DEFINITIONS AND VALIDATIONS OF METRICS OF INDIRECT PACKAGE COUPLING
IN AN AGILE, OBJECT-ORIENTED ENVIRONMENT

A dissertation submitted
to Kent State University in partial
fulfillment of the requirements for the
degree of Doctor of Philosophy

by

Saleh A. Almugrin

August 2015
© Copyright
All rights reserved
Except for previously published materials
Dissertation written by
Saleh A. Almugrin
B.S., King Saud University, KSA 1996
M.S., King Saud University, KSA, 2003
PhD, Kent State University, USA, 2015

Approved by

Austin Melton, Professor, Ph.D., Computer Science, Doctoral Advisor

Cheng-Chang Lu, Professor, Ph.D., Computer Science

Ye Zhao, Associate Professor, Ph.D., Computer Science

Pratim Datta, Associate Professor, Ph.D., Management and Information Systems

Mohammed Khan, Professor, Ph.D., Mathematical Science

Accepted by

Javed I. Khan, Professor, Ph.D., Chair, Department of Computer Science

James L. Blank, Professor, Ph.D., Interim Dean, College of Arts and Sciences
Abstract

Object-Oriented systems are dynamic, evolve over time, and have to be continuously updated, or they become outdated and irrelevant. Most of the cost during a software system lifecycle is dedicated to support the system's evolution and changes. It is clearly recognized that it is important to understand, assess, and manage the static structure of the software system. Although classes and their relationships represent a very commonly studied static structure of the software system, it may be difficult for the maintainers to understand the system and the communication between its subsystems, as the number of classes increases if the maintainers focus too heavily on classes and their relationships. The maintainers may easily be overwhelmed by the large number of classes and their interdependencies in such large systems. A way to address this complexity problem is to view the system at a higher level of abstraction by grouping classes into more coarsely grained entities, e.g., packages, and then by looking at their interdependencies at this higher level of abstraction. Moreover, organizing a software system into packages can facilitate the development, maintenance, and reusability of software components.

However, excessive coupling between packages may hinder package reusability and cause damage to the system design, which decreases the system maintainability and testability. Therefore, the inter-package coupling should be kept minimal, and unnecessary coupling should be avoided. Most researchers have concentrated on direct dependency coupling, which is coupling between similar software units that are directly related to each other. There has been comparatively little study about indirect
dependency, which includes coupling between units that are not directly related to each other. This research was originally inspired by R. C. Martin’s seminal work, presented in his book titled “Agile Software Development: Principles, Patterns, and Practices,” which discusses packaging of agile-developed, object-oriented software. Martin’s metrics [56] are well-known package design metrics that can be used in the early stages of software development. Although Martin emphasizes dependency and coupling, his metric suite only measures direct package coupling. The author believes that the accuracy and usefulness of this metric suite could be improved by including a more full and complete scope of what is being measured. In this work, Martin's metric suite is enhanced by an approach that manages packaging in object-oriented software development by analyzing the direct and indirect dependencies of all packages. In addition, we present an experimental study to validate the modified global metrics by showing their relationship to maintainability and testability, and then we construct prediction models for these two external quality attributes. The study results indicate that the new metrics are very promising and lead to improved results. Given the importance of Martin's metrics, it is expected that there will be significant applications for the new metrics.
TABLE OF CONTENTS

ABSTRACT .................................................................................................................... III
LIST OF FIGURES ........................................................................................................ IX
LIST OF TABLES ........................................................................................................... X
DEDICATION .................................................................................................................. XII
ACKNOWLEDGEMENTS ............................................................................................... XIII

CHAPTER 1 INTRODUCTION ......................................................................................... 1
  1.1 Overview ................................................................................................................ 1
  1.2 Research Framework ............................................................................................ 5
  1.3 Organization .......................................................................................................... 7

CHAPTER 2 BACKGROUND AND RELATED WORKS .............................................. 8
  2.1 The Concept of coupling ...................................................................................... 8
  2.2 Object-Oriented Metrics ..................................................................................... 8
    2.2.1 Package level metrics ................................................................................. 10
    2.2.2 Martin’s metrics ......................................................................................... 12
    2.2.3 Martin’s metrics in Literature .................................................................... 17
  2.3 Software Evolution ............................................................................................... 19
    2.3.1 Lehman’s Laws of Software Evolution ..................................................... 19
    2.3.2 Object Oriented Software Evolution ......................................................... 20
  2.4 Software Measurement ....................................................................................... 22
    2.4.1 Introduction ................................................................................................. 22
5.1.3 Packaging ........................................................................................................... 53

5.2 Instability and Abstractness .................................................................................. 55

5.3 The proposed method of calculating Instability: Dual Ranking ......................... 56

5.3.1 Introduction ........................................................................................................ 56

5.3.2 The proposed Dual Ranking Instability Metric Steps .................................... 63

5.3.3 The proposed method of calculating Package’s Abstraction ........................... 74

5.3.4 The proposed method of calculating a Package’s Normalized Distance ...... 76

5.4 Motivational Example .......................................................................................... 78

CHAPTER 6 SOFTWARE MAINTAINABILITY AND TESTABILITY .............. 90

6.1 Introduction .......................................................................................................... 90

6.2 Maintainability ..................................................................................................... 90

6.3 Testability ............................................................................................................. 91

CHAPTER 7 THEORETICAL VALIDATION ............................................. 94

7.1 Introduction .......................................................................................................... 94

7.2 Property-based measurement framework ............................................................ 94

7.3 DISTANCE framework ........................................................................................ 98

CHAPTER 8 EXPERIMENTAL DESIGN ............................................... 101

8.1 Introduction .......................................................................................................... 101

8.2 Data set ................................................................................................................. 103

8.3 Tools ..................................................................................................................... 105

