considered a measure of software reliability).

Therefore, for completeness in our stability study, we examine the ability of various software metrics to predict faults. We considered the C&K metrics as fault predictors as they are probably the best known object-oriented metrics. Empirical validation of this type has been performed on the class level by several researchers including Olague et al. and El Emam et al. for industrial software [Olague et al., 2007; El Emam et al., 2001].

## 5.1 Spearman's Correlation

To determine whether any one of the six C&K metrics might be good individual fault indicators, we calculated the Spearman's correlation between the fault number and each of the metrics. The results are given in the following Table 5.1.

**Table 5.1: Spearman’s Correlation of C&K vs. Faults**

| Metric | CBO | DIT | LCOM | NOC | RFC | WMC |
|---|---|---|---|---|---|---|
| Spearman’s rho | 0.152 | -0.015 | 0.016 | -0.148 | 0.232 | 0.240 |
| Significance | 0.534 | 0.952 | 0.950 | 0.546 | 0.338 | 0.322 |

These values were calculated using SPSS$^{®}$ 15.0. For a two-tailed test, these values did not indicate that any of the individual metrics would be a good fault predictor since none of the results were significant. This led us to consider linear combinations of the metrics as possible predictors of fault-proneness.

### 5.2 Collinearity Analysis

We then performed a collinearity analysis of the six C&K metrics was performed to realize which metrics could best be used within a model for predicting fault-proneness. The variance inflation factor (VIF) is the reciprocal of the tolerance which indicates the variance percentage unaccounted for by other independent variables within the regression equation [Olague et al., 2007]. The VIF value may be used as a threshold for determining whether multicollinearity exists [Mansfield and Helms, 1982]. The condition number, the ratio of the square root of the largest eigenvalue to all the others, is also an indicator that multicollinearity may be a problem [Olague et al., 2007]. A rule of thumb used for linear regression is a threshold of 10 for VIF [Olague et al., 2007]. All six metrics were first analyzed for our first model, Model 1, with the results of the VIF and condition number given in Appendix H, Table H.1.

The large values of the VIF for the CBO, RFC, and WMC along with the large value of the WMC condition number indicate that collinearity may be a problem. We removed the RFC metric to form Model 2 with the remaining five metrics (CBO, DIT, LCOM, NOC, and WMC). The results of the analysis are given in Appendix H, Table H.2.

The resulting VIF values are substantially reduced and the condition numbers are all in a good range as well with possibly the exception of the WMC metric. We then performed a linear regression of Model 2 with faults. The results of the linear regression are given in Table H.3, Appendix H.

The results in Table H.3 led us to look for other models since this model was not significant. We considered another group of the C&K metrics which we labeled Model 3 in which we replaced the RFC metric and removed the WMC metric since it was the metric with the largest condition number. The results of the collinearity analysis of Model 3 are given in Appendix H, Table H.4.

Although the VIF values are in an acceptable range, the condition number for RFC is questionable. However, we performed linear regression of Model 3 versus faults to determine whether this model is significant. Table H.5, Appendix H, gives the results of the linear regression using Model 3.

Model 3's results from the linear regression versus faults were no more promising than those of Model 2. Again, the results were not significant. We then considered removing two of the metrics (RFC and WMC) which had the two largest condition numbers in our original Model 1.

The analysis of the collinearity of Model 4 given in Appendix H, Table H.6, gives both acceptable VIF and Condition numbers for the four metrics of CBO, DIT, LCOM, and NOC. We proceeded with a linear regression versus the fault values with the results given in Table H.7. The results for Model 4 also do not provide significant results.

### 5.3 Discussion

We continued considering other models of combinations of metrics in the manner demonstrated above but without any success. Thus, our attempt to discover a model to predict fault-proneness using the C&K metrics on our data was unsuccessful. These results were similar to those found by Olague et al. in 2007 [Olague et al., 2007]. However, other researchers have found successful C&K fault prediction models. Some of these fault prediction models were achieved using data mining techniques which are not sensitive to collinearity problems [Olague, 2006].

# CHAPTER 6

# THE RELATIONSHIPS OF FAULTS, REUSABILITY AND REFACTORING

Refactoring is an integral aspect of agile software development [Fowler, 1999]. For this reason, the impact of refactoring on reusability is an important part of our research. We used data gathered from five different agile projects to consider the relationship of faults, reusability ratings, and refactoring numbers. Four of the projects had four iterations each and one project had only three iterations. All of these project teams collected fault and refactoring data during software development. The reusability values which were used were obtained by averaging the expert opinions given by the evaluators for each class. This kind of averaging is acceptable due to the overall good $r_{wg}$ values per class. See our earlier discussion in Chapter 3.

We began our investigation of the relationships of faults, reusability and refactoring by considering the graphs of each of the three values. In order to consider these values on the same graph, we normalized each set of values: faults, reusability, and refactoring. Normalization or standardization was accomplished by finding the mean and standard deviation of each of the concepts (faults, reusability or refactoring) and then subtracting the corresponding mean and dividing by the standard deviation.

These calculations were performed using the STANDARDIZE function in Excel. The original values along with the normalized values are given in Table 6.1. We graphed both the original and the normalized values.

**Table 6.1: Faults, Refactoring and Reusability**

| | Original Values | | | Normalized Values | | |
|---|---|---|---|---|---|---|
| Iteration | fault | refactoring | Reusability | faults | refactoring | reusability |
| A1 | 31 | 0 | 3.286 | 1.115 | -1.174 | 1.196 |
| A2 | 12 | 10 | 2.000 | -1.306 | 0.168 | -0.939 |
| A3 | 22 | 7 | 2.143 | -0.032 | -0.235 | -0.701 |
| A4 | 24 | 18 | 2.833 | 0.223 | 1.241 | 0.445 |
| | | | | | | |
| Iteration | fault | refactoring | Reusability | faults | refactoring | reusability |
| B1 | 50 | 0 | 2.714 | 1.474 | -1.098 | -0.048 |
| B2 | 10 | 23 | 2.571 | -0.235 | 0.433 | -0.358 |
| B3 | 2 | 34 | 2.286 | -0.577 | 1.165 | -0.977 |
| B4 | 0 | 9 | 3.375 | -0.662 | -0.499 | 1.384 |
| | | | | | | |
| Iteration | fault | refactoring | Reusability | faults | refactoring | reusability |
| C1 | 177 | 86 | 3.286 | 0.825 | -0.759 | 1.201 |
| C2 | 151 | 230 | 2.857 | 0.383 | -0.047 | -0.240 |
| C3 | 143 | 528 | 2.571 | 0.247 | 1.427 | -1.201 |
| C4 | 43 | 114 | 3.000 | -1.454 | -0.621 | 0.240 |
| | | | | | | |
| Iteration | fault | refactoring | reusability | faults | refactoring | reusability |
| D1 | 0 | 0 | 2.429 | -0.500 | -0.506 | -0.369 |
| D2 | 10 | 120 | 3.857 | 1.500 | 1.500 | 1.477 |
| D3 | 0 | 0 | 2.429 | -0.500 | -0.506 | -0.369 |
| D4 | 0 | 1 | 2.143 | -0.500 | -0.489 | -0.739 |
| | | | | | | |
| Iteration | fault | refactoring | reusability | faults | refactoring | reusability |
| E1 | 22 | 35 | 3.000 | -0.581 | 0.226 | 1.121 |
| E2 | 264 | 52 | 2.429 | 1.155 | 0.867 | -0.801 |
| E3 | 23 | 0 | 2.571 | -0.574 | -1.094 | -0.320 |

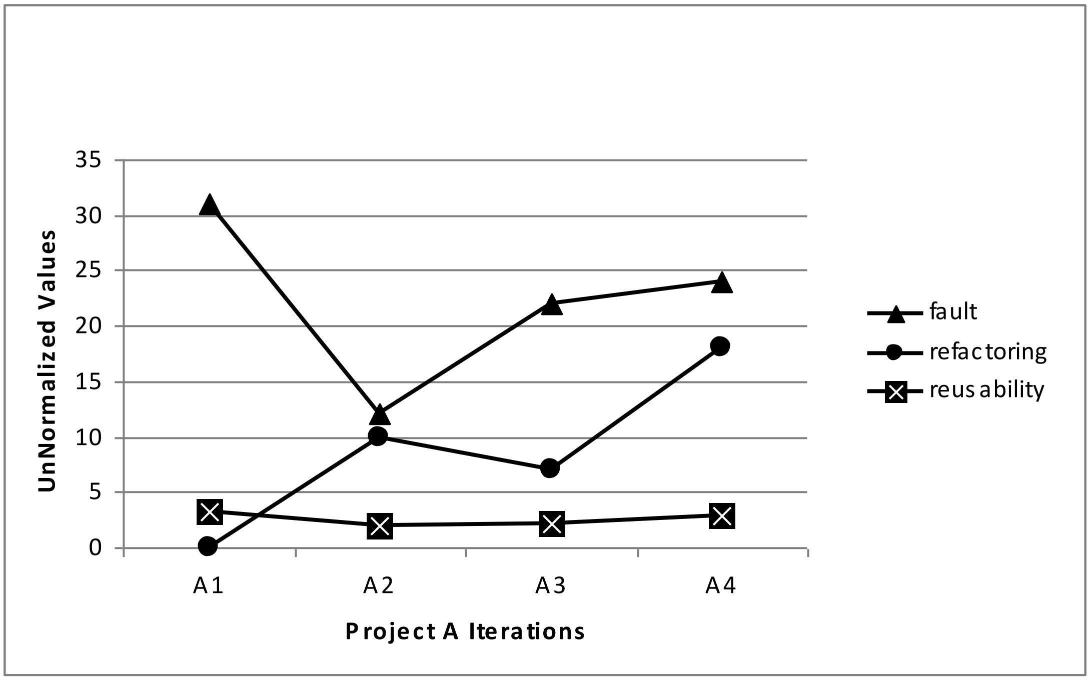


**Figure 6.1: Project A Unnormalized Comparison**

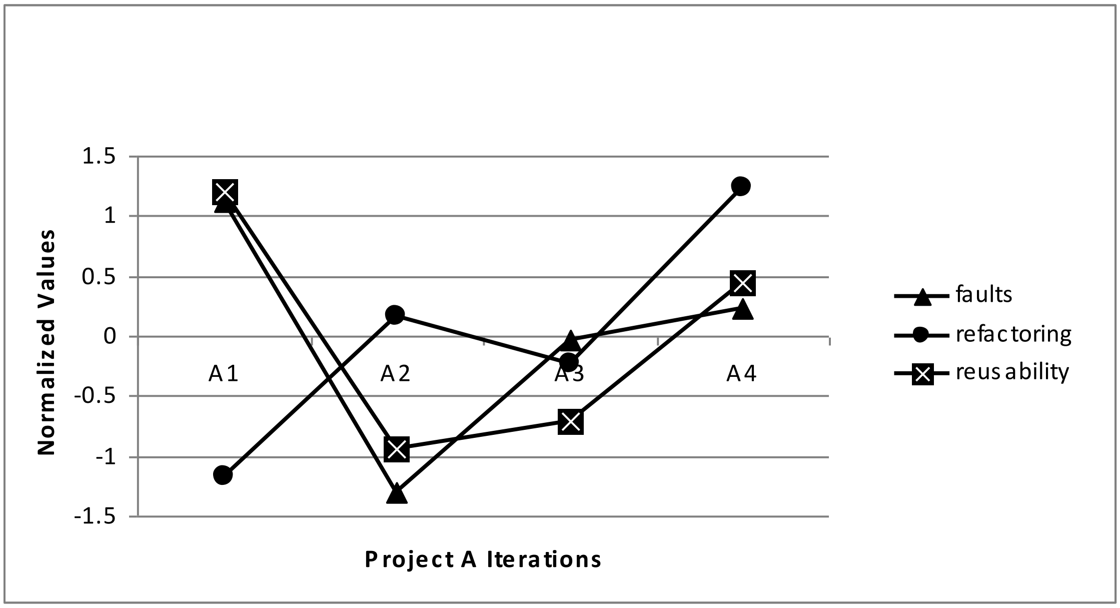


**Figure 6.2: Project A Normalized Comparison**

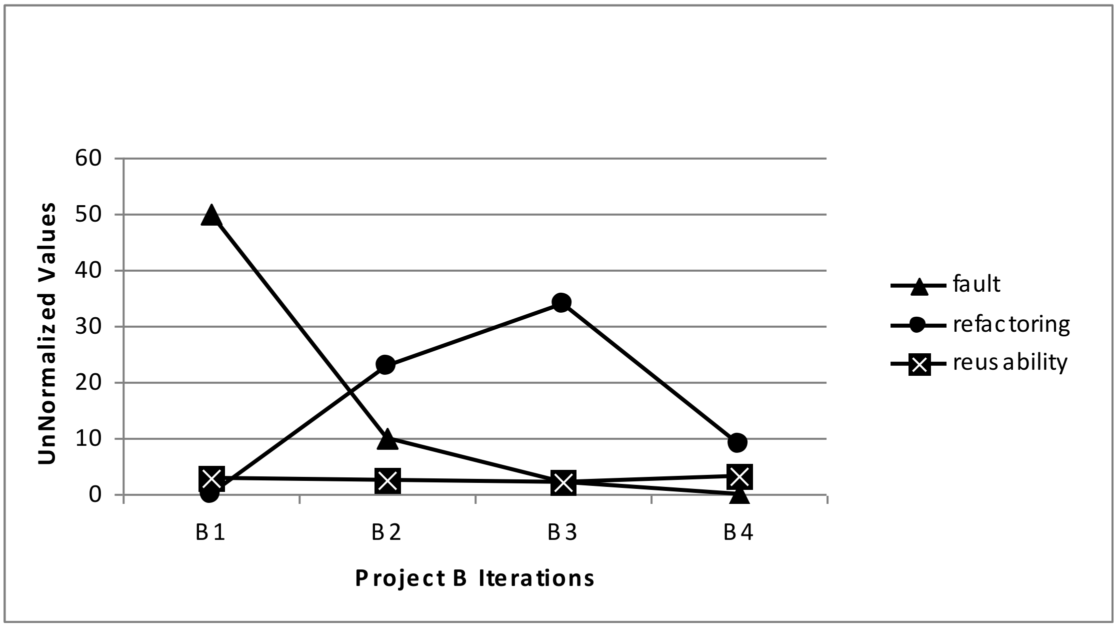


Figure 6.3: Project B Unnormalized Comparison

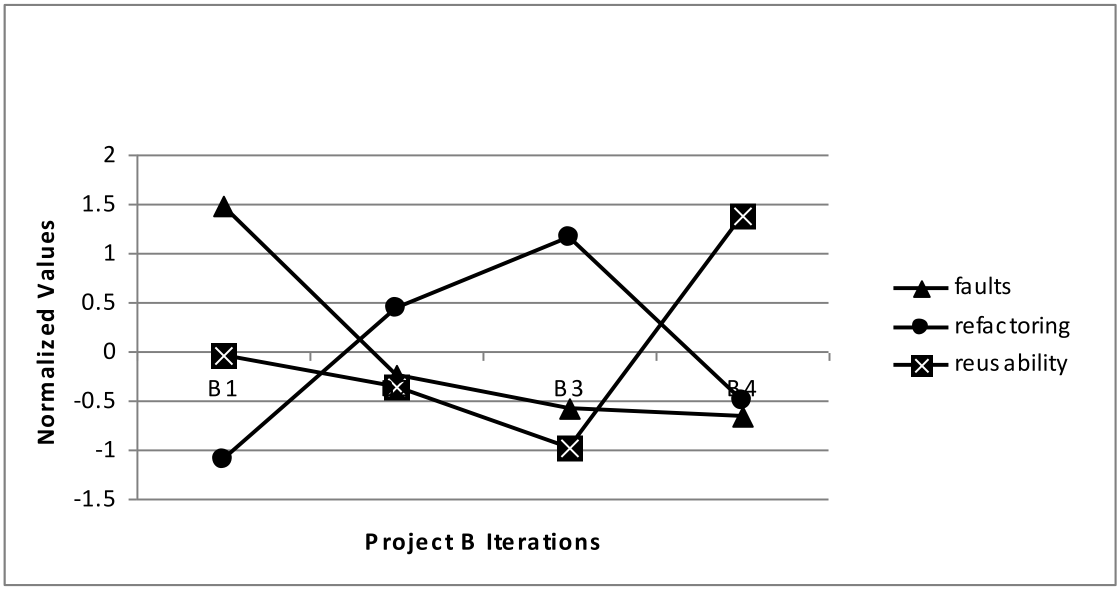


Figure 6.4: Project B  Normalized Comparison

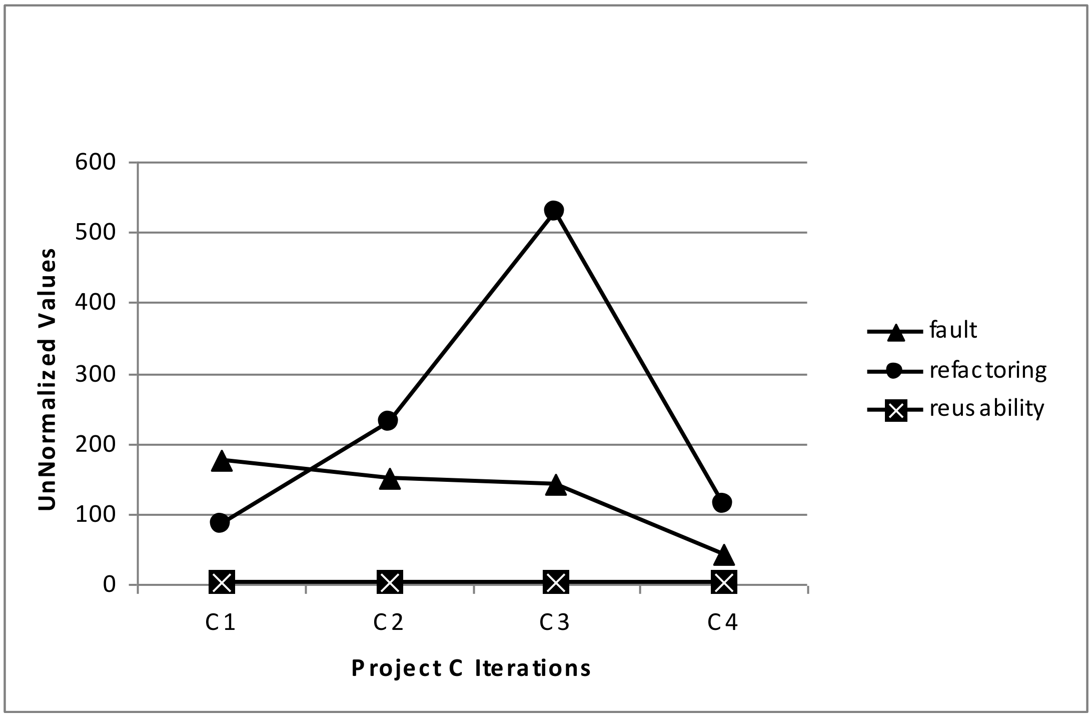


**Figure 6.5: Project C Unnormalized Comparison**

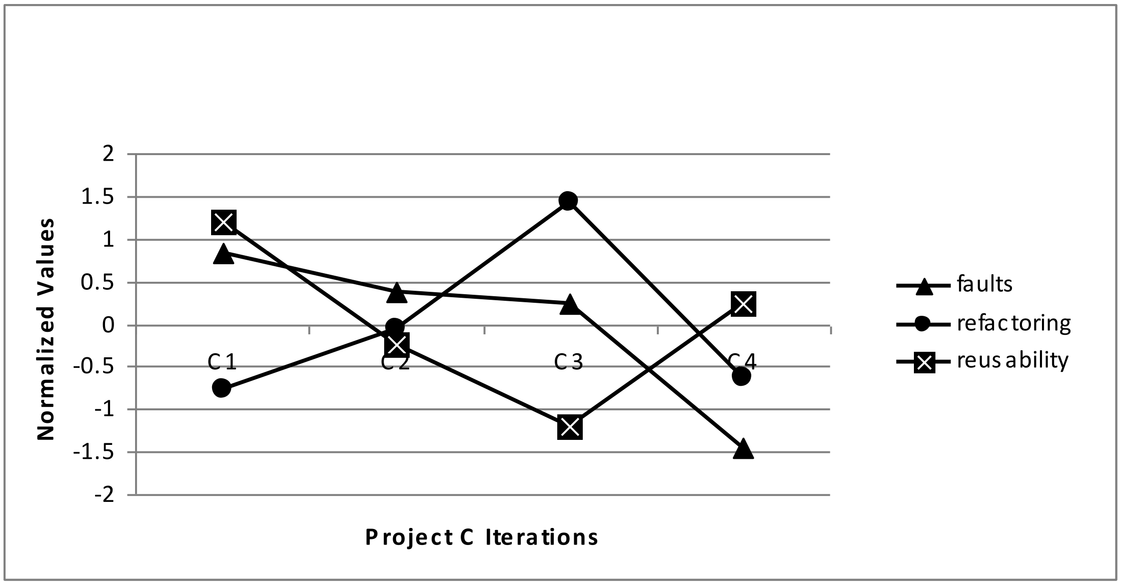


**Figure 6.6: Project C Normalized Comparison**

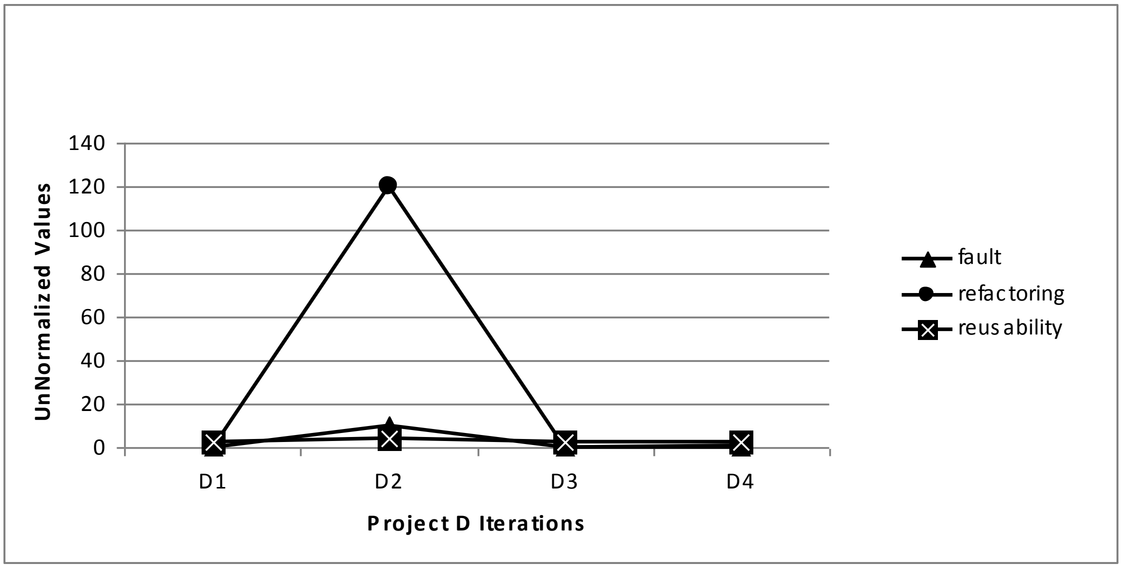


**Figure 6.7: Project D Unnormalized Comparison**

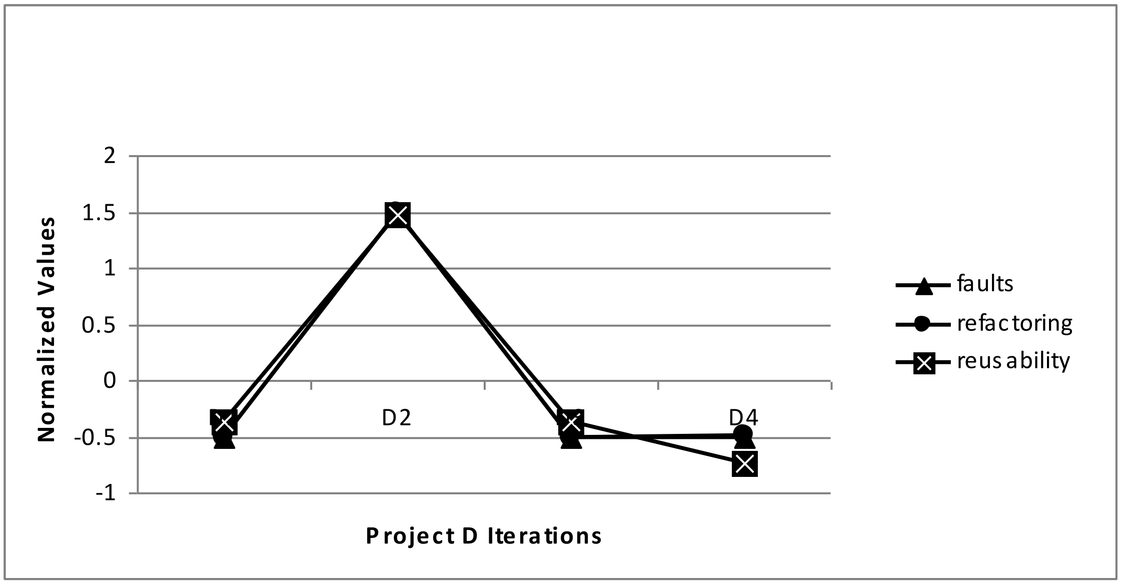


**Figure 6.8: Project D Normalized Comparison**

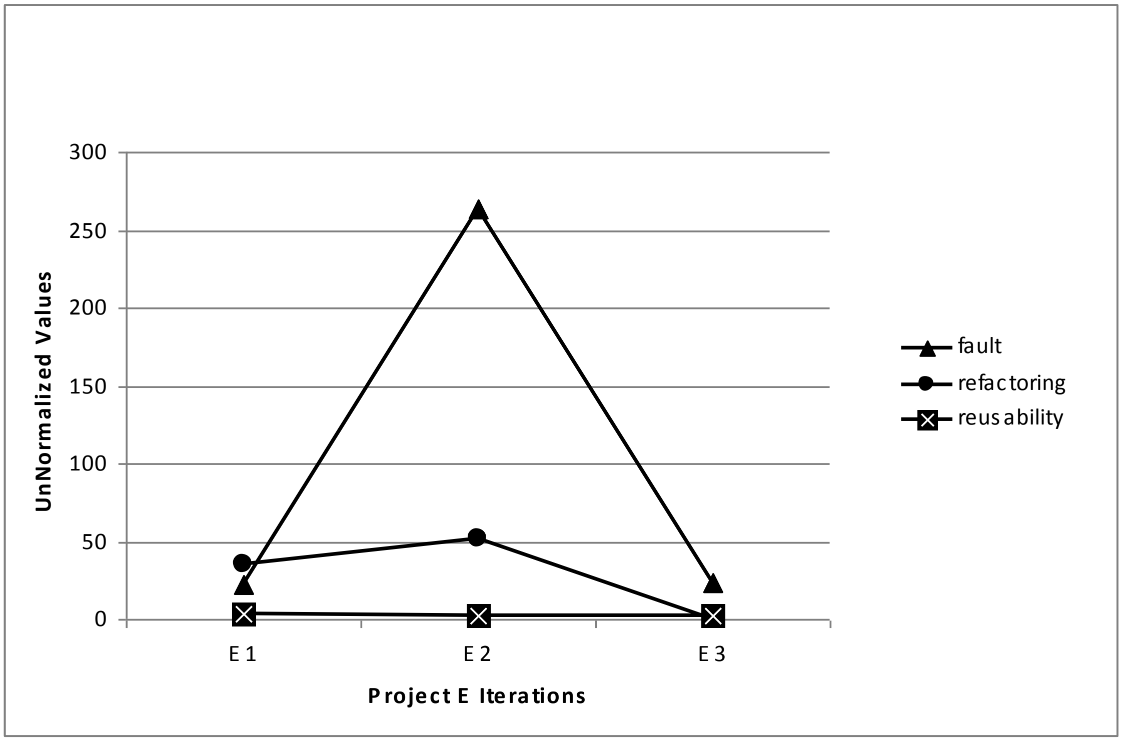


Figure 6.9: Project E Unnormalized Comparison

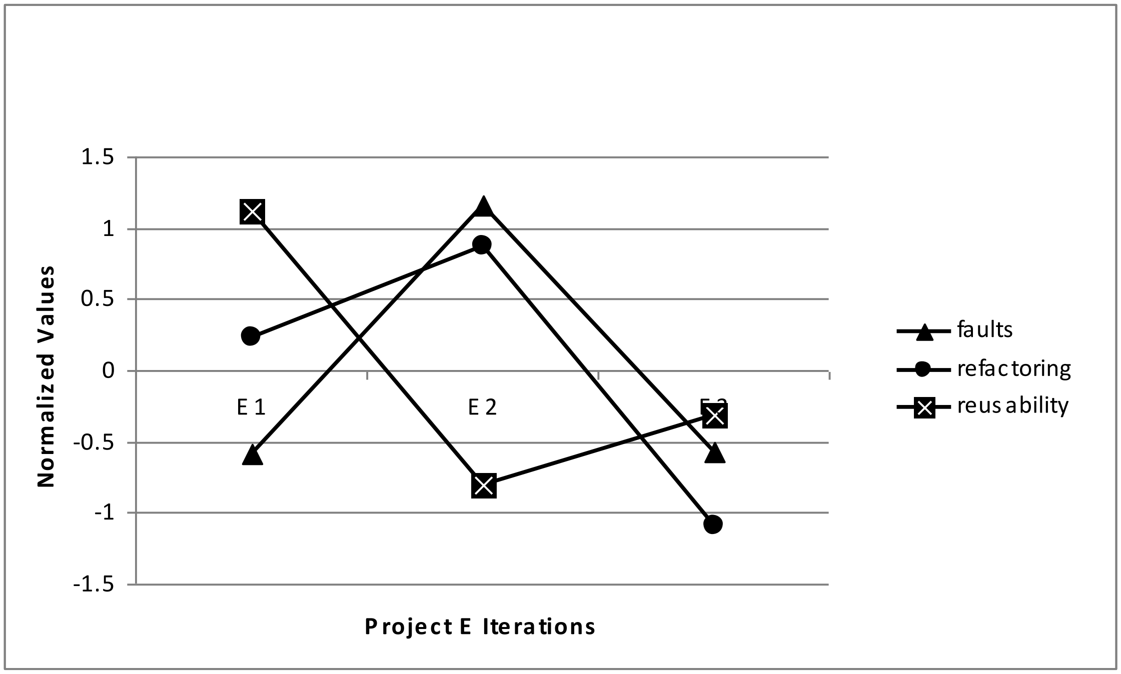


Figure 6.10: Project E Normalized Comparison

The unnormalized graphs in Figures 6.1, 6.3, 6.5, 6.7, and 6.9 were difficult to interpret due to the great differences in the scales for the three concepts of faults, refactoring, and reusability. We were unable to see the actual trends of the reusability graphs because of their relative smallness compared to refactoring values and faults. When we normalized the values which placed them on the same relative scale, we were able to get a better representation of their relationships.

Figures 6.2, 6.4, 6.6, 6.8, and 6.10 tend to indicate that some of the refactorings had as their purpose the repair of faults and did not make the code more reusable as viewed by the evaluators (as indicated by a reduction in the reusability rating). We infer this from an increase in refactorings resulting in an increase in the number of faults repaired at certain iterations. At other times the refactorings did produce a higher reusability rating from the experts. This may have been a result of the unfamiliarity of the concept of refactoring to student programmers and the manual implementation of refactoring techniques.

We further investigated the possible relationships of these values by performing a Spearman's rank correlation on pairs of values. The fault values, refactoring values and reusability averages were used for the nineteen agile iterations. The results of the correlations are given in Table 6.2.

**Table 6.2: Spearman’s Rank Correlation of Faults, Refactoring and Reusability**

| Faults vs. Refactoring | |
|---|---|
| Correlation Coefficient | Significance (2-tailed) |
| 0.513 | 0.025 |
| Faults vs. Reusability | |
| Correlation Coefficient | Significance (2-tailed) |
| 0.265 | 0.273 |
| Refactoring vs. Reusability | |
| Correlation Coefficient | Significance (2-tailed) |
| 0.350 | 0.142 |

The results of the correlation analysis demonstrate that the faults and refactoring values correlated with a significance of 0.025, which is considered a moderately strong correlation [Cohen, 1990]. However, the faults and reusability did not correlate and neither did the refactoring and reusability ratings. Although we might expect that the refactoring values should correlate with the reusability values due to the emphasis on refactoring to make the code simple which would supposedly result in more reusable code, we did not find this in our data. The novelty of the refactoring process and the necessity of performing refactorings manually might partially explain why refactoring correlated with faults and did not correlate with reusability. We did not expect faults to correlate with reusability which is what we found with our data.

## 6.1 Discussion

In summary, this study showed that refactorings indicated faults but not reusability, which was surprising. Possibly the use of refactoring tools would have removed the connection of refactoring with faults. This could be an area of future research. The lack of a relationship between refactoring and reusability tends to indicate that reusable software is not a typical outcome of the agile software development process.

# CHAPTER 7

# COMPARISON OF REUSABILITY OF SOFTWARE DEVELOPED USING TRADITIONAL PLAN-BASED METHODS WITH SOFTWARE DEVELOPED USING AGILE METHODS

The popularity of the agile software development paradigm in the literature and reportedly in practice led us to consider whether or not the software developed using agile methods is more reusable than the software developed using a traditional plan-based method. As we mentioned in the introduction, some authors have claimed that developing reusable software within an agile paradigm is quite achievable [Heinecke et al., 2003]. Some authors go further and imply that the intrinsic characteristics of the agile paradigm (primarily the emphasis on simple and understandable software) tend to result in reusable software [Knoernschild, 2006]. However, the agile paradigm's emphasis on simplicity of software violates a widely held belief: that software must be made as generic as possible in order to be reused in many different environments [Baum and Becker, 2000].

In this portion of our study, we compared human evaluators' reusability ratings of software developed using an agile method to software developed using a plan-based method. This consisted of two separate experiments, a paired experiment and an unpaired experiment. These experiments were performed on a per-class level. That is, the reusability of the classes was examined, not the (separately rated) reusability of the packages.

The paired experiment examined five agilely developed projects and five traditionally developed projects consisting of a total of fifty six classes. The unpaired experiment examined five traditionally developed projects and five agilely developed projects with a total of ninety five classes.

In the "unpaired" grouping, we specifically did not want evaluators of an agile project to be the same evaluators for the non-agile version of the project. The purpose here was to minimize any bias that resulted from the evaluator having already seen another version of the same project. In the "paired" grouping, we gave the agile and non-agile versions of the same project to the same evaluators. The purpose here was to minimize variance resulting from different evaluators with different opinions.

- For each experiment, the hypothesis to be tested was as follows:
  - ❑ $H_0$: There is no significant difference in the reusability of the software developed using highly iterative methods from those developed using traditional plan-based methods.
  - ❑ $H_1$: There is a significant difference in the reusability of the software developed using highly iterative methods from that developed using traditional plan-based methods.

For each class in the unpaired experiment, there were seven expert ratings of reusability. We were able to compute the average of reusability for each class over these seven experts since the $r_{WG(J)}$ interrater reliability values were acceptable per class (see Chapter 3 for a further discussion). Using the Kolmogorov-Smirnoff test, we determined the data in both the paired and unpaired experiments was normally distributed.