8.4 Maintenance and Test metrics ............................................................................ 107

8.5 Research Hypotheses and Questions .................................................................... 108

vii
8.6 Expected Result.................................................................................................................. 114

CHAPTER 9 EMPIRICAL ANALYSIS AND RESULTS ........................................... 119

9.1 Introduction ......................................................................................................................... 119

9.2 Maintainability ................................................................................................................. 119

  9.2.1 Descriptive statistics.................................................................................................... 120

A. 121

  9.2.2 Statistical Analysis ..................................................................................................... 121

  9.2.3 Regression Analysis ................................................................................................. 133

9.3 Testability .......................................................................................................................... 138

  9.3.1 Descriptive statistics................................................................................................ 138

  9.3.2 Statistical Analysis ................................................................................................... 140

  9.3.3 Regression Analysis ................................................................................................. 151

CHAPTER 10 THREATS TO VALIDITY ................................................................. 155

10.1 Introduction ....................................................................................................................... 155

10.2 Internal Validity ............................................................................................................... 156

10.3 External Validity ............................................................................................................... 157

10.4 Construct Validity ........................................................................................................... 158

10.5 Content validity ............................................................................................................... 160

CHAPTER 11 CONCLUSION AND FUTURE WORK .......................................... 161

REFERENCES ......................................................................................................................... 166
LIST OF FIGURES

Figure 2.1: The Instability-Abstractness graph and the main sequence .............. 15
Figure 2.2: Zones of pain and uselessness on Instability-Abstractness graph ......... 17
Figure 3.1: Cost of change for Agile and Traditional development methods......... 36
Figure 3.2: XP Programming Life Cycle .......................................................... 38
Figure 4.1: Typical dependency diagram of a procedural Architecture ................ 44
Figure 4.2: Typical dependency diagram of an OO Architecture ....................... 45
Figure 4.3: The first solution to break a dependency cycle .............................. 49
Figure 5.1: The Java code for the indirect dependency example ....................... 58
Figure 5.2: The Package dependency diagram for the indirect dependency example .......................................................... 59
Figure 5.3: The Dependency-Responsibility diagram ....................................... 70
Figure 5.4: Package sit on (0, 0) and its distance from stability and Instability points ......................................................................................................................... 71
Figure 5.5: Package sit on (0.5, 0.5) and its distance from stability and Instability points ......................................................................................................................... 72
Figure 5.6: the Instability-Abstractness diagram and the zones of Pain and Uselessness ......................................................................................................................... 77
Figure 5.7: The package dependency graph for the first motivational example ..... 79
Figure 5.8: The package dependency graph for the second motivational example .. 82
Figure 5.9: The package dependency graph for the third motivational example .... 86
LIST OF TABLES

Table 5.1. Comparison between Martin’s instability and the proposed instability metric for the packages in the first motivational example ................................................................. 81

Table 5.2. Comparison between Martin’s instability and the proposed instability metric for the packages in the second motivational example ......................................................... 85

Table 5.3. Comparison between Martin’s instability and the proposed instability metric for the packages in the third motivational example ..................................................................... 89

Table 7.1: Theoretical properties of InRank and OutRank ................................................................. 99

Table 8.1: The descriptions of the studied systems ............................................................................. 103

Table 8.2: Maintainability Studied Systems details ............................................................................. 104

Table 8.3: Testability Studied Systems details .................................................................................... 104

Table 9.1. Means and Standard Deviation of the Variables Used in Maintainability Analyses .............................................................................................................................................. 120

Table 9.2 Correlation of the studied Metrics For Camel2.0.0 .............................................................. 123

Table 9.3 Correlation of the studied Metrics For JEdit 4.5.0 ............................................................... 124

Table 9.4 Correlation of the studied Metrics For JHotDraw 7.5.1 ......................................................... 125

Table 9.5 Correlation of the studied Metrics For Tomcat 7.0.6 ............................................................ 126

Table 9.6 Spearman’s Rho Correlations for Maintainability Analysis ................................................. 128

Table 9.7 Correlations of Maintainability Metrics .................................................................................. 129

Table 9.8 Maintainability propositions summary .................................................................................. 132
Table 9.9 Means and Standard Deviations of the Data Used in Maintainability Regression Analysis.............................................................. 134

Table 9.10 Pearson Correlations among the Variables Used In the Maintainability Regression Analysis ........................................................................................................ 135

Table 9.11 Summary of the Model Predicting The Log Of Package Revision (LogRevision) ........................................................................................................................................ 136

Table 9.12 Summary of the Model Predicting The Log Of The Number of Revised LOC (LogRLOC) ........................................................................................................................................ 137

Table 9.13 Means and Standard Deviation of the Variables Used in Testability Analyses 139

Table 9.14 Correlation of the studied Metrics For Camel 2.0.0 ........................................... 142

Table 9.15 Correlation of the studied Metrics For Tomcat 7.0.6 ....................................... 143

Table 9.16 Correlation of the studied Metrics For Hadoop 2.2.0 ...................................... 144

Table 9.17 Correlation of the studied Metrics for Synapse 2.1.0 ...................................... 145

Table 9.18 Correlation of the studied Metrics for Ant 1.92 ............................................. 146

Table 9.19 Spearman’s Rho Correlations for Maintainability Analysis ............................. 148

Table 9.20 Testability propositions summary ..................................................................... 150

Table 9.21 Means and Standard Deviations of the Data Used in Testability Regression Analysis .......................................................................................................... 152

Table 9.22 Pearson Correlations among the Variables Used In the Testability Regression Analysis ............................................................................................................. 153

Table 9.23 Summary of the Model Predicting the Log of the Number of LOC of Test Package (LogTLOC) ........................................................................................................ 154
DEDICATION

This dissertation is dedicated to my always encouraging, ever faithful parents, my great kind-hearted parents-in-law, my brilliant and outrageously loving and supportive wife, Nawal, our sweet, and lovely beautiful sons, Ahmed, Shahad, Layan, Mohammed, Reema and Abdullah Almugrin.