Each class can be considered to be independent of each other class in the software. This should be true since all the software was developed using the object-oriented paradigm. The object-oriented paradigm specifies that encapsulation, or information hiding, is the primary objective when defining each class [Snyder, 1986]. Encapsulation means that interdependencies between classes are kept to an absolute minimum. Thus, the reusability ratings of each class can be considered independent.

Since the data was normally distributed, and the classes are independent, it was legitimate to compare the mean of the reusability ratings of the agilely-developed classes to the mean of the classes developed using a plan-based method using a t-test—the statistical assumptions behind the t-test were met.

We began the study of reusability by taking the average of the reusability ratings for each of the classes within the paired and unpaired data and graphing a box plot for each grouping. The box plot supplied initial data analysis and summary of the data. The box begins at the twenty-fifth percentile and stretches to the seventy-fifth percentile. The median or fiftieth percentile is marked with a line within the box. Any outliers are indicated as dots outside the box.

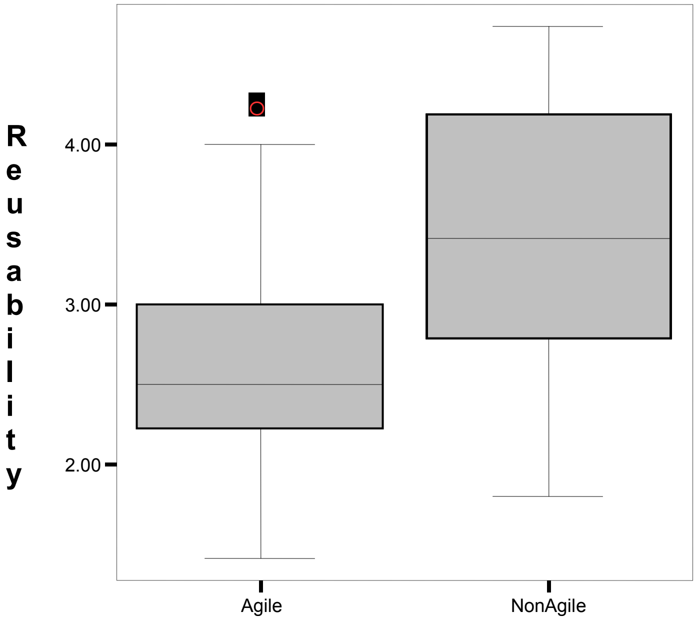


**Figure 7.1: Paired Evaluations Comparison**

Figure 7.1 represents the box plot for the paired data. The median of the traditional (nonagile) class reusability averages is substantially higher than the agile median. There was only a small amount of overlap of the two boxes. This tends to indicate that the medians of the agile and nonagile software are different. Since the nonagile is higher than the agile, it tends to indicate that the reusability of nonagile software is higher than that of agile software. Next, we examined the unpaired data. The results of this analysis are given in Figure 7.2.

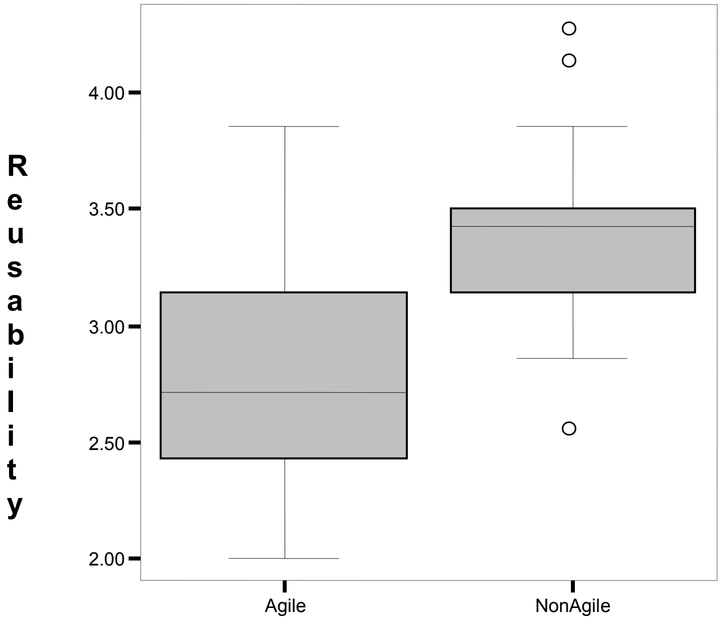


**Figure 7.2: Unpaired Evaluations Comparison**

The box plot for the unpaired evaluations in Figure 7.2 above demonstrated an even greater difference in the reusability ratings of the traditional (nonagile) classes as compared to the agile classes. The entire box for the nonagile classes is above the box for the agile classes. Thus, this tends to indicate: first, that the medians are different between the two methods, and second, that nonagile software is more reusable that agile software.

Therefore, based on the box plots, we reject the null hypothesis and accept the alternate hypothesis that there is a significant difference in the reusability of the software developed using highly iterative methods from that developed using traditional plan-based methods.

We then performed a t-test to test for the equality of the means of the reusability ratings of the traditional (nonagile) and agile classes. We performed separate t-tests on the two groups of data: paired and unpaired. Table 7.1 gives the results of this statistical analysis (performed using SPSS® 15.0).

**Table 7.1: T-Test Values for Testing Reusability**

| t-Test for Equality of Means | | | | | |
|---|---|---|---|---|---|
| Reusability | NonAgile Mean | Agile Mean | t | df | Sig (2-tailed) |
| Paired | 3.4165 | 2.5966 | 4.873 | 80 | <0.001 |
| Unpaired | 3.3393 | 2.7868 | 5.685 | 93 | <0.001 |

Based on the statistical analysis, we reject the null hypothesis that there is no difference in the reusability of agilely developed software and plan-based developed software and accept the alternative hypothesis that there is a difference in the means. The means for the agilely developed software were lower in both groupings.

## 7.1 Discussion

Both box plots and t-tests indicated that the reusability of nonagile software is higher than the reusability of agile software.

# CHAPTER 8

# CONCLUSIONS

The data gathered and analyzed within this study supported the following conclusions within the areas of stability, fault prediction, fault, reusability and refactoring relationships, and the reusability of agile software.

## 8.1 Stability

The results of this study validated the claim that the $\text{SDI}_e$ metric is a measure which can be used to characterize system stability [Olague et al., 2006]. An important result was that the classes changed category within the calculation of the $\text{SDI}_e$ metric is a very important category to be used in measuring stability. The method of determining whether a class changed by using whether the metric values changed for calculating the $\text{SDI}_e$ value may not be the best method. This is due to the inclusion of metrics in the complete Cantata++ metrics set (used in the $\text{SDI}_e$ calculation) which may not be good metrics to measure class change. For example, the ACC_EXTMETH (accessible external methods) for a class is an example of a metric value which might change due to method changes in other classes while the actual code for the class is not changed.

Utilizing an arbitrary commercial metrics tool and the corresponding set of metrics gathered by the tool for determining whether a class changed is far too connected with a proprietary tool to be repeatable; the company might change the particular set of metrics without warning, and thus metrics analysis using $SDI_e$ might not be repeatable on future metrics tool releases [Roden et al., Nov. 2007].

The collection of metrics using the source code of the project for determining the classes changed for the $SDI_e$ metric value for the project also made it less of a design stability metric than SDI which does not require that its inputs be gathered from the source code itself. However, for SDI, a manual inspection of a design or the source code was necessary to determine if a class name was changed.

There was a correlation between the Total Quality Index of Bansiya's quality model and both stability metrics SDI and $SDI_e$. There was a somewhat stronger relationship between SDI and TQI. This led us to propose that TQI, whose value could be easier to automate with less human intervention, could be used for indicating stability. There was also an apparent stronger relationship of the TQI value with the stability metrics on those projects which adhered more closely to the extreme programming practices as determined by a subjective evaluation by the instructor. Thus, potentially the TQI value compared with the stability metrics might serve as an indicator to management as to whether a development team is following the extreme programming practices.

### 8.2 C&K Metrics and Fault-Proneness

The results of this study did not support the development of a model based on the C&K metrics to predict fault-proneness . This was in opposition to the results reached in some other similar studies but is in agreement with others [Olague et al., 2007; El Emam et al., 2001; Tang et al., 1999]. El Emam et al. stated that class size had a confounding effect on the validity of using the C&K metrics to predict fault-proneness. Because our classes came from student projects which are all of a relatively small size compared to much industrial software, we might conclude that this was where our difficulty arose in attempting to development a model to predict faults using the C&K metrics.

### 8.3 Faults, Reusability and Refactoring

We first inspected the relationship of the faults, reusability, and refactoring through graphical methods. The comparison of the three values highlighted the fact the refactorings performed on an iteration did not always result in more reusable code as evaluated by the experts. There were instances when refactorings resulted in more reusable code but other times when the code decreased in reusability and increased in faults repaired. Apparently some refactorings were just used to repair faults and not to improve the quality of the design. This might partially be explained by using student projects for this study in which the refactoring was performed manually—this might introduce more errors than if automated refactoring were performed. Secondly, because refactoring was a new topic introduced to the students before they began their assignment, their expertise in refactoring was limited and opportunities for refactoring might have been overlooked.

Statistical analysis indicated a positive correlation between faults and refactoring, as we might expect due to the opportunities for introducing both syntactic and logical errors when attempting to refactor manually.  Although we expected that there might be a correlation between refactoring and reusability, there was none for our data.  There was also no correlation between faults and reusability which was more expected.

## 8.4  Reusability of Traditionally Developed Software Compared to Agile Software

Once again we first used a graphical representation to compare the reusability ratings of the classes of traditionally developed software to the reusability ratings of classes developed using agile methods.  Data was graphed using the paired data evaluations and the unpaired evaluations and gave us our first indication that the traditional projects possessed a higher reusability rating.  We then performed statistical analysis using a t-test.  The results of the t-test supported what we had seen in the graphs, that is: the software developed using traditional plan-based methods had a higher reusability rating than did the software developed using agile methods.

# CHAPTER 9

# FUTURE RESEARCH

There are several opportunities for study. Some future work could be done using the data we collected for this study. Other future work could be expanded to other sets of data. First, methods for classifying how a class has changed that could be used in calculating the $SDI_e$ metric need to be considered. Possible options include using actual code changed or using a specific set of recognized metrics changed. Secondly, further validation of the relationship of the stability metrics with the TQI value is needed.

The evaluator data collected for our study using the package questionnaire also included responses on the package level for hierarchy, size, flexibility, understandability, functionality, extendibility, effectiveness and total quality. Because we also have fault and refactoring data available for these packages, we might look for relationships between faults and refactoring and the other various evaluator quality ratings. We might also consider the relationships of these other evaluator quality ratings to how the projects were developed, whether by traditional methods or agile methods. The relationship of the QMOOD metrics to the expert total quality rating from the package questionnaire is another possible area of research [Bansiya and Davis, 2002].

Similarly, the class questionnaire resulted in responses for cohesion, coupling, modularity, interface, documentation, size, complexity, simplicity, encapsulation, composition, inheritance, abstraction, and polymorphism. The relationship of these responses could be analyzed relative to how the class was developed. An analysis of possible interrelations of some of the responses might also be explored.

Data was also gathered by a demographics questionnaire (a unique evaluator code was employed to associate evaluations to demographics in order to insure anonymity). The demographic data was not used within our study but gives opportunity for consideration from several perspectives. Differences in the background of evaluators could be investigated in relation to how their background affected their various quality ratings.

Another avenue of study might be concerned with the type of application assigned and what effect it might have on the ratings. Because there were three similar in complexity projects assigned as both traditional and agile projects for the same time span, we might consider whether the type of project was a factor in the expert evaluations.

# APPENDICES

# APPENDIX A

# CLASS QUESTIONNAIRE

# Software Quality Questionnaire for Each Class in a Package

## Part I

**Directions:**

This questionnaire contains statements about code properties. Definition of code properties and criteria is given. Select the appropriate number that you think is most descriptive of the class

1. **Cohesion:** Assesses the relatedness of methods and attributes in class. Strong overlap in the method parameters and attribute types is an indication of strong cohesion**.**
    - How big is the class in terms of number of attributes and number of methods? (A large class is less likely to be cohesive)?
    - Are methods in the class using disjoint sets of attributes (do there exist methods that have no attributes in common with other methods -- this could be a hint that the class should be broken into two or more classes)?
    - Are the methods in the class closely related in functionality?

Now, using the criteria stated in the above question, **rate the class for cohesiveness.**

| | | | | |
|---|---|---|---|---|
| **5** | **4** | **3** | **2** | **1** |
| **Excellent** | **Good** | **Fair** | **Poor** | **Awful** |

2. **Coupling:** Defines the interdependency of an object on other objects in a design. It is measure of the number of other objects that would have to be accessed by an object in order for that object to function correctly.
    - Are class methods using global data?
    - Does the class have friend functions or classes?
    - How many class methods access the attributes of any other class not in the class' s direct hierarchy? (list of direct ancestor classes)
    - How many class methods access the methods of any other class not in the class' direct hierarchy? (list of direct ancestor classes)
    - How many class methods access any external-to-the-class (standalone) functions (except for methods of other classes)?
    - Are there any variable definitions, either in the class definition, or local to a member function, that uses another class as an abstract data type?

Now, using the criteria stated in the above question, and any other criteria you think is appropriate, **rate the class for coupling.**

| | | | | |
|---|---|---|---|---|
| **5** | **4** | **3** | **2** | **1** |
| **Excellent** | **Good** | **Fair** | **Poor** | **Awful** |

3. **Modularity:** Considering the cohesion and coupling and any other criteria that you think is appropiate , **rate the class for modularity**.

| 5 | 4 | 3 | 2 | 1 |
|---|---|---|---|---|
| **Excellent** | **Good** | **Fair** | **Poor** | **Awful** |

4. **Interface:** A count of number of public methods that are available as services to other classes. This is a measure of the services that a class provides.
    - How many public methods are there?
    - How many formal parameters, on the average, do the public method definitions have?
    - Are the public methods at the appropriate granularity level? That is, do they do too much, not enough, or too little for the functionality provided by the class? Should some of their required functionality be moved to an internal private method, then that method be called by the public method?
    - Are the public methods clean or easy to understand?

Now, using the criteria stated in the above question, and any other criteria you think is appropriate, **rate the class for interface**.

| 5 | 4 | 3 | 2 | 1 |
|---|---|---|---|---|
| **Excellent** | **Good** | **Fair** | **Poor** | **Awful** |

5. **Documentation**
    - How many comments are there in the class definition?
    - How many comments are there in each method, on the average?
    - What percentages of methods have any comments at all?
    - Are the comments in general well written, understandable, or meaningful?
    - Are the identifier names (class names, variable names, method names, etc.) well chosen, understandable, or meaningful?

Now, using the criteria stated in the above question, and any other criteria you think is appropriate, **rate the class for documentation.**

| 5 | 4 | 3 | 2 | 1 |
|---|---|---|---|---|
| **Excellent** | **Good** | **Fair** | **Poor** | **Awful** |

6. **Size:** This is the measure of how big is the class in terms of number of attributes methods etc.
    - How big is the class in terms of number of attributes?
    - How big is the class in terms of number of methods?
    - How many lines of executable semicolons are there in the class definition (ignoring comments, blank lines, etc.)?
    - How many executable semicolons are there, on the average, in the methods?

- How many formal parameters are there in the method definitions, on the average?

Now, using the criteria stated in the above questions, and any other criteria you think is appropriate, **rate the class for size**.

| 5 | 4 | 3 | 2 | 1 |
|---|---|---|---|---|
| **Very Large** | **Large** | **Medium** | **Small** | **Extra Small** |

7. **Complexity:** A measure of the degree of difficulty in understanding and comprehending the internal and external structure of classes and their relationships.
    - How complex is the code in the methods, on the average?

Now, using the criteria stated in the above question, and any other criteria you think is appropriate, rate the class for **complexity**

| 5 | 4 | 3 | 2 | 1 |
|---|---|---|---|---|
| **Very Complex** | **Complex** | **Mostly Complex** | **Somewhat Complex** | **Simple** |

8. **Simplicity:** Considering the Size and complexity and any other criteria that you think is appropiate , **rate the class for simplicity**.

| 5 | 4 | 3 | 2 | 1 |
|---|---|---|---|---|
| **Excellent** | **Good** | **Fair** | **Poor** | **Awful** |

9. **Encapsulation:** Defined as the enclosing of data and behavior within a single construct.
    - How many attributes are declared in this class?
    - How many attributes are declared private in this class?

Now, using the criteria stated in the above question, and any other criteria you think is appropriate, **rate the class for encapsulation**

| 5 | 4 | 3 | 2 | 1 |
|---|---|---|---|---|
| **Excellent** | **Good** | **Fair** | **Poor** | **Awful** |

10. **Composition:** This is a measure of aggregation relationships in an object-oriented design.
    - How many user-defined classes are employed as data types in the class?

Now, using the criteria stated in the above question, and any other criteria you think is appropriate, **rate the class for composition**

| 5 | 4 | 3 | 2 | 1 |
|---|---|---|---|---|
| **Excellent** | **Good** | **Fair** | **Poor** | **Awful** |

**11. Inheritance:** This is a measure of the “is-a” relationship between classes.

- How many methods are accessible to this class?
- How many methods does this class inherit?

Now, using the criteria stated in the above question, and any other criteria you think is appropriate, **rate the class for inheritance**

| | | | | |
|---|---|---|---|---|
| **5** | **4** | **3** | **2** | **1** |
| **Excellent** | **Good** | **Fair** | **Poor** | **Awful** |

**12. Abstraction:** This is a measure of generalization – specialization aspect of the design.

- How many classes does this class inherit from? (That is the number of classes along all paths from the root classes to this class)

Now, using the criteria stated in the above question, and any other criteria that you think appropriate, **rate the class for abstraction**

| | | | | |
|---|---|---|---|---|
| **5** | **4** | **3** | **2** | **1** |
| **Excellent** | **Good** | **Fair** | **Poor** | **Awful** |

**13. Polymorphism:** It is the measure of services that are dynamically determined at run time in an object.

- How many methods exhibit polymorphic behavior?