Thank you for your love, support, and encouragement...
ACKNOWLEDGEMENTS

I gratefully acknowledge the contributions of:

My supervisor, Dr. Austin Melton, who consistently provided encouragement, advice and warped humor throughout this research. A special thanks for his countless hours of reflecting, reading, and most of all patience throughout the entire process. His ideas are woven inextricably into this research.

My committee members who were more than generous with their expertise and precious time. Thank you Dr. Cheng-Chang Lu, Dr. Ye Zhao, Dr. Pratim Datta, and Dr. Mohammed Khan for agreeing to serve on my committee.

My school division, Computer Science Department, for allowing me to conduct my research and providing any assistance requested.

My lovely sons, Ahmed, Shahad, Layan, Mohammed, Reema and Abdullah, who have never known a father free from the obsession of writing a dissertation for the last four years.

My lovely wife Nawal, above all, who supported me and allowed me to direct my time and energy away from her for so long.

Thank you,

Saleh A. Almugrin

August 2015, Kent, Ohio
CHAPTER 1

Introduction

1.1 Overview

Many software metrics apply to software implementation (code) when the product is completed or nearly completed. However, by this time, it is difficult to restructure a system. It is recognized that mistakes committed and poor choices made during the design phase result in the most costly and inflexible problems [99]. In other words, detecting a design’s failure in the later stages can decrease software qualities, including maintainability and testability. Software design is one of the most important activities in the system development life cycle, and is vital for the successful implementation of software systems that are maintainable and testable. It involves comparing design choices, finding alternatives, and choosing the best choice. Choosing one of the alternatives without immediate feedback is very difficult, since there is no or little knowledge about its consequences. It would be much easier for the designer if there were objective measures that could help him/her to pinpoint strengths and weaknesses for different design choices. Therefore, design metrics can be an important part of the software development process.

Evolving over time during maintenance with the modification, addition, and removal of new classes, methods, functions, and dependencies, software may gradually lose its quality, i.e., maintainability and testability. Although a new software system may satisfy its requirements, the requirements are dynamic, and thus, the system must change. Therefore, as stated by Lehman [50], Object-Oriented systems have to be continuously updated or become outdated and irrelevant. Two of Lehman’s laws [50, 48, and 51] stated that software must continuously evolve to stay useful, and that this evolution is accompanied by an increase of complexity. Therefore,
these systems become difficult to be analyzed and maintained. Hence, most of the cost during a software system lifecycle is dedicated to support its evolution [27]. Object-Oriented software systems evolve over time with the modification, addition, and removal of new classes, methods, functions, and dependencies. Also, new unanticipated requirements cannot be predicted, because they are driven by the market or emerging technologies. For example, some new dependencies between classes have emerged, and some classes may not be placed in the right packages which lead to a high coupling and low cohesive systems. Therefore, the class dependency management is very challenging. It is clearly recognized that it is important to understand, assess, and manage the static structure of the software system. Usually, classes and their relationships represent the static structure of the software system.

However, as the number of classes increases, it will be very difficult for the maintainers to understand the system and the communication between its subsystems. Class-level dependency relationship diagrams of large object-oriented programs has a very large number of classes (nodes) and dependency relationships (edges) which makes it very hard for software designers to understand such complex structure. They can easily be overwhelmed by the large number of classes and their interdependencies in such large systems. A common way to address the complexity problem is to view the system at higher level of abstraction by grouping classes into coarse-grained entities, i.e., packages, and then looking at their interdependencies at a higher level of abstraction. Hence, modularization at the package level facilitates understanding by dividing the application into categories containing related classes that are easier to maintain. As stated by Martin [56], software designers organize classes into packages because of the problem of complexity of large object-oriented software systems. Moreover, organization into packages facilitates organizing the development teams as well as reusing of software packages. A package
provides and requires services from other packages, and it is a unit of reuse and release [56]. Packaging is the organization of classes into multiple packages. Good packaging has been claimed to ease the understanding, maintenance, testing, and evolution of software systems [24, 25, 56, and 61].

Packages of the large system have a high number of interdependent classes that communicate with each other to perform the required actions. It is normal that classes communicate with other classes residing inside and outside of their own packages. Therefore, packages become more complicated and risky. It is important for maintainers to recognize package dependencies before applying any changes, as making changes to a package may impact the entire system. Of course, modifying some packages may have a larger impact on the system than others. As a consequence, the packaging degrades as the system evolves. Excessive coupling between packages prevents package reusability, and it also causes damage to the system design which decreases the system maintainability and understandability. The more independent package can be easily reused in other systems. To improve packaging, the inter-package coupling should be kept to a minimum, and the bad coupling should be avoided.

Most researchers have measured only package direct dependency that is, coupling between packages that have some direct dependency relationship. There has been comparatively little study about package indirect dependency that is, coupling between packages that have no direct dependency relationship. For example, Briand et al. [13] closely discussed thirty coupling metrics of which only two metrics measure some form of indirect coupling. They stated that “To account for indirect coupling, we need only use the transitive closure of that relation.” Package indirect coupling is a special form of coupling that manifests between two seemingly unrelated packages through a hidden dependency relationship. For instance, a ripple effect is one of the
most significant maintenance problems that cannot be solved properly without considering package indirect coupling. A ripple effect is a situation when one modification causes a cascading effect on the dependent entities (packages) along the dependency relationship path. The larger the transitive dependencies would be, the more challenging to keep the changes consistent with each other’s packages without mistakes.