Now, using the criteria stated in the above question, and any other criteria that you think appropriate, rate the class for **abstraction**

| | | | | |
|---|---|---|---|---|
| **5** | **4** | **3** | **2** | **1** |
| **Very Large** | **Large** | **Medium** | **Small** | **Extra Small** |

**<u>Part II</u>**

**Directions:**

Please rate the quality factor for the class

**1.** Using the criteria stated in the above questions, and any other criteria you think is appropriate, rate the class for **Reusability-in-the-Class:**

| | | | | |
|---|---|---|---|---|
| **5** | **4** | **3** | **2** | **1** |
| **Excellent** | **Good** | **Fair** | **Poor** | **Awful** |

# APPENDIX B

# PACKAGE QUESTIONNAIRE

# Software Quality Questionnaire for a Package

## Part I

**Directions:**

This questionnaire contains statements about code properties. Definition of code properties and criteria is given. Select the appropriate number that you think is most descriptive of the package.

1. **Hierarchy:** This is the count of the number of non-inherited classes than have children in a design**.**
    - How many class hierarchies are there in this package?

Now, using the criteria stated in the above question, and any other criteria that you think is important, **rate the class for hierarchy**

| 5 | 4 | 3 | 2 | 1 |
|---|---|---|---|---|
| **Very Large** | **Large** | **Medium** | **Small** | **Extra Small** |

2. **Package Size**: This is a measure of how big is the class in terms of number of classes.

    Now, using the criteria stated above, and any other criteria that you think is important**, rate the class for package size**

| 5 | 4 | 3 | 2 | 1 |
|---|---|---|---|---|
| **Very Large** | **Large** | **Medium** | **Small** | **Extra Small** |

## Part II

**Directions:**

Please rate the quality factor for the package

**2.** Based on your analysis of all the classes in the package and any other criteria you think is appropriate**, rate the package for Reusability-in-the-Package**

| 5 | 4 | 3 | 2 | 1 |
|---|---|---|---|---|
| **Excellent** | **Good** | **Fair** | **Poor** | **Awful** |

**3.** Based on your analysis of all the classes in the package and any other criteria you think is appropriate, **rate the package for flexibility.**

| 5 | 4 | 3 | 2 | 1 |
|---|---|---|---|---|
| **Excellent** | **Good** | **Fair** | **Poor** | **Awful** |

**4.** Based on your analysis of all the classes in the package and any other criteria you think is appropriate, **rate the package for understandability**.

| 5 | 4 | 3 | 2 | 1 |
|---|---|---|---|---|
| **Excellent** | **Good** | **Fair** | **Poor** | **Awful** |

**5.** Based on your analysis of all the classes in the package and any other criteria you think is appropriate, **rate the package for functionality.**

| | | | | |
|---|---|---|---|---|
| **5** | **4** | **3** | **2** | **1** |
| **Excellent** | **Good** | **Fair** | **Poor** | **Awful** |

**6.** Based on your analysis of all the classes in the package and any other criteria you think is appropriate, **rate the package for extendibility.**

| | | | | |
|---|---|---|---|---|
| **5** | **4** | **3** | **2** | **1** |
| **Excellent** | **Good** | **Fair** | **Poor** | **Awful** |

**7.** Based on your analysis of all the classes in the package and any other criteria you think is appropriate, **rate the package for effectiveness.**

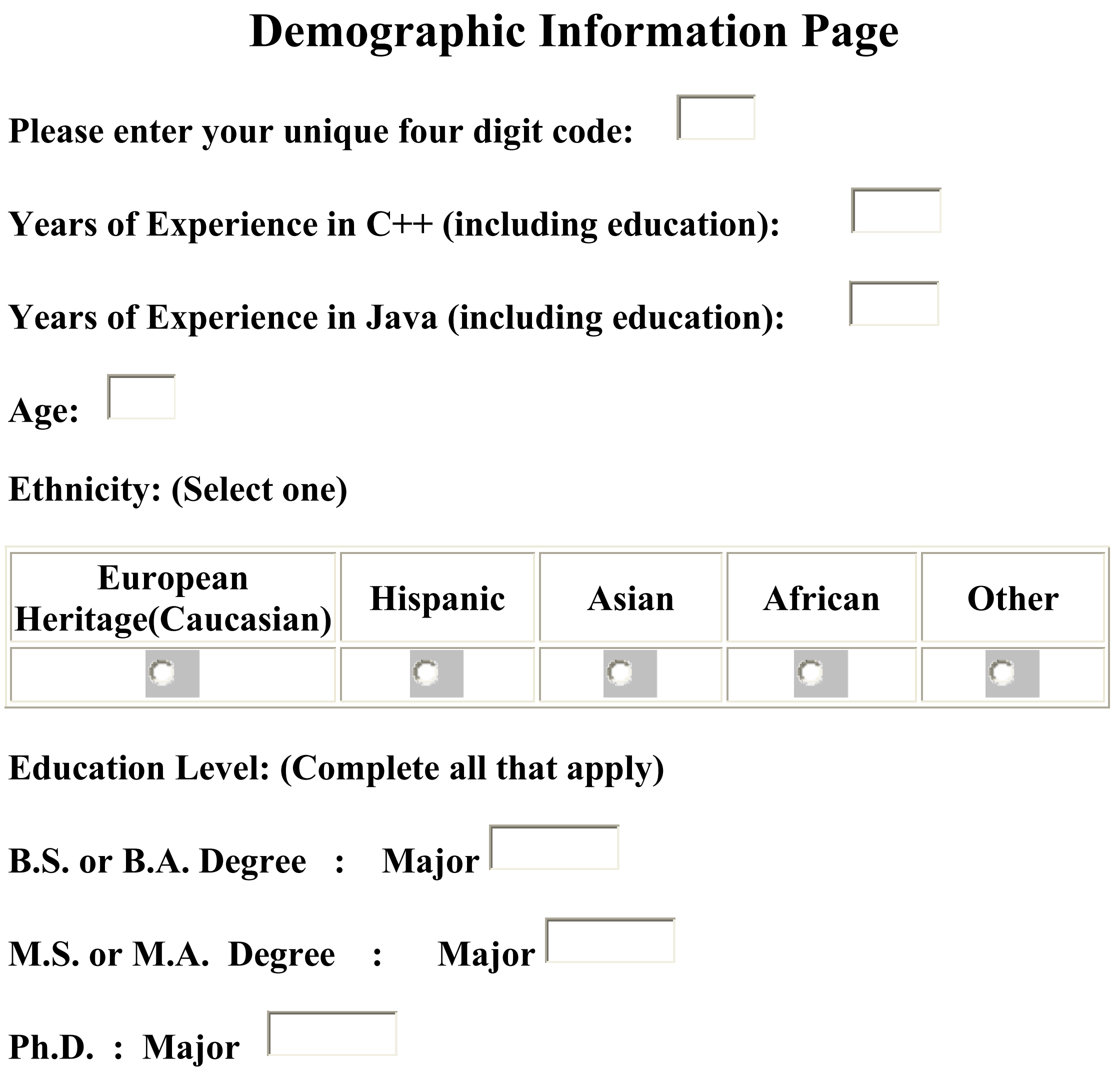
| | | | | |
|---|---|---|---|---|
| **5** | **4** | **3** | **2** | **1** |
| **Excellent** | **Good** | **Fair** | **Poor** | **Awful** |

**7.** Please rate the given package for **total quality** of the package

| | | | | |
|---|---|---|---|---|
| **5** | **4** | **3** | **2** | **1** |
| **Excellent** | **Good** | **Fair** | **Poor** | **Awful** |

## APPENDIX C

## DEMOGRAPHIC QUESTIONNAIRE



# Demographic Information Page

Please enter your unique four digit code: □

Years of Experience in C++ (including education): □

Years of Experience in Java (including education): □

Age: □

Ethnicity: (Select one)

| European Heritage(Caucasian) | Hispanic | Asian | African | Other |
|---|---|---|---|---|
| ○ | ○ | ○ | ○ | ○ |

Education Level: (Complete all that apply)

B.S. or B.A. Degree : Major □

M.S. or M.A. Degree : Major □

Ph.D. : Major □

# APPENDIX D

# EVALUATOR DEMOGRAPHICS

| Code | Student? | C_exp | Java_exp | Age | Ethnicity | Bachelors Degree | Masters Degree |
|---|---|---|---|---|---|---|---|
| 1295 | Y | 7 | 1 | 40 | 5 | Computer Science | |
| 1691 | Y | 2.00 | 2.00 | 22 | 5 | Pursuing CS | |
| 1529 | Y | 12 | 3 | 35 | 5 | Aerospace Engineering | |
| 1279 | Y | 6 | 1 | 32 | 5 | CS | |
| 2432 | N | 4 | 2 | 25 | 5 | computer science | |
| 1214 | Y | 2 | 1 | 23 | 3 | Information Technology | Computr Science |
| 2376 | Y | 1 | 1 | 24 | 5 | Computer Science | |
| 2277 | N | 5 | 2 | 27 | 5 | Computer Science | |
| 5013 | N | 4 | 1 | 33 | 5 | computer science | |
| 2452 | Y | 4 | 1 | 22 | 5 | | |
| 2439 | Y | 3 | 1 | 21 | 5 | | |
| 1859 | Y | 1 | 2 | 22 | 5 | CS and Math | |
| 2236 | Y | 1 | 2 | 22 | 5 | CS and Math | |
| 1952 | Y | 6 | 6 | 24 | 5 | Computer Science | |
| 2274 | Y | 3 | 0.5 | 22 | 5 | Computer Science | |
| 2017 | Y | 3 | 1 | 25 | 3 | CS | |
| 4750 | Y | 8 | 3 | 23 | 5 | | |
| 2182 | N | 7 | 5 | 29 | 4 | Computer Science | |
| 2064 | N | 3 | 3 | 25 | 2 | Senior | |
| 1628 | N | 4 | 2 | 42 | 5 | Computer Science | |
| 1546 | N | 6 | 6 | 24 | 5 | Computer Science | |
| 1770 | N | 2 | 1 | 32 | 5 | | |
| 4152 | N | 4 | 4 | 33 | 5 | Computer Engineering | |
| 1599 | Y | 1 | 3 | 26 | 3 | Computer Engineering | |
| 1500 | Y | 1 | 3 | 26 | 3 | Computer Engineering | |
| 1255 | Y | 4 | 0 | 22 | 3 | CS & Engineering | Computer Science |
| 1349 | Y | 4 | 1.5 | 23 | 3 | Information Technology | Computer Science |
| 1942 | N | 6 | 6 | 22 | 5 | Computer Science | |
| 1184 | Y | 4 | 1 | 24 | 3 | computer science | computer science |
| 3169 | N | 7 | 1 | 29 | 5 | Computer Science | |
| 1876 | N | 12 | 4 | 56 | 5 | Industrial Engineering | Theology |
| 2644 | Y | 4 | 3 | 22 | 5 | Computer Science | |
| 5299 | N | 3.5 | 0 | 38 | 5 | Computer Science | |
| 5217 | N | 2 | 0 | 31 | 5 | BS | |
| 5272 | N | 5 | 1 | 24 | 5 | Computer Science | |
| 5197 | N | 4 | 3 | 35 | 5 | Math/Computer Science | Probability and Stat. |
| 5129 | N | 4 | 0 | 28 | 5 | CS/Math/Physics | |
| 5318 | N | 3.5 | 0.5 | 40 | 5 | Math/ Computer Science | |
| 5233 | N | 4 | 1 | 24 | 5 | Computer Science | |
| 5283 | N | 3 | 0 | 30 | 5 | Computer Science | |
| 5259 | N | 6 | 0 | 25 | 5 | Computer Science | |
| 5072 | N | 15 | 0 | 47 | 4 | Computer Science | |

| Code | Student? | C_exp | Java_exp | Age | Ethnicity | Bachelors Degree | Masters Degree |
|---|---|---|---|---|---|---|---|
| 5161 | Y | 3 | 0.5 | 23 | 5 | Computer Science/Math | |
| 5137 | Y | 3 | 0 | 23 | 5 | | |
| 5136 | Y | 4 | 1 | 22 | 5 | | |
| 5123 | Y | 3 | 1 | 23 | 2 | Computer Science | |
| 5096 | Y | 3 | 0 | 22 | 5 | | |
| 5053 | Y | 6 | 3 | 21 | 5 | Computer Science | |
| 5075 | Y | 4 | 0 | 22 | 5 | | |
| 5134 | Y | 3 | 0 | 20 | 5 | | |
| 5169 | N | 6 | 2 | 38 | 5 | computer science | |
| 5055 | Y | 4 | 1 | 21 | 5 | Computer Science | |
| 5349 | Y | 3 | 0 | 22 | 5 | | |
| 5322 | N | 10 | 0 | 47 | 5 | Computer Science | |
| 5013 | N | 10 | 0 | 47 | 5 | Computer Science | |
| 5165 | Y | 4 | 0 | 22 | 5 | | |
| 5372 | N | 6 | 2 | 38 | 5 | computer science | |
| 5364 | Y | 3 | 1 | 23 | 2 | Computer Science | |
| 5319 | Y | 4 | 1 | 22 | 5 | Computer Science | |
| 5018 | N | 6 | 2 | 38 | 5 | computer science | |
| 5022 | Y | 4 | 1 | 22 | 5 | Computer Science | |
| 5433 | N | 6 | 2 | 26 | 5 | Computer Science | |
| 5326 | Y | 0 | 0 | 40 | 5 | CIS | computer science |
| 5136 | N | 4 | 1 | 24 | 5 | B.S. ComputerScience | |
| 5419 | N | 15 | 5 | 59 | 5 | CS, Biology | CS |
| 6757 | Y | 7 | 0 | 23 | 5 | CS / Math / Physics | |
| 6001 | Y | 9 | 5 | 25 | 5 | Computer Science | Mathematics |
| 5999 | N | 10 | 9 | 31 | 5 | Math/CS Education | Computer Science |
| 5707 | Y | 3 | 3 | 21 | 5 | | |
| 5747 | N | 7 | 2 | 27 | 5 | BS Computer Science | |
| 5804 | N | 5 | 4 | 25 | 5 | Computer Science | |
| 5775 | Y | 1 | 4 | 22 | 5 | Computer Science | |
| 5741 | N | 5 | 1 | 30 | 5 | Computer Science | CS. (in progress) |
| 5792 | Y | 4 | 1 | 21 | 5 | Computer Science | |
| 5800 | N | 3 | 1 | 25 | 5 | Computer Science | Software Engineering |
| 5716 | Y | 5 | 3 | 22 | 5 | Computer Science | |
| 5711 | Y | 4 | 1 | 21 | 5 | Computer Science | |
| 5746 | N | 9 | 2 | 27 | 5 | Computer Science | |
| 5775 | Y | 1 | 4 | 22 | 5 | Computer Science | |
| 5733 | N | 7 | 1 | 27 | 5 | CS | |
| 5729 | Y | 3 | 1 | 23 | 3 | Computer Science | N/A |
| 5742 | Y | 0 | 1 | 22 | 3 | CS | CS |
| 5729 | Y | 3 | 1 | 23 | 3 | computer science | n/a |

| Code | Student? | C_exp | Java_exp | Age | Ethnicity | Bachelors Degree | Masters Degree |
|---|---|---|---|---|---|---|---|
| 5703 | Y | 8 | 5 | 21 | 5 | | |
| 5807 | N | 1 | 0 | 40 | 5 | Information Systems | Computer Science |
| 5737 | Y | 4 | 0 | 24 | 5 | computer engineering | |
| 7006 | Y | 2 | 1 | 31 | 5 | Computer Science | S.E. (In progress) |
| 7009 | Y | 2 | 1 | 44 | 3 | EE | MS |
| 7018 | Y | 5 | 5 | 30 | 3 | EE | MS |
| 7024 | Y | 3 | 2 | 26 | 3 | computer applications | Pursuing MS in CS |
| 7027 | Y | 9 | 0 | 29 | 5 | Computer Science | |
| 7123 | Y | 0.5 | 0.9 | 25 | 3 | Computer Engineering | Computer Science |
| 7126 | Y | 2 | 4 | 24 | 3 | Comp. Engg. | pursuing Comp. Sc. |
| 7135 | Y | 12 | 4 | 33 | 2 | Computer Engineering | Computer Engineering |
| 7030 | Y | 3 | 2 | 47 | 5 | Science Management | Working on MSSE |
| 7033 | Y | 4 | 5 | 23 | 1 | ComputerScience | |
| 7036 | Y | 4 | 2 | 22 | 5 | B.S. Computer Science | |
| 7039 | Y | 8 | 3 | 23 | 3 | BS ,Major: Computers | Computer Science |
| 7042 | Y | 5 | 5 | 23 | 3 | CS | CS(ongoing) |
| 7045 | Y | 2 | 3 | 23 | 3 | B-Tech IT | MS CS |
| 7048 | Y | 2 | 1 | 23 | 3 | computer Engineering) | Computer Science |
| 7051 | Y | 2 | 1 | 24 | 3 | Computer Science | Computer Science |
| 7054 | Y | 2 | 2 | 24 | 2 | Computer Science | |
| 7084 | Y | 4 | 1 | 24 | 3 | Computer Science | Computer Science |
| 7087 | Y | 1 | 2 | 26 | 3 | Elec.s & Comm. Eng. | Pursuing CS |
| 7090 | Y | 4 | 0 | 27 | 3 | E. E. | Computer Science |
| 7093 | Y | 0.5 | 0.5 | 23 | 3 | Ind. Biotechnology | Computer Science |
| 7096 | Y | 1.25 | 1 | 23 | 3 | Electrical Engineering | Computer Science |
| 7099 | Y | 3 | 2 | 24 | 3 | B.E in Elec & Comm | M.S in C.S |
| 7102 | Y | 1 | 1 | 23 | 3 | Elec & Comm | Computer Science |
| 7105 | Y | 2 | 2.5 | 24 | 3 | Electrical&Electronics | Computer Science |
| 7108 | Y | 3 | 2 | 22 | 3 | B.E. (Comp Science) | Computer science |
| 7117 | Y | 2.5 | 4 | 27 | 3 | Information Technology | Computer science |
| 7129 | Y | 6 | 2 | 24 | 3 | Elec & Comm, Eng. | Computer Science |
| 7132 | Y | 0.5 | 1 | 22 | 3 | Electrical Engineering | Computer Science |
| 7138 | Y | 2 | 3 | 22 | 3 | CS Engineering | CS  (in progress) |
| 7144 | Y | 3 | 3 | 26 | 3 | MECHANICAL ENG | CS |
| 7147 | Y | 4 | 4 | 22 | 3 | Computer Science | ComputerScience |
| 7150 | Y | 3 | 1 | 24 | 3 | Computer Science | Computer Science |

5=Caucasian

4 = Hispanic

3 = Asian

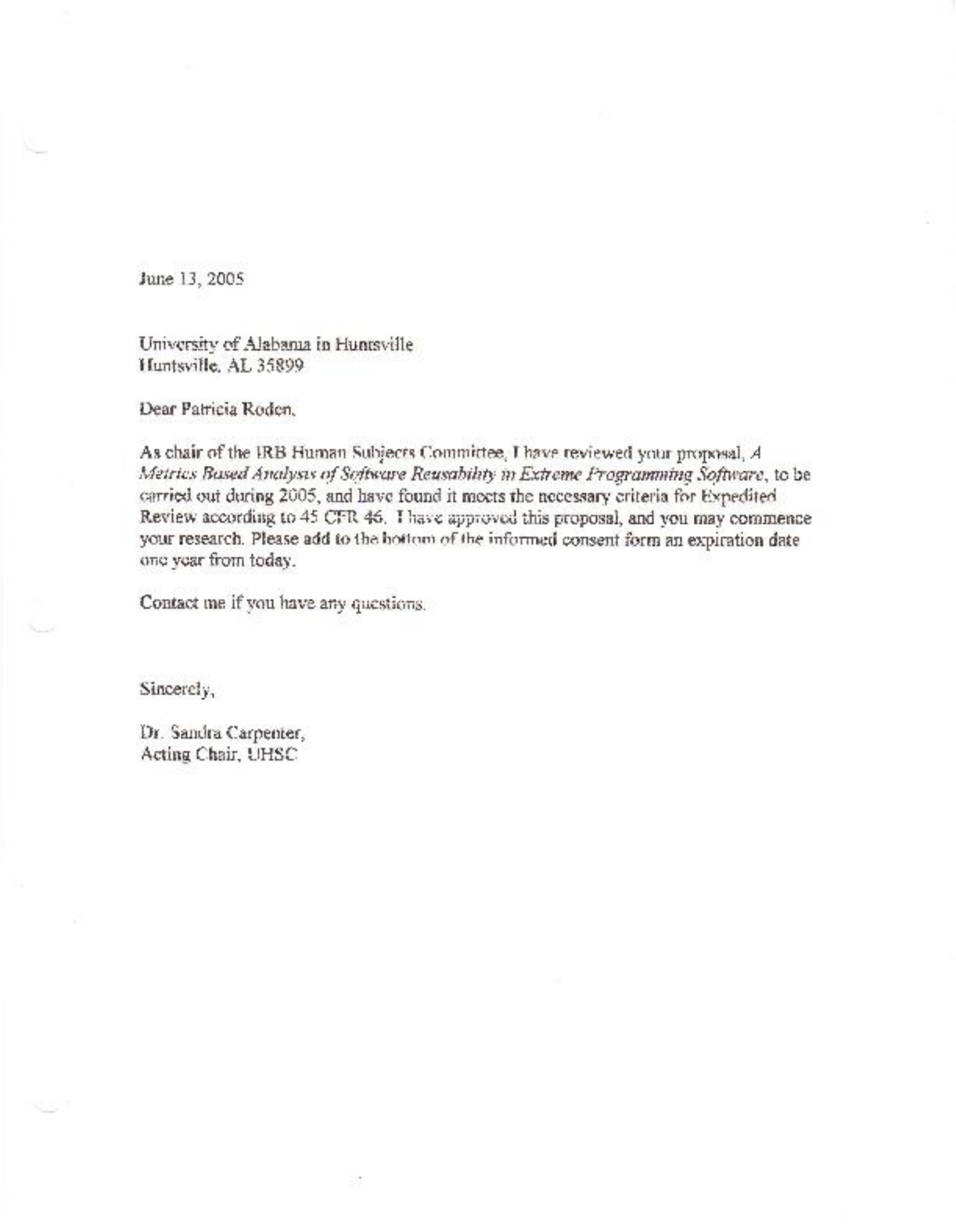
3 = Asian

2 = African

1 = Other

## APPENDIX E

## HUMAN SUBJECTS COMMITTEE APPROVAL



June 13, 2005

University of Alabama in Huntsville
Huntsville, AL 35899

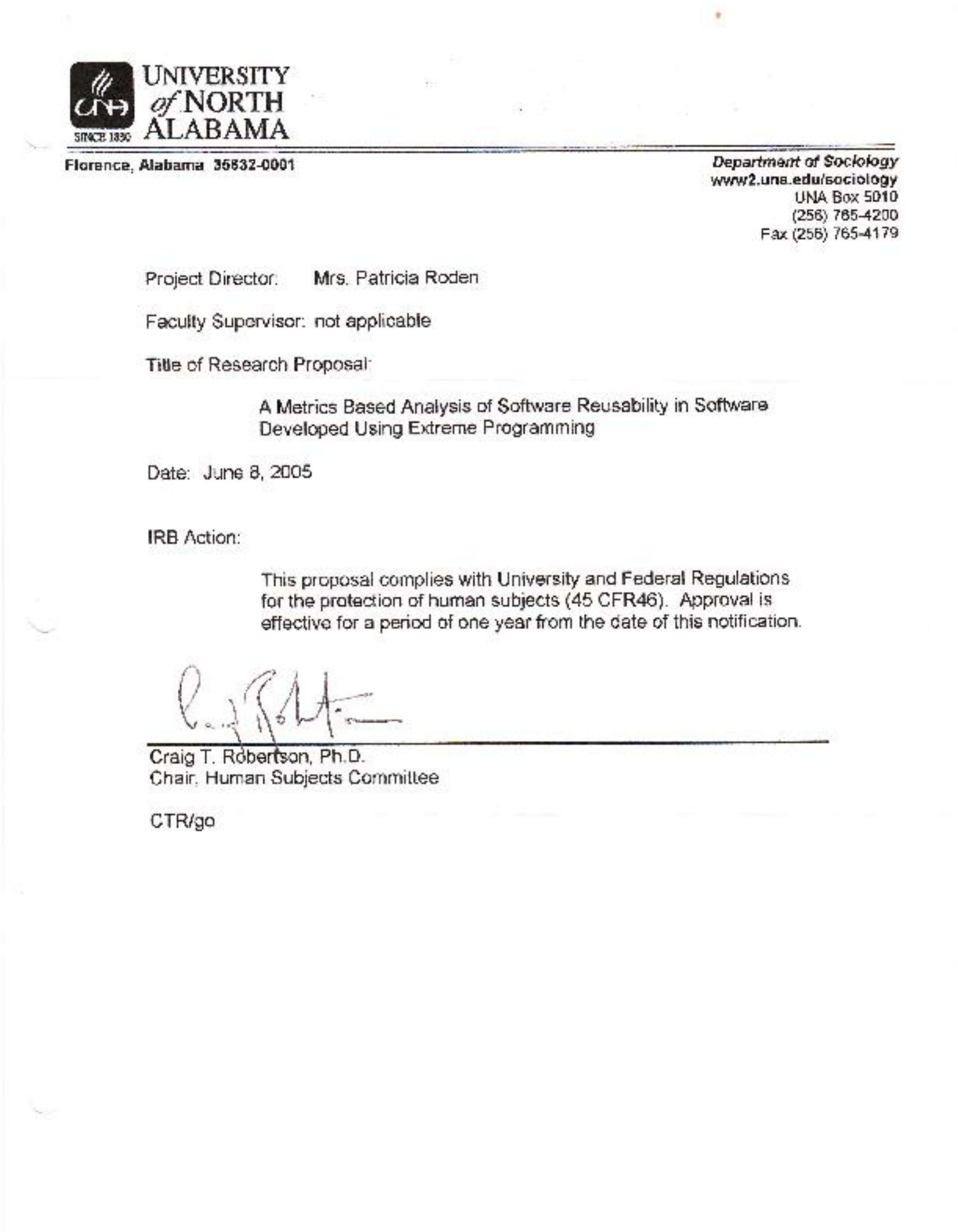
Dear Patricia Roden,

As chair of the IRB Human Subjects Committee, I have reviewed your proposal, *A Metrics Based Analysis of Software Reusability in Extreme Programming Software*, to be carried out during 2005, and have found it meets the necessary criteria for Expedited Review according to 45 CFR 46. I have approved this proposal, and you may commence your research. Please add to the bottom of the informed consent form an expiration date one year from today.

Contact me if you have any questions.

Sincerely,

Dr. Sandra Carpenter,
Acting Chair, UHSC



UNIVERSITY *of* NORTH ALABAMA
SINCE 1830

Florence, Alabama 35632-0001

*Department of Sociology*
www2.una.edu/sociology
UNA Box 5010
(256) 765-4200
Fax (256) 765-4179

Project Director: Mrs. Patricia Roden

Faculty Supervisor: not applicable

Title of Research Proposal:

A Metrics Based Analysis of Software Reusability in Software Developed Using Extreme Programming

Date: June 8, 2005

IRB Action:

This proposal complies with University and Federal Regulations for the protection of human subjects (45 CFR46). Approval is effective for a period of one year from the date of this notification.

Craig T. Robertson, Ph.D.
Chair, Human Subjects Committee

CTR/go

# APPENDIX F

# PACKAGE $R_{WG}$

## $R_{WG}$ for Packages

| Package | Data Group | Hierarchy | Packagesize | Reuse | Flexibility | Understandability | Functionality | Extendibility | Effectiveness | TotalQuality |
|---|---|---|---|---|---|---|---|---|---|---|
| TR1 | Unpaired | 0.762 | 0.548 | 0.762 | 0.262 | 0.690 | 0.857 | 0.333 | 0.929 | 0.548 |
| | Paired | 0.667 | 0.714 | 0.833 | 0.762 | 0.881 | 0.857 | 0.690 | 0.667 | 0.857 |
| | Combined | 0.560 | 0.635 | 0.810 | 0.538 | 0.791 | 0.865 | 0.525 | 0.714 | 0.723 |
| | | | | | | | | | | |
| TR2 | Unpaired | 0.214 | 0.595 | 0.833 | 0.762 | 0.762 | 0.881 | 0.548 | 0.714 | 0.762 |
| | Paired | 1.000 | 0.500 | 1.000 | 0.500 | 1.000 | 0.833 | 0.833 | 0.833 | 0.833 |
| | Combined | 0.394 | 0.644 | 0.778 | 0.839 | 0.728 | 0.911 | 0.661 | 0.772 | 0.772 |
| | | | | | | | | | | |
| TR3 | Unpaired | 0.524 | 0.262 | 0.857 | 0.833 | 0.762 | 0.929 | 0.667 | 0.857 | 0.857 |
| | Paired | 0.100 | 0.900 | 0.900 | 0.850 | 0.900 | 0.650 | 0.850 | 0.850 | 1.000 |
| | Combined | 0.955 | 0.954 | 0.981 | 0.992 | 0.987 | 0.992 | 0.981 | 0.992 | 0.992 |
| | | | | | | | | | | |
| TR4 | Unpaired | 0.262 | 0.714 | -0.119 | 0.048 | 0.595 | 0.333 | 0.214 | 0.381 | 0.595 |
| | Paired | 0.179 | 0.857 | 0.009 | 0.795 | -0.429 | 0.464 | 0.223 | 0.464 | 0.571 |
| | Combined | -0.071 | 0.795 | -0.033 | 0.467 | -0.062 | 0.238 | 0.271 | 0.343 | 0.514 |
| | | | | | | | | | | |
| TR5 | Unpaired | 0.857 | 0.881 | 0.690 | 0.429 | -0.167 | 0.500 | 0.190 | 0.048 | 0.357 |
| | Paired | 0.625 | 0.736 | 0.944 | 0.944 | 0.750 | 0.861 | 0.361 | 0.736 | 0.861 |
| | Combined | 0.635 | 0.719 | 0.808 | 0.733 | 0.342 | 0.685 | 0.169 | 0.408 | 0.502 |
| | | | | | | | | | | |
| E3 | Unpaired | 0.167 | 0.690 | 0.357 | 0.190 | 0.262 | 0.381 | 0.333 | 0.333 | 0.429 |
| | | | | | | | | | | |
| C2 | Unpaired | 0.381 | 0.714 | 0.595 | 0.595 | 0.762 | 0.690 | 0.167 | 0.690 | 0.381 |
| | | | | | | | | | | |
| C1 | Unpaired | 0.714 | 0.762 | 0.714 | 0.595 | 0.190 | 0.881 | 0.548 | 0.762 | 0.762 |
| | | | | | | | | | | |
| E1 | Unpaired | 0.762 | 0.857 | 0.667 | 0.667 | 0.095 | 0.357 | 0.595 | 0.262 | 0.667 |
| | | | | | | | | | | |
| C4 | Unpaired | 0.714 | 0.714 | 0.190 | 0.381 | 0.714 | 0.714 | 0.262 | 0.762 | 0.667 |
| | Paired | 0.500 | 0.875 | 0.653 | 0.653 | 0.694 | 0.819 | 0.694 | 0.903 | 0.944 |
| | Combined | 0.600 | 0.752 | 0.435 | 0.533 | 0.633 | 0.742 | 0.542 | 0.852 | 0.835 |
| | | | | | | | | | | |
| C3 | Unpaired | 0.524 | 0.762 | 0.833 | 0.667 | 0.881 | 0.929 | 0.762 | 0.857 | 0.762 |
| | | | | | | | | | | |

## $R_{WG}$ for Packages

| Package | Data Group | Hierarchy | Packagesize | Reuse | Flexibility | Understandability | Functionality | Extendibility | Effectiveness | TotalQuality |
|---|---|---|---|---|---|---|---|---|---|---|
| E2 | Unpaired | 0.262 | 0.429 | 0.524 | 0.548 | 0.333 | 0.548 | 0.190 | -<br>0.119 | 0.548 |
| | Paired | 0.778 | 0.736 | 0.278 | 0.694 | 0.653 | 0.319 | 0.569 | 0.500 | 0.750 |
| | Combined | 0.569 | 0.619 | 0.408 | 0.608 | 0.542 | 0.452 | 0.433 | 0.275 | 0.619 |
| | | | | | | | | | | |
| P13 | Unpaired | 0.524 | 0.762 | 0.762 | 0.762 | 0.881 | 0.690 | 0.762 | 0.762 | 0.881 |
| | | | | | | | | | | |
| D1 | Unpaired | 0.857 | 0.833 | 0.524 | 0.595 | 0.048 | 0.690 | 0.690 | 0.833 | 0.714 |
| | | | | | | | | | | |
| D2 | Unpaired | 0.381 | 0.762 | 0.762 | 0.929 | 0.762 | 0.690 | 0.857 | 0.690 | 0.690 |
| | | | | | | | | | | |
| D3 | Unpaired | 0.690 | 0.881 | 0.690 | 0.429 | 0.548 | 0.595 | 0.524 | 0.524 | 0.762 |
| | Paired | 0.929 | 0.881 | 0.762 | 0.762 | 0.714 | 0.548 | 0.262 | 0.548 | 0.690 |
| | Combined | 0.799 | 0.865 | 0.679 | 0.558 | 0.615 | 0.580 | 0.371 | 0.503 | 0.415 |
| | | | | | | | | | | |
| D4 | Unpaired | 0.595 | 0.857 | 0.429 | 0.595 | 0.833 | 0.690 | 0.595 | 0.714 | 0.429 |
| | Paired | 0.667 | 1.000 | 0.875 | 0.833 | 0.667 | 0.875 | 1.000 | 0.875 | 0.875 |
| | Combined | 0.655 | 0.873 | 0.464 | 0.655 | 0.673 | 0.764 | 0.755 | 0.791 | 0.600 |
| | | | | | | | | | | |
| B1 | Unpaired | 0.881 | 0.381 | 0.714 | 0.762 | 0.595 | 0.524 | 0.429 | 0.929 | 0.500 |
| | | | | | | | | | | |
| B2 | Unpaired | 0.000 | 0.214 | 0.190 | 0.595 | 0.024 | 0.762 | 0.762 | 0.881 | 0.667 |
| | | | | | | | | | | |
| B3 | Unpaired | 0.500 | 0.690 | 0.381 | 0.524 | 0.167 | 0.857 | 0.833 | 0.762 | 0.690 |
| | | | | | | | | | | |
| B4 | Unpaired | 0.438 | 0.723 | 0.723 | 0.509 | 0.509 | 0.750 | 0.714 | 0.866 | 0.714 |
| | | | | | | | | | | |
| A1 | Unpaired | 0.548 | 0.762 | 0.381 | 0.833 | 0.262 | 0.333 | 0.667 | 0.333 | 0.595 |
| | | | | | | | | | | |
| A2 | Unpaired | 0.381 | 0.762 | 0.833 | 0.548 | 0.524 | 0.714 | 0.500 | 0.524 | 0.833 |
| | | | | | | | | | | |
| A3 | Unpaired | 0.595 | 0.262 | 0.595 | 0.850 | 0.095 | 0.167 | 0.714 | 0.524 | 0.357 |
| | | | | | | | | | | |
| A4 | Unpaired | 0.667 | 0.850 | 0.317 | 0.917 | 0.600 | 0.317 | 0.717 | 0.800 | 0.650 |
| | Paired | 0.850 | 0.750 | 0.750 | 0.850 | 0.500 | 0.650 | 0.650 | 0.650 | 0.750 |
| | Combined | 0.700 | 0.791 | 0.555 | 0.891 | 0.578 | 0.500 | 0.700 | 0.755 | 0.691 |