In large object-oriented software architectures, we need to have an easy way to assess the packaging of the system and to help the maintainers to manage it by detecting undesirable package dependencies which could help create structural problems. Object-oriented design principles help in the designing, building, and maintaining of robust, maintainable, and testable software systems. Following these principles during all software phases can help assure software effectiveness, and, also, software measurements which are based on these principles can be helpful in ensuring that the principles are followed. Our work was originally inspired by R. C. Martin’s seminal work [56] in the packaging of object-oriented software. However, Martin’s metrics and the newly developed metrics only measure direct coupling. The authors believe that this limitation decreases these metrics’ accuracy and can be misleading because it is a purely local measure.

In this study, we propose an approach that manages packaging in object-oriented software development by analyzing all dependencies of all packages. Our approach relies therefore on packages, classes, and direct as well as indirect dependencies. The main goal of this research is to provide software engineers with a means of discriminating between design alternatives at an early stage, as well as during the maintenance stage in order to continuously meet design goals of maintainability and testability. The newly developed metrics can help to
improve the design of the systems and help the designer to constantly increase the quality of the systems as they evolve.

1.2 Research Framework

The overall contribution of this study is that it documents the first research attempt to measure package dependency, responsibility, instability, abstractness, and the distance by considering indirect coupling through a scientific basis. More specifically, this research contributes in the following:

1. The set of metrics for dependency, responsibility, instability, abstractness, and the distance, defined in Chapter 5, based on global view of the system.

2. Proposing a useful application for the proposed dependency, responsibility, instability, abstractness, and the distance metrics.

3. The set of theoretical studies which establish the validity of the proposed metrics.

4. The set of empirical studies which enables sensible judgment as to how to check the current, or the future, design of a system and which also establish the validity of the model behind the relationship between these metrics and both maintainability and testability of a system.

Through the above contributions, my research is expected to help fill the current gaps of knowledge in software package-level metrics studies. First, this research helps in understanding the full nature of package dependency. Also, this research provides insight into the relationship between the package dependency and both maintainability and testability.

In the following, a brief about the publications of my research:

1. Part of chapter 5 of this dissertation has already been published:

2. Part of chapter 5, and chapters 6 and 7 have been included in a paper that has already been accepted by the International Conference on Trustworthy Systems and Their Applications (TSA).

Almugrin, S.; Melton, A., “Indirect Package Coupling Based on Responsibility in an Agile, Object-Oriented Environment,” the International Conference on Trustworthy Systems and Their Applications (TSA 2015), accepted.

3. A paper that covers part of chapters 8 and 9 has already been submitted to Computer Software and Applications Conference (COMPSAC 2015).

4. Another paper that covers part of chapters 8 and 9 that has already been accepted by the International Conference on Trustworthy Systems and Their Applications (TSA).


5. Chapters 8 and 9 have been included in a journal paper that has already been submitted to The Journal of Systems and Software.

6. Another paper that covers chapters 8 and 9 will be submitted in the near future.
1.3 **Organization**

The rest of this prospectus is organized as follows: Chapter 2 provides a background on coupling, object oriented metrics, and Martin’s metrics suite and its use in the literature. In addition, it gives some background on software evolution and software measurement. Graph theory and the concept of PageRank are also presented in Chapter 2. Chapter 3 presents the agile software development while Chapter 4 presents object oriented design principles. The proposed solution is introduced in Chapter 5. Software maintainability and testability are covered in chapter 6. Chapter 7 presents the theoretical validation of this study. The experimental design and the empirical analysis and results are presented in chapter 8 and 9, respectively. Chapter 10 discusses the threats to validity of the research. The conclusion and future work are presented in chapter 11.
2.1 The Concept of coupling

The notion of coupling in procedural systems was introduced in the early 1970’s. For example, Stevens et al. [68] defined the concept of coupling in structured software design as “the measure of the strength of association established by a connection from one module to another.” Since then, a lot of researchers proposed general guidelines to structure the procedural systems design in a modular way by maintaining low coupling between modules. Also, they indicated that high coupling decreases the maintainability, understandability, and reusability of the structured systems. The object-oriented paradigm has offered new concepts that influence coupling. For example, inheritance and polymorphic relationships are new kinds of relationships in an object-oriented paradigm. Also, the new concepts of classes and methods as compared to procedures (modules) in procedural languages are introduced by the object-oriented paradigm. Accordingly, the researchers have suggested extending the concept of coupling.

2.2 Object-Oriented Metrics

Since the beginning of the object oriented programming, there has been lots of research work towards offering software metrics for object oriented systems. The idea of object oriented programming that depends on object oriented metrics has been implemented in the design and implementation phases of a software system. Many of the object oriented metrics have been suggested in literature [16]. One of the most widespread and most discussed suites was proposed by Chidamber and Kemerer [19, 20]. Chidamber and Kemerer suggested a metrics suite for
object oriented design which is composed of six metrics that evaluate class cohesion, coupling, inheritance, and complexity. These metrics are Weighted Methods per Class (WMC), Response sets for Class (RFC), Lack of Cohesion in Methods (LCOM), Coupling Between Object Classes (CBO), Depth of Inheritance (DIT), and Number of Children of a Class (NOC). They enriched the definition of some of these metrics in [20]. Chidamber and Kemerer Metrics assess the design of an Object-Oriented System, which makes them more suitable for an object oriented paradigm. Several studies were conducted for validating Chidamber and Kemerer’s Metrics [4, 6, 11, 14, 15, 22, 52, 69, and 70]. Tang et al. [70] have studied the correlation between CK metrics and the likelihood of the existence of OO faults. Their results suggest that WMC can be a good indicator for faulty classes. Basili et al. [6] found that several of the Chidamber and Kemerer Metrics were linked with the fault proneness of classes. For example, the authors found that the larger the DIT value, the greater the probability of fault detection. Also, they observed that the larger the NOC, the lower the probability of fault detection. Li and Henry [52] investigated the link between some of Chidamber and Kemerer metrics and the extent of code change as a measure of maintenance effort. They revealed that CK metrics appeared to be sufficient in predicting the frequency of changes across classes during the maintenance phase. The authors suggested class level metrics that measures different internal attributes such as coupling, complexity, and size. These metrics are: The number of attributes in a class that have another class as their type (DAC), the number of different classes that are used as types of attributes in a class (DAC’), the number of local methods (NOM), and the total number of attributes and local methods (SIZE2). Briand et al. [13] indicated that DAC and DAC’ do not fulfill all the properties for coupling measures proposed by Briand et al. [10]. However, Li and
Henry [52] have conducted an empirical validation for their own metrics, and found that the maintenance effort, measured by the number of lines changed per class, could be predicted from the values of these metrics. Abreu proposed the MOOD metric set model [1] which is one of the basic structural methods of the object oriented paradigm. MOOD metrics are: Method Inheritance Factor (MIF), Attribute Inheritance Factor (AIF), Method Hiding Factor (MHF), Attribute Hiding Factor (AHF), Polymorphism Factor (POF), and Coupling Factor (COF). Recently the MOOD metric set model has been extended to MOOD2 metrics [2]. Lorenz and Kidd [47], W. Li et al. [53], Briand et al. [12, 13], Genero et al. [29], and J. Bansiya et al. [5] are some other examples for the software metrics that are presented in literature for object oriented paradigms at the class level.