# APPENDIX G

## CLASS $R_{WG}$

Note:  Only final iterations had paired and combined data collected

| RWG for Classes | | | | |
|---|---|---|---|---|
| PACKAGE | CLASS | Unpaired Reusability | Paired Reusability | Combined Reusability |
| P1 | P1A | 0.524 | 0.690 | 0.538 |
| | P1B | 0.548 | 0.524 | 0.473 |
| | P1C | 0.595 | 0.524 | 0.569 |
| | P1D | 0.690 | 0.524 | 0.538 |
| | | | | |
| P2 | P2A | 0.524 | 0.208 | 0.464 |
| | P2B | 0.429 | 0.875 | 0.583 |
| | P2C | 0.357 | 0.542 | 0.464 |
| | P2D | 0.357 | 0.875 | 0.355 |
| | P2E | 0.381 | 0.833 | 0.573 |
| | P2F | 0.779 | 0.542 | 0.221 |
| | P2G | 0.381 | 0.875 | 0.591 |
| | P2H | 0.722 | 0.875 | 0.255 |
| | P2I | 0.568 | 0.833 | 0.500 |
| | P2J | 0.429 | 0.891 | 0.191 |
| | P2K | 0.333 | 0.500 | 0.582 |
| | P2L | 0.429 | 0.333 | 0.364 |
| | P2M | 0.524 | 0.500 | 0.418 |
| | | | | |
| P3 | P3A | 0.429 | 0.900 | 0.636 |
| | P3B | 0.714 | 0.500 | 0.673 |
| | P3C | 0.595 | 0.208 | 0.473 |
| | P3D | 0.524 | 0.900 | 0.652 |
| | P3E | 0.690 | 0.400 | 0.561 |
| | P3F | 0.524 | 0.682 | 0.140 |
| | P3G | 0.524 | 0.850 | 0.686 |
| | | | | |
| P4 | P4A | -0.119 | 0.223 | 0.129 |
| | P4B | -0.143 | 0.000 | -0.014 |
| | P4C | 0.190 | 0.438 | 0.367 |
| | | | | |
| P5 | P5A | 0.381 | 0.861 | 0.608 |
| | P5B | 0.429 | 0.750 | 0.633 |
| | P5C | 0.519 | 0.236 | 0.444 |
| | P5D | 0.664 | 0.750 | 0.419 |
| | P5E | 0.429 | 0.403 | 0.452 |

## RWG for Classes

| PACKAGE | CLASS | Unpaired Reusability | Paired Reusability | Combined Reusability |
|---|---|---|---|---|
| P6 | P6A | 0.381 | n/a | n/a |
| | P6B | 0.024 | n/a | n/a |
| | P6C | 0.667 | n/a | n/a |
| | P6D | 0.524 | n/a | n/a |
| | P6E | 0.762 | n/a | n/a |
| | P6F | 0.524 | n/a | n/a |
| | P6G | 0.548 | n/a | n/a |
| | P6H | 0.690 | n/a | n/a |
| | P6I | 0.333 | n/a | n/a |
| | P6J | 0.524 | n/a | n/a |
| | P6K | 0.524 | n/a | n/a |
| | P6L | 0.357 | n/a | n/a |
| | P6M | 0.333 | n/a | n/a |
| | P6N | 0.524 | n/a | n/a |
| | P6O | 0.442 | n/a | n/a |
| | P6P | 0.357 | n/a | n/a |
| | P6Q | 0.614 | n/a | n/a |
| | P6R | 0.381 | n/a | n/a |
| | P6S | 0.262 | n/a | n/a |
| | | | | |
| P7 | P7A | 0.762 | n/a | n/a |
| | P7B | 0.429 | n/a | n/a |
| | P7C | 0.381 | n/a | n/a |
| | P7D | 0.429 | n/a | n/a |
| | P7E | 0.548 | n/a | n/a |
| | P7F | 0.500 | n/a | n/a |
| | | | | |
| P8 | P8A | 0.766 | n/a | n/a |
| | | | | |
| P9 | P9A | 0.381 | n/a | n/a |
| | P9B | 0.381 | n/a | n/a |
| | P9C | 0.548 | n/a | n/a |
| | P9D | 0.595 | n/a | n/a |
| | P9E | 0.595 | n/a | n/a |
| | P9F | 0.500 | n/a | n/a |

## RWG for Classes

| PACKAGE | CLASS | Unpaired Reusability | Paired Reusability | Combined Reusability |
|---|---|---|---|---|
| P10 | P10A | 0.378 | 0.750 | 0.408 |
|  | P10B | 0.381 | 0.375 | 0.352 |
|  | P10C | 0.595 | 0.625 | 0.542 |
|  | P10D | 0.667 | 0.625 | 0.608 |
|  | P10E | 0.667 | 0.361 | 0.485 |
|  | P10F | 0.524 | 0.500 | 0.500 |
|  | P10G | 0.714 | 0.625 | 0.685 |
|  |  |  |  |  |
| P11 | P11A | 0.690 | n/a | n/a |
|  | P11B | 0.524 | n/a | n/a |
|  |  |  |  |  |
| P12 | P12A | 0.442 | 0.457 | 0.451 |
|  | P12B | 0.381 | 0.278 | 0.333 |
|  | P12C | 0.690 | 0.572 | 0.581 |
|  | P12D | 0.690 | 0.491 | 0.552 |
|  | P12E | 0.500 | 0.403 | 0.402 |
|  | P12F | 0.333 | 0.403 | 0.335 |
|  | P12G | 0.357 | 0.278 | 0.352 |
|  | P12H | 0.333 | 0.528 | 0.402 |
|  | P12I | 0.548 | 0.531 | 0.540 |
|  | P12J | 0.381 | 0.194 | 0.275 |
|  | P12K | 0.701 | 0.504 | 0.603 |
|  | P12L | 0.667 | 0.653 | 0.485 |
|  | P12M | 0.714 | 0.236 | 0.469 |
|  | P12N | 0.511 | -0.014 | -0.231 |
|  | P12O | 0.524 | -0.014 | 0.267 |
|  | P12P | 0.619 | 0.111 | 0.075 |
|  | P12Q | 0.714 | 0.500 | 0.608 |
|  | P12R | 0.524 | 0.361 | 0.469 |

## RWG for Classes

| PACKAGE | CLASS | Unpaired Reusability | Paired Reusability | Combined Reusability |
|---|---|---|---|---|
| P13 | P13A | 0.762 | n/a | n/a |
| | P13B | 0.429 | n/a | n/a |
| | P13C | 0.595 | n/a | n/a |
| | P13D | 0.400 | n/a | n/a |
| | P13E | 0.762 | n/a | n/a |
| | P13F | 0.690 | n/a | n/a |
| | P13G | 0.762 | n/a | n/a |
| | P13H | 0.762 | n/a | n/a |
| | P13I | 0.429 | n/a | n/a |
| | P13J | 0.714 | n/a | n/a |
| | P13K | 0.690 | n/a | n/a |
| | P13L | 0.881 | n/a | n/a |
| | P13M | 0.762 | n/a | n/a |
| | P13N | 0.857 | n/a | n/a |
| | P13O | 0.595 | n/a | n/a |
| | P13P | 0.357 | n/a | n/a |
| | | | | |
| T1 | T1A | 0.714 | n/a | n/a |
| | T1B | 0.595 | n/a | n/a |
| | T1C | 0.762 | n/a | n/a |
| | T1D | 0.381 | n/a | n/a |
| | | | | |
| T2 | T2A | 0.381 | n/a | n/a |
| | T2B | 0.857 | n/a | n/a |
| | T2C | 0.762 | n/a | n/a |
| | T2D | 0.548 | n/a | n/a |
| | | | | |
| T3 | T3A | 0.712 | 0.595 | 0.649 |
| | T3B | 0.433 | 0.599 | 0.518 |
| | T3C | 0.548 | 0.762 | 0.582 |
| | T3D | 0.595 | 0.524 | 0.549 |
| | T3E | 0.667 | 0.592 | 0.618 |
| | T3F | 0.548 | 0.357 | 0.484 |
| | T3G | 0.881 | 0.381 | 0.385 |
| | T3H | 0.690 | 0.690 | 0.615 |

| RWG for Classes | | | | |
|---|---|---|---|---|
| PACKAGE | CLASS | Unpaired Reusability | Paired Reusability | Combined Reusability |
| T4 | T4A | 0.357 | 0.208 | 0.200 |
| | T4B | 0.548 | 0.557 | 0.540 |
| | T4C | 0.429 | 0.542 | 0.364 |
| | T4D | 0.333 | 0.667 | 0.500 |
| | T4E | 0.333 | 0.542 | 0.264 |
| | T4F | 0.548 | 0.612 | 0.582 |
| | T4G | 0.429 | 0.542 | 0.364 |
| | T4H | 0.701 | 0.667 | 0.664 |
| | | | | |
| T5 | T5A | 0.262 | n/a | n/a |
| | T5B | 0.690 | n/a | n/a |
| | T5C | 0.548 | n/a | n/a |
| | T5D | -0.071 | n/a | n/a |
| | T5E | 0.429 | n/a | n/a |
| | T5F | 0.690 | n/a | n/a |
| | T5G | 0.690 | n/a | n/a |
| | T5H | 0.690 | n/a | n/a |
| | T5I | 0.548 | n/a | n/a |
| | T5J | 0.690 | n/a | n/a |
| | T5K | 0.262 | n/a | n/a |
| | T5L | 0.429 | n/a | n/a |
| | | | | |
| T6 | T6A | 0.595 | n/a | n/a |
| | T6B | 0.714 | n/a | n/a |
| | T6C | 0.429 | n/a | n/a |
| | T6D | 0.357 | n/a | n/a |
| | T6E | 0.429 | n/a | n/a |
| | T6F | 0.357 | n/a | n/a |
| | T6G | 0.595 | n/a | n/a |
| | T6H | 0.881 | n/a | n/a |
| | T6I | 0.381 | n/a | n/a |
| | T6J | 0.595 | n/a | n/a |
| | T6K | 0.690 | n/a | n/a |
| | T6L | 0.857 | n/a | n/a |
| | T6M | 0.690 | n/a | n/a |
| | T6N | 0.690 | n/a | n/a |
| | T6O | 0.095 | n/a | n/a |
| | T6P | 0.690 | n/a | n/a |
| | T6Q | 0.595 | n/a | n/a |
| | T6R | 0.095 | n/a | n/a |
| | T6S | 0.214 | n/a | n/a |

## RWG for Classes

| PACKAGE | CLASS | Unpaired Reusability | Paired Reusability | Combined Reusability |
|---|---|---|---|---|
| T7 | T7A | 0.524 | n/a | n/a |
| | T7B | 0.524 | n/a | n/a |
| | T7C | 0.612 | n/a | n/a |
| | T7D | 0.589 | n/a | n/a |
| | T7E | 0.674 | n/a | n/a |
| | T7F | 0.667 | n/a | n/a |
| | T7G | 0.524 | n/a | n/a |
| | T7H | 0.511 | n/a | n/a |
| | T7I | 0.595 | n/a | n/a |
| | T7J | 0.481 | n/a | n/a |
| | T7K | 0.690 | n/a | n/a |
| | T7L | 0.857 | n/a | n/a |
| | T7M | 0.690 | n/a | n/a |
| | T7N | 0.690 | n/a | n/a |
| | T7O | 0.510 | n/a | n/a |
| | T7P | 0.690 | n/a | n/a |
| | T7Q | 0.595 | n/a | n/a |
| | T7R | 0.822 | n/a | n/a |
| | T7S | 0.214 | n/a | n/a |
| | T7T | 0.214 | n/a | n/a |
| | | | | |
| T8 | T8A | 0.509 | n/a | n/a |
| | T8B | 0.723 | n/a | n/a |
| | T8C | 0.509 | n/a | n/a |
| | T8D | 0.295 | n/a | n/a |
| | T8E | 0.857 | n/a | n/a |
| | T8F | 0.571 | n/a | n/a |
| | T8G | 0.750 | n/a | n/a |
| | T8H | 0.857 | n/a | n/a |
| | T8I | 0.893 | n/a | n/a |
| | T8J | 0.795 | n/a | n/a |
| | T8K | 0.714 | n/a | n/a |
| | T8L | 0.438 | n/a | n/a |
| | T8M | 0.723 | n/a | n/a |
| | T8N | 0.652 | n/a | n/a |
| | T8O | 0.795 | n/a | n/a |
| | T8P | 0.652 | n/a | n/a |
| | T8Q | 0.857 | n/a | n/a |
| | T8R | 0.857 | n/a | n/a |
| | T8S | 0.571 | n/a | n/a |
| | T8T | 0.438 | n/a | n/a |
| | T8U | 0.152 | n/a | n/a |
| | T8V | 0.690 | n/a | n/a |

## RWG for Classes

| PACKAGE | CLASS | Unpaired Reusability | Paired Reusability | Combined Reusability |
|---|---|---|---|---|
| T9 | T9A | 0.333 | n/a | n/a |
| | T9B | 0.524 | n/a | n/a |
| | T9C | 0.381 | n/a | n/a |
| | T9D | 0.762 | n/a | n/a |
| | T9E | 0.524 | n/a | n/a |
| | | | | |
| T10 | T10A | 0.595 | n/a | n/a |
| | T10B | 0.333 | n/a | n/a |
| | T10C | 0.524 | n/a | n/a |
| | T10D | 0.762 | n/a | n/a |
| | T10E | 0.500 | n/a | n/a |
| | | | | |
| T11 | T11A | 0.595 | n/a | n/a |
| | T11B | 0.540 | n/a | n/a |
| | T11C | 0.488 | n/a | n/a |
| | T11E | 0.429 | n/a | n/a |
| | T11F | 0.518 | n/a | n/a |
| | | | | |
| T12 | T12A | 0.778 | 0.524 | 0.685 |
| | T12B | 0.500 | 0.100 | 0.140 |
| | T12C | 0.524 | 0.850 | 0.504 |
| | T12D | 0.690 | 0.600 | 0.561 |
| | T12E | 0.262 | 0.850 | 0.470 |
| | T12F | 0.524 | 0.750 | 0.606 |
| | T12H | 0.524 | 0.750 | 0.606 |
| | T12I | 0.262 | 0.900 | 0.561 |
| | | | | |

# APPENDIX H

# COLLINEARITY STUDIES

# AND

# LINEAR REGRESSION RESULTS

**Table H.1: Collinearity Analysis of Model 1**

| Model 1 | | |
|---|---|---|
| | VIF | Condition # |
| CBO | 18.728 | 2.218 |
| DIT | 1.676 | 3.176 |
| LCOM | 1.805 | 4.664 |
| NOC | 1.806 | 6.172 |
| RFC | 27.578 | 9.858 |
| WMC | 11.357 | 37.718 |

**Table H.2: Collinearity Analysis of Model 2**

| Model 2 | | |
|---|---|---|
| | VIF | Condition # |
| CBO | 1.612 | 2.015 |
| DIT | 1.665 | 2.977 |
| LCOM | 1.455 | 4.27 |
| NOC | 1.741 | 5.656 |
| WMC | 1.696 | 9.781 |

**Table H.3: Linear Regression of Model 2**

| Linear Regression Results | | | | | |
|---|---|---|---|---|---|
| Model 2 | | | | | |
| | Coefficient | Standard Error | Std. Coefficient | t | Sig. |
| Constant | 64.834 | 70.767 | - | 0.916 | 0.379 |
| CBO | 3.826 | 11.308 | 0.112 | 0.338 | 0.741 |
| DIT | -67.31 | 63.264 | -0.36 | -1.064 | 0.307 |
| LCOM | -0.098 | 0.406 | -0.077 | -0.243 | 0.812 |
| NOC | 26.501 | 188.007 | 0.049 | 0.141 | 0.89 |
| WMC | 2.176 | 6.246 | 0.119 | 0.348 | 0.733 |

### Table H.4: Collinearity Analysis of Model 3

| Model 3 | | |
|---|---|---|
| | VIF | Condition # |
| CBO | 3.128 | 2.048 |
| DIT | 1.674 | 3.09 |
| LCOM | 1.401 | 5.544 |
| NOC | 1.804 | 5.755 |
| RFC | 4.112 | 12.726 |

### Table H.5: Linear Regression of Model 3

| Linear Regression Results | | | | | |
|---|---|---|---|---|---|
| Model 3 | | | | | |
| | Coefficient | Standard Error | Std. Coefficient | t | Sig. |
| Constant | 75.531 | 62.416 | - | 1.21 | 0.248 |
| CBO | 0.49 | 15.801 | 0.014 | 0.031 | 0.976 |
| DIT | -64.594 | 63.634 | -0.345 | -1.015 | 0.329 |
| LCOM | -0.07 | 0.4 | -0.055 | -0.176 | 0.963 |
| NOC | 11.751 | 191.939 | 0.022 | 0.061 | 0.952 |
| RFC | 0.89 | 4.5 | 0.105 | 0.198 | 0.846 |

### Table H.6: Collinearity Analysis of Model 4

| Model 4 | | |
|---|---|---|
| | VIF | Condition # |
| CBO | 1.492 | 5.151 |
| DIT | 1.517 | 1.836 |
| LCOM | 1.401 | 2.968 |
| NOC | 1.188 | 4.969 |

**Table H.7: Linear Regression of Model 4**

| Linear Regression Results | | | | | |
|---|---|---|---|---|---|
| Model 4 | | | | | |
| | Coefficient | Standard Error | Std. Coefficient | t | Sig. |
| Constant | 84.556 | 41.115 | - | 2.057 | 0.059 |
| CBO | 2.75 | 10.532 | 0.081 | 0.261 | 0.798 |
| DIT | -60.74 | 58.462 | -0.324 | -1.039 | 0.316 |
| LCOM | -0.071 | 0.386 | -0.055 | -0.185 | 0.856 |
| NOC | -10.443 | 150.301 | -0.019 | -0.069 | 0.946 |

# REFERENCES

Abrahamsson, P. & Koskela, J., “Extreme Programming: A Survey of Empirical Data from a Controlled Case Study.” *Proceedings of the 2004 International Symposium on Empirical Software Engineering 2004 ISESE ’04,* 19-20 August 2004, pp. 73-82.