### 2.2.1 Package level metrics

Using classes or objects only to model a system is insufficient to gain robust, maintainable, and reusable designs. A higher abstraction model such as on the package or subsystem level is desirable when the size of the system is rising. Accordingly, it has been acknowledged that a group of interrelated classes or objects is a better unit of organization than a single class or object. Different names have been given to these higher abstraction entities such as: class categories [9], subsystems [64], and set of classes [12]. In the Unified Modeling Language (UML), Java, and Smalltalk, these higher abstraction entities are called packages. The notion of a package is also supported in C++ in the form of namespace. Package level metrics do not uncover faulty or risky source code, because they only know the classes which belong to the package or interact with, and dependencies between these classes. The main purpose of package
level metrics is to identify unwell-designed packages by inspection of the distribution of classes in the packages and the interdependencies between these packages.

Unlike the class level metrics, only a few package level metrics were suggested in the literature. In the following are some examples of these works. Ducasse, Lanza and Ponisio [25] proposed Butterflies, tools for visualizing a package. They proposed a visualized chart which was built from simple package metrics based on a language-independent meta-model. The measures are simple size and coupling metrics. Ponisio in his dissertation [63] extended the work and proposed a new cohesion metric called Contextual Package Cohesion (CPC). Zhou, Xu, Shi, Zhou and Chen [76] presented another package-level cohesion metrics. Lakos [45] proposed a package dependency metric. Package’s cyclic dependencies have been studied by many researchers [31, 46, and 58].

Packaging of the system should be designed in a way that improves the communications within the design, raises the reusability of the system by detaching reusable elements from non-reusable packages, and reduces the maintenance costs by blocking the propagation of change. Nevertheless, highly interdependent designs have a tendency to be rigid, non-reusable, and hard to maintain. On the other hand, interdependence is sometimes required when the packages of the design are to cooperate with each other. This fact of package cooperation indicates that some forms of dependency are good and other forms are bad. Package level metrics is one of the design quality metrics that can be applied to the system design to check the conformance of that design to the desired pattern of dependencies and gives the designer a better understanding of their system and whether the system of such design has the required quality features such as maintainability and reusability. The details of this point will be presented in the next section.
2.2.2 Martin’s metrics

Robert C. Martin’s metric suite [56] is one of the most widely well-known software package measures. Martin’s metric suite consists of eight measures which calculate different features of a package. These measures are: efferent coupling (Ce), afferent coupling (Ca), Instability (I), Number of Abstract Classes (Na), Number of Classes (Nc), Abstractness (A), the distance from the main sequence (D), and the normalized distance from the main sequence (Dn).

Later, Martin has proposed a new cohesion metric (H) which does not belong to his original metric suite. Since the main focus is on the package dependency management, Martin’s cohesion metric (H) will not be discussed in this research. The dependency management metrics measure the packages dependency structure of a system. This metric suite includes: efferent coupling (Ce), afferent coupling (Ca), Instability (I), Abstractness (A), the distance from the main sequence (D), and the normalized distance from the main sequence (Dn). The main contribution of these metrics is in helping software designers to evaluate the maintainability and reusability of a package. The two coupling measures in Martin’s metric suite are efferent coupling (Ce) and afferent coupling (Ca). Efferent coupling (Ce) is defined as the number of classes inside a given package that depend upon classes outside this package. Efferent coupling (Ce) of a package symbolizes the reasons for a change and is an indicator of the independence of a package. In contrast, afferent coupling (Ca) can be defined as the number of classes outside a package which depend on the classes inside this package. Afferent coupling (Ca) of a package is an indicator of the package’s responsibility. Martin derives from these two coupling measures a package stability metric which is built based on his Stable Abstraction Principle (SAP) and Stable Dependencies Principle (SDP). More details about SAP and SDP and many other object oriented
design principles will be given in Chapter 4. The derived metric, the instability (I) metric, is defined as the ratio of efferent coupling (Ce) to total coupling (Ce / (Ca + Ce)), and has the range [0, 1]. I=1 indicates a maximally unstable package. I=0 indicates a maximally stable package. The stability of the packages, as stated by Martin, does not measure the frequency or likelihood of the change. However, the metric measures the package’s resilience to change. Martin describes the stability of a package as the “amount of work required to make a change” [56]. For example, if one package is depended upon by three other packages but depends on nothing. Then, this package has three reasons not to change, and has no reason to change. As a result, it is very difficult to change this package. Since this package does not have any dependencies, there are many fewer reasons for the package to be forced to change. Therefore, this package is claimed to be responsible and hence stable. On the other hand, if another package depends on three packages and no other packages are depended upon it, then this package has three reasons to change, and it has no responsibility. It has dependencies, which means that it is more likely to be forced to change as its depended upon packages do. It has no dependents, so there are very few barriers to changing it. Therefore, this package is claimed to be instable. A responsible package, which has lots of dependent packages, is hard to change because a modification of it may cause a refactoring of other packages. Nc and Na are the total number of classes in the package and the number of abstract classes or interfaces in the package, respectively. The sixth measure is the Abstractness (A), which is the ratio of the number of abstract classes (and interfaces) in a package to the total number of classes in that package, A= Na / Nc. Abstractness is a measure of the rigidity of a package, and has the range [0,1]. A = 0 indicates a completely concrete package and has no abstract package. A = 1 indicates a completely abstract package. To
define the relationship between stability (I) and abstractness (A), Martin created a graph with A on the vertical axis and I on the horizontal axis. The extremely stable and abstract packages are at the upper left corner (0, 1), and the extremely instable and concrete are at the lower right corner (1, 0). Martin in [56] claims that a stable package should be abstract. Due to the fact that stable packages are responsible packages, they are hard to change. In addition, they should also have a high abstractness so that they can be extended as per the Open Close Principle (OCP). On the contrary, unstable packages should be concrete so their code can be easily changed, since they have no responsibility. In real life, there are different degrees of abstraction and instability. Hence, not all packages will be located in these two points. Martin draws a line from (0, 1) to (1, 0) that he calls the main sequence. The diagram is illustrated below.
Figure 2.1: The Instability-Abstractness graph and the main sequence

The main sequence can be defined as the ideal line \((A + I = 1)\), in terms of its abstractness and instability. According to Martin, the package that sits on the main sequence is not too abstract, nor too instable; instead it has the right abstractness in proportion to its stability or inversely proportional to its instability \((I)\). Since the most desirable situation is when a package sits on the main sequence, Martin came up with the distance \((D)\) from the main sequence, \(D=|(A+I-1)/\sqrt{2}|\). His hypothesis is that the further away a package is from a point on
the main sequence, the worse it is from the dependency point of view. This metric represents how far a package is from ideal characteristics. The distance from the main sequence (D) has the range \([0, \sim0.707]\). The last related measure is the normalized distance from the main sequence (Dn), the distance of a package from the main sequence \(Dn \equiv |(A+I-1)|\). The normalized distance from the main sequence (Dn) indicates the package’s balance between abstractness and stability and its range is \([0, 1]\). As described before, ideal packages are either completely abstract \((A=1)\) and stable \((I=0)\), or completely concrete \((A=0)\) and instable \((I=1)\). Martin claims that the coordinates \((0, 0)\) and \((1, 1)\) in the previous figure represent bad design. In Figure 2.2, the area near the lower left corner at \((0, 0)\) holds packages that are very rigid. They are hard to change because of their responsibilities, and difficult to extend because they have very low abstractness. Martin calls this area “the Zone of Pain.” The area near the upper right corner at \((1, 1)\) holds packages that are abstract, but they do not have much responsibility. Undoubtedly, these kinds of packages are impractical. Martin calls this area “the zone of uselessness.”
2.2.3 Martin’s metrics in Literature

Many research studies in the literature have acknowledged and applied Martin’s metrics [18, 39, 40, 41, 59, 66, 77, 78, 79, and 80]. Madhwaraj and Babu [55] have studied the relationships between different package design metrics, such as Marin’s metrics, and their influence on software maintainability. Likewise, some works in the literature have accomplished the validation of Martin’s metrics and principles. One work has been conducted by John C.
Champaign [18]. He applied an empirical study of software packaging stability. Volatility describes a package that changes frequently over time and contextual stability describes the stability of a package over different releases. The study tried to validate what was claimed by the author as Martin’s assumption, a package changes slowly over releases if it has more dependents than those it depends on and vice versa. He studied three systems as case studies: systems, Eclipse, and Linux. The author found that Martin’s assumption was not valid for these systems. As a result, the study found no correlation between contextual stability and volatility for packages. However, Martin stated that the volatility of a package is a difficult thing to understand because it depends upon many kinds of factors. Martin also stated that stability is not a measure of the likelihood that a package will change; rather it is a measure of how hard it is to change a package. So, the result of this research needs to be checked and reevaluated to reflect the exact definition of Martin’s metrics and their implications.

Another work in the literature has been carried out by Hyrynsalmi and Leppänen to validate Martin’s metrics [32]. The authors stated that theoretical and empirical study is necessary to validate any software metric. Their study fails to validate Martin’s Instability and Abstractness metrics theoretically because of the lack of an applicable evaluation framework for such relative measures. After conducting experimental evaluation of five large open source systems, they were able to validate Martin’s metrics, since packages that have been evaluated meet his assumptions and reflect the well and poorly designed packages.

Zaretska and Besedina propose a new maintainability metric based on Martin’s metrics [75]. They claims that future system maintainability can be calculated based on the UML design. They introduce the enhancement metric (E) by the formula, E=nI/nC, which shows, as claimed
by the authors, the degree to which the main principles of object oriented design are satisfied. \( nC \) is the total number of connections between classes, while \( nI \) is the number of the “class - interface” or “interface - interface” connections. The closer \( E \) is to 1, the better; and the closer \( E \) is to 0, the worse.

2.3 **Software Evolution**

2.3.1 **Lehman’s Laws of Software Evolution**

The laws of software evolution are formulated by Lehman et al. [48, 49, 50, and 51]. The term software evolution is used in Lehman’s work to address the difference with the post-deployment activity of software maintenance. Software evolution is defined as the sequential series of changes that occur during a development lifecycle. He uses the term E-type system to mean the programs that must be evolved because they operate in or address a problem or activity of the real world. The laws of software evolution are summarized as follows:

1. Continuing change: E-type systems must be continually adapted to stay satisfactory.

2. Increasing complexity: Complexity increases as the system evolves, unless work is done to maintain or reduce it.


5. Conservation of familiarity: The average growth rate of E-type systems tends to remain constant or to decline.

6. Continuing growth: The Functional content of an E-type system increases to maintain user satisfaction over its lifetime.