Abrahamsson, P., Warsta, J., Siponen, M. T., & Ronkainen, J., “New Directions on Agile Methods: A Comparative Analysis.” *Proceedings of the 25th International Conference on Software Engineering,* 3-10 May 2003, pp. 244-254.

Abreu, F. B., & Melo W., “Evaluating the Impact of Object-Oriented Design on Software Quality.” *Proceedings of the Third International Software Metrics Symposium,* 25-26 March 1996, pp. 90-99.

Ågerfalk, P. & Fitzgerald, B., “Flexible and Distributed Software Processes: Old Petunias in New Bowls?” *Communications of the ACM,* Vol. 49, No. 10, October 2006, pp. 27-34.

Agile Manifesto, http://agilemanifesto.org/, [Last accessed October 20, 2008].

Almeida, M., Lounis, H., & Melo, W., “An Investigation on the Use of Machine Learned Models for Estimating Software Correctability.” *International Journal of Software Engineering and Knowledge Engineering,* Vol. 9, Issue 5, October 1999, pp. 565-593.

Alshayeb, M. & Li, W., “An Empirical Study of Relationships Among Extreme Programming Engineering Activities.” *Information and Software Technology,* Vol. 48, No. 11, November 2006, pp. 1068-1072.

Alyshayeb, M., & Li, W., “An Empirical Study of System Design Instability Metric and Design Evolution in an Agile Software Process.” *The Journal of Systems and Software,* Vol. 74, No. 3, February 2005, pp. 269-274.

Alshayeb, M. & Li, W., “An Empirical Validation of Object-Oriented Metrics in Two Different Iterative Software Processes.” *IEEE Transactions on Software Engineering,* Vol. 29, No. 11, November 2003, pp. 1043-1049.

Ambler, S. “Lessons in Agility From Internet-Based Development.” *IEEE Software,* Vol. 19, Issue 2, March-April 2002, pp. 66-73.

Anderson, A., Beattie, R., Beck, K., et al. "Chrysler Goes to Extremes." *Distributed Computing,* Vol. 1, No. 10, October 1998, pp. 24 – 28.

Balasubramanian, N., "Object-oriented Metrics." *Proceedings of the 1996 Asia-Pacific Software Engineering Conference,* December 1996, pp. 30-34.

Bansiya, J., "Evaluating Framework Architecture Structural Stability." *ACM Computing Surveys,* Vol. 32, Issue 1, March 2000, Article 18.

Bansiya, J., Etzkorn, L., Davis, C. & Li, W., "A Class Cohesion Metric for Object-Oriented Designs." *Journal of Object-Oriented Programming*, Vol.11, No. 8, January 1999, pp. 47-52.

Bansiya, J., Davis, C., & Etzkorn, L., "An Entropy-Based Complexity Measure for Object-Oriented Designs." *Theory and Practice of Object Systems,* Vol. 5, No. 2, 1999, pp. 111-118.

Bansiya, J., & Davis C. G., "A Hierarchical Model for Object-Oriented Design Quality Assessment." *IEEE Transactions on Software Engineering,* Vol. 28, No. 1, January 2002, pp. 4-17.

Basili, V., Briand, L., & Melo, W., "A Validation of Object-Oriented Design Metrics as Quality Indicators." *IEEE Transactions on Software Engineering*, Vol. 22, No. 10, October 1996, pp. 751-761.

Baskerville, R., Levine, L., Pries-Heje, J., Ramesh, B., & Slaughter, S., "How Internet Software Companies Negotiate Quality." *IEEE Computer,* Vol. 34, Issue 5, May 2001, pp. 51-57.

Baum, L., & Becker, M., "Generic Components to Foster Reuse." *Proceedings of the 37th International Conference On Technology of Object-Oriented Languages and Tools*, 20-23 November 2000, pp.266-277.

Beck, K., *Extreme Programming Explained: Embrace Change.* Reading, MA: Addison-Wesley, 2000.

Beck, K., "Embracing Change with Extreme Programming." *IEEE Computer,* Vol. 32, Issue 10, October 1999, pp. 70-77.

Berander, P., "Using Students as Subjects in Requirements Prioritization." *Proceedings of the 2004 International Symposium on Empirical Software Engineering,* August 2004, pp.167-176.

Beynon-Davies, P., & Williams, M. D., "The Diffusion of Information Systems Development Methods." *Journal of Strategic Information Systems,* Vol. 12, Issue 1, March 2003, pp. 29-46.

Boehm, B., "Get Ready for Agile Methods, with Care." *IEEE Computer,* Vol. 35, Issue 1, January 2002, pp. 64-69.

Boehm, B., & Turner, R., "Balancing Agility and Discipline: Evaluating and Integrating Agile and Plan-Drive Methods." *Proceedings of the 26th International Conference on Software Engineering,* May 2004, pp. 718-719.

Boudnik, K., "Software Reliability." http://weblogs.java.net/blog/cos/archive/2007/03/software_reliab_1.html [last accessed September 18, 2008.].

Brilliant, S., & Knight, J., "Empirical Research in Software Engineering." *ACM SIGSOFT Software Engineering Notes,* Vol. 24, No. 3, May 1999, pp. 45-52.

Brooks, F., "No Silver Bullet: Essence and Accidents of Software Engineering." *Computer,* Vol. 20, No. 4, April 1987, pp. 10-19.

Caldiera, G., & Basili, V., "Identifying and Qualifying Reusable Software Components." *IEEE Computer,* Vol. 24, Issue 2, February 1991, pp. 61-70.

*Cantata++ v. 4.0.* Bath, England: IPL Information Processing Limited, 2004.

Carver, J., Jaccheri, L., Morasca, S., & Shull, R., "Issues in Empirical Studies with Students." *Technical Report MSU-060714,* July 14, 2006, pp. 1-46.

Carver, J., Jaccheri, L., Morasca, S., & Shull, R., "Issues in Using Students in Empirical Studies in Software Engineering Education." *Proceedings of the Ninth International Software Metrics Symposium,* September 2000, pp. 239-249.

Chikofsky, E. & Cross, J., "Reverse Engineering and Design Recovery: A Taxonomy." *IEEE Software*, Vol. 7. No. 1, January 1990, pp. 13-17.

Chidamber, S., & Kemerer, C., "A Metrics Suite for Object Oriented Design." *IEEE Transactions on Software Engineering,* June 1994, Vol. 20, No. 6, pp. 476-493.

Ciolkowski, M., Muthig, D., & Rech, J., "Using Academic Courses for Empirical Validation of Software Development Processes." *Proceedings of the 30th Euromicro Conference 2004*, pp. 354-361.

Clark, B., "Eight Secrets of Software Measurement." *IEEE Software,* Vol. 19, Issue 5, September-October 2002, pp.12-14.

Cohen, A., Doveh, E., & Eick, U., "Statistical Properties of the $r_{WG(J)}$ Index of Agreement." *Psychological Methods,* Vol. 6, No. 3, 2001, pp. 297-310.

Cohen, D., Lindvall, M., & Costa, P., “An Introduction to Agile Methods.” *Advances in Computer, Advances in Software Engineering,* 2004, Vol. 62, pp. 1-66.

Cohen, J., “Things I Have Learned (So Far).” *American Psychologist*, Vol. 15, Issue 12, Dec. 1990, pp. 1304-1312.

Compare It 3.5. http://www.grigsoft.com/wincmp3.htm#overview, [last accessed October 12, 2008].

Cook, T., & Campbell, D., *Quasi-Experimentation – Design and Analysis Issues for Field Settings.* Boston: Houghton Mifflin Company, 1979.

Coppick, J., & Cheatham, T., “Software Metrics for Object-Oriented Systems.” *Proceedings of the 1992 ACM Annual Conference on Communications CSC ’92,* pp. 317-322.

Cortina, J., “What is Coefficient Alpha? An Examination of Theory and Applications.” *Journal of Applied Psychology,* Vol. 78, No. 1, 1993, pp. 98-104.

Counsell, S., & Swift, S., “The Interpretation and Utility of Three Cohesion Metrics for Object-Oriented Design.” *ACM Transactions on Software Engineering and Methodology,* Vol. 15, No. 2, April 2006, pp. 123-149.

Dandashi, F., “A Method for Assessing the Reusability of Object-Oriented Code using a Validated Set of Automated Measurements.” *Proceedings of the 2002 ACM Symposium on Applied Computing, March* 2002, pp. 997 – 1003.

Devanbu, P., Karstu, S., Melo, W., & Thomas, W., “Analytical and Empirical Evaluation of Software Reuse Metrics.” *Proceedings of the 18th International Conference on Software Engineering,* 25-30 March 1996, pp. 189-199.

Dingsøyr, T., Dybå, T., & Abrahamsson, P., “A Preliminary Roadmap for Empirical Research on Agile Software Development,” *Agile 2008 Conference,* August 2008, pp. 83-94.

Dybå, T., “An Empirical Investigation of the Key Factors for Success in Software Process Improvement.” *IEEE Transactions on Software Engineering,* Vol. 31, No. 5, May 2005, pp. 410-424.

Dybå, T., & Dingsøyr, T., “Empirical Studies of Agile Software Development: A Systematic Review.” *Information and Software Technology,* Vol. 50, No. 9-10, August 2008, pp. 833-859.

Ebert, C., & Morschel, I., “Metrics for Quality Analysis and Improvement of Object-Oriented Software.” *Information and Software Technology,* Vol. 39, Issue 7, July 1997, pp. 497-509.

Elish, M., & Rine, D., “Indicators of Structural Stability of Object-Oriented Designs: A Case Study.” *Proceedings of the 29th Annual IEEE/NASA Software Engineering Workshop,* April 2005, pp. 183-192.

Elish, M., & Rine, D., “Investigation of Metrics for Object-Oriented Design Logical Stability.” *Proceedings of the Seventh European Conference on Software Maintenance and Reengineering 2003,* March 2003, pp. 193-200.

El Emam, K., “A Primer on Object-Oriented Measurement.” *Proceedings of the Seventh International Software Metrics Symposium 2001,* April 2001, pp. 185-187.

El Emam, K., Benlarbi, S., Goel, N., & Rai, S., “The Confounding Effect of Class Size on the Validity of Object-Oriented Metrics.” *IEEE Transactions on Software Engineering,* Vol. 27, No. 7, July 2001, pp. 630-650.

El Emam, K., Benlarbi, S., Goel, N., Melo, W., Lounis, H., & Rai, S., “The Optimal Class Size for Object-Oriented Software.” *IEEE Transactions on Software Engineering,* Vol. 28, No. 5, May 2002, pp. 494-509.

Erickson, J., Lyytinen, K., & Siau, K., “Agile Modeling, Agile Software Development, and Extreme Programming: The State of Research.” *Journal of Database Management,* Vol. 16, No. 4, 2005, pp. 88-100.

Esteva, J., & Reynolds, R., “Identifying Reusable Software Components by Induction.” *International Journal of Software Engineering and Knowledge Engineering,* Vol. 1, Issue 3, 1991, pp. 271-292.

Etzkorn, L. H., A *Metrics-Based Approach to the Automated Identification of Object-Oriented Reusable Software Components.* Doctoral Dissertation, The University of Alabama in Huntsville, 1997.

Etzkorn, L. H., & Davis C., “Automatically Identifying Reusable OO Legacy Code.” *Computer,* Vol. 30, No. 10, October 1997, pp. 66-71.

Etzkorn, L., Davis, C., & Li, W., "A Practical Look at the Lack of Cohesion in Methods Metric." *Journal of Object-Oriented Programming*, Vol. 11, No. 5, Sept. 1998, pp.27-34.

Etzkorn, L. H., Hughes, W. E., & Davis, C. G., “Automated Reusability Quality Analysis of OO Legacy Software.” *Information and Software Technology,* Vol. 43, No. 5, April 2001, pp. 295 – 308.

Etzkorn, L. H., Gholston, S. E., Fortune, J. L., Stein, C. E., Utley, D., Farrington, P. A., & Cox, G. W., “A Comparison of Cohesion Metrics for Object-Oriented Systems.” *Information and Software Technology,* Vol. 46, No. 10, August 2004, pp. 677-687.

Etzkorn, L. H., Gholston, S., & Hughes, W. E., “A Semantic Entropy Metric.” *Journal of Software Maintenance and Evolution*: *Research and Practice,* Vol. 14, No. 4, July/August 2002, pp. 293-310.

Fenton, N., “Software Measurement: A Necessary Scientific Basis.” *IEEE Transactions on Software Engineering,* Vol. 20, No. 3, March 1994, pp. 199-206.

Fenton, N., & Neil, M., “Software Metrics: Roadmap.” *Proceedings of the Conference on the Future of Software Engineering,* May 2000, pp. 359-370.

Fenton, N., & Neil, M., “Software Metrics: Successes, Failures and New Directions.” *The Journal of Systems and Software,* Vol. 47, No. 2, July 1999, pp. 149-157.

Fowler, M., *Refactoring: Improving the Design of Existing Programs.* Boston, MA: Addison Wesley, 1999.

France, R., Ghosh, S., Song, E., & Kim, D., “A Metamodeling Approach to Pattern-Based Model Refactoring.” *IEEE Software,* Vol. 20, Issue 5, September-October 2003, pp. 52-58.

Fraser, S., & Manci, D., “No Silver Bullet: Software Engineering Reloaded.” *IEEE Software,* Vol. 25, Issue 1, Jan-Feb. 2008, pp. 91-94.

Ganesan, K., Khoshgoftaar, T., & Allen, E., “Case-Based Software Quality Prediction.” *International Journal of Software Engineering and Knowledge Engineering,* Vol. 10, No. 2, April 2000, pp. 139-152.

García, F., Bertoa, M., Calero, C., Vallecillo, A., Ruíz, F., Piattini, M., & Genero, M., “Towards a Consistent Terminology for Software Measurement.” *Information and Software Technology,* Vol. 48, No. 8, August 2006, pp. 631-634.

Gavin, D., “What Does “Product Quality” Really Mean?” *Sloan Management Review,* Fall 1984, pp. 25-45.

Genest, C., & Wagner, C., “Further Evidence Against Independence Preservation in Expert Judgement Synthesis.” *Aequationes Mathematicae,* Vol. 32, No. 1, December 1987, pp. 74-86.

Grosser, D., Sahraoui, H., & Valtchev, P., “An Analogy-based Approach for Predicting Design Stability of Java Classes.” *Proceedings of the Ninth International Software Metrics Symposium,* 2003, pp. 252-262.

Grosser, D., Sahraoui, H., & Valtchev, P., “Predicting Software Stability Using Case-Based Reasoning.” *Proceedings of the 17th IEEE International Conference on Automated Software Engineering,* September 2002, pp. 295-298.

Gyimóthy, T., Ferenc, R., & Siket, I., "Empirical Validation of Object-Oriented Metrics on Open Source Software for Fault Prediction." *IEEE Transactions on Software Engineering,* Vol. 31, No. 10, October 2005, pp. 897 – 910.

Harrison, R., Counsell, S., & Nithi, R., "An Evaluation of the MOOD Set of Object-Oriented Software Metrics." *IEEE Transactions on Software Engineering,* Vol. 24, No. 6, June 1998, pp. 491-496.

Harrison, W., "An Entropy-Based Measure of Software Complexity." *IEEE Transactions on Software Engineering,* Vol. 18, No. 11, November 1992, pp.1025-1029.

Heinecke, H., Noack, C., & Schweizer, D., "Software Reuse in Agile Projects." *4th International Conference on Object-Oriented and Internet-based Technologies, Concepts, and Applications for a Networked World (NetObjectDays '03)*, 2003.

Highsmith, J., "What is Agile Software Development?" *CrossTalk, The Journal of Defense Software Engineering,* October 2002, pp. 4 – 9.

Highsmith, J., & Cockburn, A., "Agile Software Development: The Business of Innovation." *IEEE Computer,* Vol. 34, Issue 9, September 2001, pp. 120-121.

Hislop, G. W., Lutz, M. J., Naveda, J. Fernando, McCracken, W. M., Mead, N. R., & Williams, L. A., "Integrating Agile Practices into Software Engineering Courses." *Computer Science Education, 2002*, Vol. 12, No. 3, pp. 169-185.

Hitz, M., & Montazeri, B., "Chidamber and Kemerer's Metrics Suite: A Measurement Theory Perspective." *IEEE Transactions on Software Engineering,* Vol. 22, No. 4, April 1996, pp. 267-271.

Höst, M., Regnell, B., & Wohlin, C., "Using Students as Subjects – A Comparative Study of Students and Professionals in Lead-Time Impact Assessment." *Journal of Empirical Software Engineering,* November 2000, Vol. 5, No. 3, pp. 201-214.

Huo, M., Verner, J., Zhu, L., & Barbar, M., "Software Quality and Agile Methods." *Proceedings of the 28th Annual International Computer Software and Applications Conference,* 2004, Vol. 1, pp. 520-525.

*ISO/IEC 9126: Information technology – Software Product Evaluation – Quality characteristics and guidelines for their use*. Geneva: International Organization for Standardization, 2001.