7. Declining quality: The quality of E-type systems will appear to be declining, unless great steps are taken to adapt it to the changes.

8. Feedback system: Evolution processes are multilevel, multi-loop, multi-agent feedback systems.

Laws 1, 2, 6, and 7 are of particular interest for this research. Software changes over time. Many external attributes influence the software such as requirements changes, features addition, bug fixing, and environment changes. These forces cause incremental changes to the software structure over time. Changes imply increasing complexity, which makes the software more difficult for humans to understand and modify.

2.3.2 Object Oriented Software Evolution

Starting after Lehman’s work in the 1970s to the 1990s, large number of research papers studied E-type systems and analyzed the laws of software evolution created by Lehman. After the introduction of the object oriented paradigm, the advances in software metrics, and the changes in software distribution, research on software evolution has evolved to study these new challenges that must be considered by researchers. The popularity of object oriented languages gave researchers greater motivation to focus on changes to software structure in addition to size. Many metrics that measure structural characteristics specific to object oriented software have been proposed by researchers. These metrics give a general idea of changes in the system by
measuring classes/modules added, classes/modules deleted, classes/modules names changed, etc. For example Ali and Maqbool [3] examined the evolution of software by looking at the number of modules added, deleted, and changed over a series of releases. Higher abstractions of design have been used in object oriented languages such as design patterns and packages. Design patterns are defined by Gamma, et. al., [30] as “simple and elegant solutions to specific problems in object-oriented software design.” Gustafsson, et. al., gathered metrics specific to design patterns such as relative number of classes which calculates the number of classes that play a role in a design pattern to the total number of classes [30]. They use the proposed metrics as well as traditional metrics as quality indicators, and compare their results over a series of releases to analyze trends of design patterns over the evolution of software. Izurieta and Bieman [39, 40, and 41] analyzed how design patterns can evolve and decay in a way that affects the quality of the system. They studied the design pattern decay, a specialized type of software evolution, and stated that the most prominent form of decay is the modular grime. Modular grime builds up when design pattern classes develop new dependency relationships over time which are not specified in the original design of the design pattern. In the next chapters, we will present more details about different researchers who have studied software evolution at the package level, another higher abstraction level. The third contributor to the study of software evolution was the new software distribution. The open source software distribution has made it easier for the researchers to conduct their study. Hence, source code and revision histories are now available to the interested researchers. However, researchers should consider how to validate their results on commercial software.
2.4 Software Measurement

2.4.1 Introduction

Software measurement plays a crucial role in software engineering, and its significance has increased over the past decades. The main reason is that software measurement provides the tool for software engineers to be informed, track, and have the control over the development life cycle. Lord Kelvin mentioned the need for numerical data and stated: “In physical science a first essential step in the direction of learning any subject is to find principles of numerical reckoning and methods for practically measuring some quantity connected with it. I often say that when you can measure what you are speaking about, and express it in numbers, you know something about it, but when you cannot measure it, when you cannot express it in numbers, your knowledge is of a meager and unsatisfactory kind; it may be the beginning of knowledge, but you have scarcely, in your thoughts, advanced to the stage of science, whatever the matter may be.” [81]. He stated also in [42] that if you cannot measure it, then you cannot improve it. Similarly, DeMarco stated that you can't control what you can't measure [60]. Therefore, it is vital to make everything measurable. Although some of the software artifacts’ characteristics or properties are hard to measure, we should find a consistent way to make it measurable. As a result, there is an exigent need for well-established software measurements which quantify different attributes of software systems. This urgent need has led to an enormous search for measurements that provide insights on many internal and external attributes of a software system.

22
2.4.2 Terminology

Software Measurement

There are several definitions for software measurement. One of the most popular definitions is suggested by Fenton in [28] “mapping from the empirical world to the formal, relational world. Consequently, a measure is the number or symbol assigned to an entity by this mapping in order to characterize an attribute.” In this definition, an entity can be a software related object such as a person or a specification and design document. An entity can also be a software related event such as an organization or a program (source code). Every entity has one or more of attributes which are a feature or property of the entity. Examples of these attributes are the size of program or organization, or the number of lines (of a coded program). Another definition is “the act or process of assigning a number or category to an entity to describe an attribute of that entity.”[33]. Software measurement scales will be presented later in this section.

Measure

The term “measure” is incorrectly used interchangeably with the term “metric.” The definition of measure proposed in [33] stated that to measure is to apply a metric. According to [34], measures are “variables for which a value is assigned as the result of measurement.” Hence, measures are quantifiable variables which can be applied directly to the already mentioned entities to provide the needed assistance for the concern of people. Since measures are taken directly without calculations, they are called “absolute measures.”

Metric

A metric is defined as a combination of two or more measures or attributes [37]. Another definition stated that a metric is “A quantitative scale and method which can be used to
determine the value a sub-characteristic takes for a specific software product.” [34]. Metrics are used to evaluate entities and hence, they enable a quantititative comparison with other entities. As stated in [17] software metrics are useful to measure both the developing process and the product characteristics associated with software development. Selecting the most appropriate metrics is very critical and important for the organization to be successful. Examples of software metrics are: Source Line of Code (SLOC), McCabe Cyclomatic Complexity (CC), defect rates, error rates, Weighted Method per Class (WMC), Depth of Inheritance Tree (DIT), Number of Children (NOC), and Coupling Between Objects (CBO).