James, L., Demaree, R., & Wolf, G., "Estimating Within-Group Interrater Reliability With and Without Response Bias." *Journal of Applied Psychology,* Vol. 69, No. 1, 1984, pp. 85-98.

Jeffries, R., Anderson, A., & Henrickson, C., *Extreme Programming Installed*. Boston, MA: Addison-Wesley, 2001.

Jeon, S., Lee, J., & Bai, D., “An Automated Refactoring Approach to Design Pattern-Based Program Transformations in Java Programs.” *Proceedings of the Ninth Asia-Pacific Software Engineering Conference,* December 2002, pp. 337-345.

Kähkönen, T., & Abrahamsson, P., “Digging into the Fundamentals of Extreme Programming.” *Proceedings of the 29th Euromicro Conference 2003,* 1-6 September 2003, pp. 273-280.

Kalagiakos, P., “The Non-Technical Factors of Reusability.” *Euromicro Conference, 2003 Proceedings 29th,* 1-6 September 2003, pp. 124-127.

Kataoka, Y., Ernst, M., Griswold, W., & Notkin, D., “Automated Support for Program Refactoring using Invariants.” *Proceedings of the IEEE International Conference on Software Maintenance,* November 2001, pp. 736-743.

Keeney, R. L., & Von Winterfeldt, D., “On the Uses of Expert Judgment on Complex Technical Problems.” *IEEE Transactions on Engineering Management*, Vol. 36, No. 2, May 1989, pp. 83-86.

Knoernschild, K., “The Agile Matrix.” *Agile // Journal: An Agile Business Publication,* www.agilejournal.com , 2006. [last accessed September 18, 2008].

Kim, Yongbeom, “A Method for the Classification and Retrieval Problem of Reusable Software Resources.” *Information Processing and Management,* Vol. 33, No. 4, 1997, pp. 513 – 522.

Kim, Yongbeom, & Stohr, E.A., “Software Reuse: Issues and Research Directions.” *Proceedings of the Twenty-Fifth Hawaii International Conference on System Science,* Vol. 4, January 1992, pp. 612 – 623.

Kitchenham, B., Linkman, S., & Law, D., “DESMET: A Methodology for Evaluating Software Engineering Methods and Tools.” *Computing and Control Engineering Journal,* Vol. 8, Issue 3, June 1997, pp. 120-126.

Kitchenham, B., & Pfleeger, S., “Software Quality: The Elusive Target.” *IEEE Software,* Vol. 13, Issue 1, pp. 12-21.

Kitchenham, B., Pfleeger, S., & Fenton, N., “Towards a Framework for Software Measurement Validation.” *IEEE Transactions on Software Engineering,* Vol. 21, No. 12, December 1995, pp. 929-944.

Kitchenham, B., Pfleeger, S., Pickard, L., Jones, P., Hoaglin, D., Emam, E., & Rosenberg, J., “Preliminary Guidelines for Empirical Research in Software Engineering.” *IEEE Transactions on Software Engineering,* Vol. 28, No. 8, August 2002, pp. 721-734.

Kivi, J., Haydon, D., Hayes, J., Schneider, R., & Succi, G. “Extreme Programming: A University Team Design Experience.” *2000 Conference on Electrical and Computer Engineering,* Vol. 2, March 2000, pp. 816-820.

Koru, A., & Tian, J., “An Empirical Comparison and Characterization of High Defect and High Complexity Modules.” *The Journal of Systems and Software,* Vol. 67, Issue 3, September 2003, pp. 153-163.

Kosar, T., Mernick, M., & Žumar, V., “JART: Grammar Based Approach to Refactoring.” *Proceedings of the 28th Annual International Computer Software and Applications Conference,* September 2004, Vol. 1, pp. 502-507.

Kuppuswami, S., Vivekananadan, K., Ramaswamy, P., & Rodrigues, P., “The Effects of Individual XP Practices on Software Development Effort.” *ACM SIGSOFT Software Engineering Notes,* Vol. 28, Issue 6, November 2003, pp. 1-6.

Larman, C. & Basili, V.R., “Iterative and Incremental Development: A Brief History.” *IEEE Software,* Vol. 36, Issue 6, June 2003, pp. 47-56.

Layman, L., “Empirical Investigation of the Impact of Extreme Programming Practices on Software Projects.” *Conference on Object Oriented Programming System Languages and Applications,* October 2004, pp. 328-329.

Li, W., Etzkorn, L., Davis, C., & Talburt, J., “An Empirical Study of Object-Oriented System Evolution.” *Information and Software Technology,* Vol. 42, Issue 6, April 2000, pp. 373-381.

Li, W., “Another Metric Suite for Object-Oriented Programming.” *The Journal of Systems and Software,* Vol. 44, Issue 2, February 1998, pp. 155-162.

Li, M., & Smidts, C. S., “A Ranking of Software Engineering Measures Based on Expert Opinion.” *IEEE Transactions on Software Engineering*, Vol. 29, No. 9, September 2003, pp. 811-824.

Mambella, E., Ferrari, R., De Carli, F., & Lo Surdo, A., “An Integrated Approach to Software Reuse Practice.” *ACM SIGSOFT Software Engineering Notes Proceedings of the 1995 Symposium on Software Reusability,* Vol. 20, Issue SI, August 1995, pp.63-71.

Mansfield, E., & Helms, B., “Detecting Multicollinearity.” *The American Statistician,* Vol. 36, No.3, August 1982, pp. 158-160.

Marcus, A., & Poshyvanyk, D., “The Conceptual Cohesion of Classes.” *Proceedings of the 21st International Conference on Software Maintenance,* September 2005, pp.133-142.

Maruyama, K., & Shima, K., “Automatic Method Refactoring Using Weighted Dependence Graphs.” *Proceedings of the 1999 International Conference on Software Engineering,* May 1999, pp. 236-245.

Mattsson, M., & Bosch, J., “Characterizing Stability in Evolving Frameworks.” *Proceedings of Technology of Object-Oriented Languages and Systems 1999,* June 1999, pp. 118-130.

Mayer, T., & Hall, T., “Measuring OO Systems: A Critical Analysis of the MOOD Metrics.” *Proceedings of the Technology of Object-Oriented Languages and Systems,* June 1999, pp. 108-117.

Mens, T., & Tourwé, T., “A Survey of Software Refactoring.” *IEEE Transactions on Software Engineering*, Vol. 30, No. 2, February 2004, pp. 126-139.

Meyer, B., “The Reusability Challenge.” *Computer,* Vol. 29, Issue 2, February 1996, pp. 76-78.

Meyer, B., “The Role of Object-Oriented Metrics.” *Computer,* Vol. 31, Issue 11, November 1998, pp. 123-127.

Mitchell, A., & Power, J., “Run-Time Cohesion Metrics: An Empirical Investigation.” *Proceedings of the International Conference on Software Engineering Research and Practice, SERP’04,* 2004, pp. 532-537.

Moser, R., Sillitti, A., Abrahamsson, P., & Succi, G., “Does Refactoring Improve Reusability?” *Proceedings of the International Conference on Software Reuse*, 2006, pp. 287-297.

Müller, M., “Two Controller Experiments Concerning the Comparison of Pair Programming to Peer Review.” *The Journal of Systems and Software,* Vol. 78, Issue 2, November 2005, pp. 166-179.

Müller, M., & Tichy, W., “Case Study: Extreme Programming in a University Environment.” *Proceedings of the 23rd International Conference on Software Engineering,* May 2001, pp. 537-544.

Mumpower, J., & Stewart, T., “Expert Judgement and Expert Disagreement.” *Thinking and Reasoning,* Vol. 2, No. 2/3, 1996, pp. 191-211.

Nazareth, D. L., & Rothenberger, M. A., “Assessing the cost-effectiveness of software reuse: A model for planned reuse.” *The Journal of Systems and Software,* Vol. 73, Issue 2, October 2004, pp. 245-255.

Noble, J., Marshall, Stuart, Marshall, Stephen, & Biddle, R., “Less Extreme Programming.” *Conferences in Research and Practice in Information Technology,* January 2004, Vol. 30, pp. 217-226.

Nerur, S. & Balijepally, V., “Theoretical Reflections on Agile Development Methodologies.” *Communications of the ACM,* Vol. 50, No. 3, March 2007, pp. 79-83.

Nunnally, J. C., *Psychometric Theory First Edition*. New York: McGraw-Hill, 1967.

Olague, H., “Assessing Maintainability: Information Theory, Metrics, and Iterative Software.” Doctoral Dissertation, University of Alabama in Huntsville, 2006.

Olague, H., Etzkorn, L., Gholston, S., & Quattlebaum, S., “Empirical Validation of Three Software Metrics Suites to Predict Fault-Proneness of Object-Oriented Classes Developed Using Highly Iterative or Agile Software Development Processes.” *IEEE Transactions on Software Engineering,* Vol. 33, No. 6, June 2007, pp. 402-419.

Olague, H., Etzkorn, L., Li, W., & Cox, G., “Assessing Design Instability in Iterative (agile) Object-Oriented Projects.” *Journal of Software Maintenance and Evolution Research and Practice,* Vol. 18, Issue 4, July/August 2006, pp. 237-266.

Paulk, M.C., “Agile Methodologies and Process Discipline.” *CrossTalk, The Journal of Defense Software Engineering,* October 2002, pp 15-18.

Pfleeger, S., & Atlee, J., *Software Engineering, Theory and Practice Third Edition.* Upper Saddle River, New Jersey: Pearson Prentice Hall, 2006.

Pfleeger, S., Jeffery, R., Curtis, B., & Kitchenham, B., “Status Report on Software Measurement.” *IEEE Software,* Vol. 14, Issue 2, March/April 1997, pp. 33-43.

Poels, G., & Dedene, G.,”Distance-Based Software Measurement: Necessary and Sufficient Properties for Software Measures.” *Information and Software Technology,* Vol. 42, Issue 1, January 2000, pp. 35-46.

Poulin, J. S., “Measuring Software Reusability.” *Advances in Software Reusability 1994, Proceedings of the Third International Conference on Software Reuse,* 1-4 Nov. 1994*,* pp. 126-138.

Pressman, R., *Software Engineering a Practitioner’s Approach, Sixth Edition.* Boston, MA: McGraw-Hill, 2005.

Prieto-Díaz, R., “Status Report: Software Reusability.” *IEEE Software,* Vol. 10, Issue 3, May 1993, pp. 61-66.

Prieto-Díaz, R., & Freeman, P., “Classifying Software for Reusability.” *IEEE Software,* Vol.1, Issue 1, January 1987, pp. 6-16.

Price, M. W., & Demurjian, S. A., “Analyzing and Measuring Reusability in Object-Oriented Design.” *Proceedings of the 12th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications,* Vol. 32, Issue 10, 1997, pp. 22-33.

Rakitin, S., “Manifesto Elicits Cynicism.” *IEEE Computer,* Vol. 34, No. 12, December 2001, p. 4.

Raley, M. A., & Etzkorn, L. H., “Entropy Metrics-Based Approach to Risk Analysis in the Maintenance Phase of Large-Scale Computer Systems.” *The 2004 International Multi-Conference in Computer Science and Computer Engineering,* June 2004, 883-889.

Repenci, J., “Creativyst Software Stability Ratings,” http://www.creativyst.com/Doc/Std/StableSW/StabSW.htm [last accessed September 18, 2008].

Rising, L., & Janoff, N., “The Scrum Software Development Process for Small Teams.” *IEEE Software,* Vol. 17, Issue 4, July-August 2000, pp. 26-32.

Roden, P., Etzkorn, L., Virani, S., Messimer, S., & Vinz, B., “A Validation of Stability Metrics.” *Proceedings of the International Conference on Software Engineering (SEA 2007),* November 2007, pp. 57-65.

Roden, P., Virani, S., Etzkorn, L., & Messimer, S., “An Empirical Study of the Relationship of Stability Metrics and the QMOOD Quality Models Over Software Developed Using Highly Iterative or Agile Software Processes.” *Seventh IEEE International Working Conference on Source Code Analysis and Manipulation,* September 2007, pp. 171-179.

Sandri, S., Dubois, D., & Kalfsbeek, H., “Elicitation, Assessment, and Pooling of Expert Judgments Using Possibility Theory.” *IEEE Transactions on Fuzzy Systems,* Vol. 3, No. 3, August 1995, pp. 313-335.

Santos, J., “Cronbach’s Alpha: A Tool for Assessing the Reliability of Scales.” *Journal of Extensions,* Vol. 37, No. 2, April 1999, pp. 1-5.

Scott, W., “Extreme Programming Turning the World Upside Down.” *Computing & Control Engineering Journal,* Vol. 14, Issue 3, June/July 2003, pp. 18-23.

Schach, S., & Yang, X., “Metrics for Targeting Candidates for Reuse: An Experimental Approach.” *Proceedings of the 1995 ACM Symposium on Applied Computing,* 1995, pp. 379-383.

Schroeder, M., “A Practical Guide to Object-Oriented Metrics.” *IT Professional,* Vol. 1, Issue 6, November-December 1999, pp. 30-36.

Seskin, D., *Handbook of Parametric and Nonparametric Statistical Procedures*. Boca Raton: Chapman & Hall/CRC, 2004.

Shanteau, J., “Competence in Experts: The Role of Task Characteristics.” *Organizational Behavior and Human Decision Processes*, Vol. 53, 1992, pp. 252-266.

Sjøberg, D., Anda, B., Arisholm, E., Dybå, T., Jørgensen, M. Karahasanovic, A., Koren, E., & Vokác, M., “Conducting Realistic Experiments in Software Engineering.” *Proceedings of the 2002 International Symposium on Empirical Software Engineering,* October 2002, pp. 17-26.

Sjøberg, D., Dybå, T., & Jørgensen, M., “The Future of Empirical Methods in Software Engineering Research,” *Future of Software Engineering 2007,* May 2007, pp. 358-378.

Smith, M., & Sodhi, J., “Marching Towards a Software Reuse Future.” *ACM Ada Letters,* Vol. XIV, Number 6, November/December 1994, pp. 62-72.

Smith, S., & Stoecklin, S., “What We Can Learn From Extreme Programming.” *Journal of Computing Science in Colleges,* Vol. 17, Issue 2, December 2001, pp. 144-151.

Snyder, A., “Encapsulation and Inheritance in Object-Oriented Programming Languages,” *Proceedings of OOPSLA ’86,* September 1986, pp. 38-45.

*Synchronizer*® *9.1 XL User Manual*. Zurich, Switzerland: XL Consulting GmbH, 2006.

Tang, M., Kao, M., & Chan, M., “An Empirical Study on Object-Oriented Metrics.” *Proceedings of the Sixth International Software Metrics Symposium,* November 1999, pp. 242-249.

Tichelaar, S., Ducasse, S., Demer, S., & Nierstrasz, O., “A Meta-model for Language-Independent Refactoring.” *Proceedings of the International Symposium on Principles of Software Evolution,* November 2000, pp. 154-164.

Tonu, S., Ashkan, A., & Tahvildari, L., “Evaluating Architectural Stability Using a Metrics-Based Approach.” *Proceedings of the 10th European Conference on Software Maintenance and Reengineering,* March 2006, pp. 261-270.

Torres, W. R., & Samadzadeh, M. H., “Software Reuse and Information Theory Based Metrics.” *Proceedings of the 1991 Symposium on Applied Computing,* April 1991, pp. 437-446.

Washizaki, H., Yamamoto, H., & Fukazawa, Y., “A Metrics Suite for Measuring Reusability of Software Components.” *Proceedings of the Ninth International Software Metrics Symposium,* September 2003, pp. 211-223.

Weyuker, E., “Evaluating Software Complexity Measures.” *IEEE Transactions on Software Engineering,* Vol. 14, No. 9, September 1988, pp. 1357-1365.

Williams , L., “Integrating Pair Programming into a Software Development Process.” *Proceeding of the 14th Conference on Software Engineering Education and Training,* February 2001, pp. 27-36.

Williams, L., “The XP Programmer: The Few-Minutes Programmer.” *IEEE Software,* Vol. 20, Issue 3, May/June 2003, pp. 16-20.

Williams, L. A., & Kessler, R. R., “All I Ever Needed to Know About Pair Programming I Learned in Kindergarten.” *Communications of the ACM,* Vol. 43, No. 5, May 2000, pp. 108-114.

Williams, L. A., & Kessler, R. R., “Experimenting with Industry’s Pair-Programming Model in the Computer Science Classroom.” *Journal of Computer Science Education* Vol. 10, No. 4, December 2000.

Williams, L. A., Kessler, R. R., Cunningham, W., & Jeffries, R., “Strengthening the Case for Pair Programming.” *IEEE Software,* Vol. 17, Issue 4, July 2000, pp. 19-25.

Williams, L. A., & Upchurch, R. L., “In Support of Student Pair-Programming.” *ACM SIGCSE Bulletin Proceedings of the Thirty-second SIGCSE Technical Symposium on Computer Science Education,* February 2001, Vol. 33, Issue 1, pp. 327 – 331.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M., Regnell, B., & Wesslén, A., *Experimentation in Software Engineering: An Introduction.* Boston: Kluwer Academic, 2000.

Wolf, M., Bowyer, K., Gotterbarn, D., & Miller, K., “Open Source Software: Intellectual Challengers to the Status Quo.” *SIGCSE’02*, February 27-March 3, 2002, pp. 317-318.

Wood, W., & Kleb, W., “Exploring XP for Scientific Research.” *IEEE Software,* Vol. 20, Issue 3, May-June 2003, pp. 30-36.

Zannier, C., Melnik, G., & Maurer, F., “On the Success of Empirical Studies in the International Conference on Software Engineering.” *Proceedings of the 28th International Conference on Software Engineering,* May 2006, pp. 341-350.

Zuse, Horst, “Foundations of Object-Oriented Software Measures.” *Proceedings of the 3rd International Software Metrics Symposium,* March 1996, pp. 75-88.