Indicator

An indicator is defined as a measure that provides an estimate or evaluation of specified attributes derived from a model with respect to defined information needs [38]. According to [35], an indicator is a measure that can be used to estimate or predict another measure. Indicators are sometimes called coefficients. The word “metric” and the word “indicator” are frequently confused by many software engineers.

Value

Value is defined as a number or a category assigned to an attribute of an entity by making a measurement [38]. Value is also defined as numerical or categorical result assigned to a base measure, derived measure, or indicator [36].

Goals of Metrics

Bundschuh et al., Goldensen et. al., from the Software Engineering Institute (SEI) at Carnegie Melon University states that, “the following goals can be achieved with a good metrics program:
1. To establish a common understanding throughout the organization.

2. To determine the information requirements of the organization and management processes.

3. To identify or develop a reasonable selection of measures according to the information requirements.

4. To identify and accomplish the activities for measurement.

5. To collect, store, analyze, and interpret the results of measurement.

6. To use measurement results for decision support as well as a basis for communication.

7. To evaluate and communicate the measurement process to the process owner.”

[17].

2.4.3 Software Measurement Scales

Five measurement scales are generally used in software measurement: Nominal, Ordinal, Interval, Ratio, and Absolute. These measurement scales are hierarchical, and each level scale has all the properties of the lower scales. Therefore, converting higher level scales into lower level scales is possible.

Nominal Scale

The nominal type, sometimes also called the qualitative type, is the most primitive form of measurement. A nominal scale divides the set of entities into mutually exclusive groups. These groups have no ranking among them. Examples of nominal scales are color, gender,
religion, nationality, race, and language. The mode is allowed as the measure of a central tendency for the nominal scale. On the other side, the median is not allowed.

**Ordinal Scale**

The ordinal scale divides the set of entities into exclusive and exhaustive equivalent classes that have some explicit relationship among them. Also, these classes have some sort of ordering among them but without quantitative comparison. Examples of ordinal scales are quality level (Very High, High, Medium, Low, Very Low), and opinion level (Strongly Agree, Agree, Neutral, Disagree, Strongly Disagree). The median and mode is allowed as the measure of central tendency for the ordinal scale. On the other side, the mean is not allowed.

**Interval Scale**

The interval scale captures the degree of difference in units of equal intervals. However, in this scale, ratios between them are not allowed. This scale captures information about the size of the intervals that separate classes. Examples of interval scale are temperature with the Celsius scale or Fahrenheit scale, and Georgian date. The mode, median, and arithmetic mean are allowed as the measure of central tendency for the interval scale.

**Ratio Scale**

The ratio scale preserves ordering, the size of intervals between entities, and ratios between entities. Unlike the interval scale, the ratio scale has an absolute zero point. Absolute zero means that the object measuring zero does not have the measured property. All statistical measures are allowed for the ratio scale. Examples of ratio scale are weight, length, duration, and income.
Absolute Scale

The absolute scale is the most informative scale. The measurement in this scale is done by a counting method. Examples of absolute scale are number of failures, and number of bugs.

2.4.4 Object-Oriented Design Metrics

Object-oriented Design (OOD) is a relatively new technology. Object oriented design is all about developing an object-oriented module of a software system to apply the identified requirements [65]. Objects are the basic units of object oriented design. Every object has an identity, states, and behaviors. A class is a static structure which represents a template for object(s) that share(s) common behaviors and definitions. Unlike the classical paradigm, an object oriented paradigm focuses more on the design phase of software development. Procedural and structural metrics, called traditional metrics, may not all be suitable to be applied directly on the object-oriented design. Software engineers need to have quantitative measurements for accessing the quality of OO designs. As a consequence, object oriented design metrics play an important role in maintaining high quality software systems. One of the first object-oriented designs metrics suite attempts was made by Chidamber and Kemerer (CK) [20]. A CK metrics suite is a set of metrics which captures different aspects of OO design. Currently, there are many object oriented design metrics that have been proposed in the literature. More information about CK metrics suites and other object oriented design metrics was given before in this chapter.
2.5 Graph Theory

2.5.1 Introduction

In mathematics, a graph \( G \) is a representation of a finite non-empty set of objects \( V \) and a set \( E \subseteq V \times V \) of edges consisting of unordered pairs of vertices. An edge \( e = (v_i, v_j) \) means that \( v_i \) and \( v_j \) are adjacent to one another, and that they are neighbors. The edges can be directed or undirected. In the following, a list of the related main terms definitions in the graph theory is presented. For more details on the graph theory, the reader is referred to the seminal work in [74].

2.5.2 Terminology

Directed Graph

A directed graph (digraph) is a graph that has an edge set \( E \) consisting of ordered pairs of vertices (directed edges).

Node Degree

The degree of a node \( v_i \in V \) is the number of edges incident with it, and is denoted as \( d(v_i) \). For directed graphs, the indegree of a node is the number of incoming edges at this node. The outdegree of a node is the number of outgoing edges at this node.

Walk

A walk in a graph \( G \) between nodes \( v_1 \) and \( v_n \) is an ordered sequence of vertices, starting at \( v_1 \) and ending at \( v_n \), such that between every pair of consecutive vertices there is an edge. i.e., \((v_{i-1}, v_i) \in E \) for all \( i = 1, 2, \cdots, n \). The length of the walk, \( n \), is measured in terms of number
of edges along the walk. A closed walk is a walk that is starting and ending at the same vertex. A walk with distinct edges is called a trail and a walk with distinct vertices is called a path.

Cycle

A cycle is a closed path with length \( n \geq 3 \). Therefore, a cycle has distinct vertices and begins and ends at the same vertex.

Shortest Path

A shortest path between node \( v_1 \) and \( v_n \) is a path that has the minimum length between nodes \( v_1 \) and \( v_n \).

Distance

The distance \( d(v_1, v_n) \) is the length of the shortest path between \( v_1 \) and \( v_n \). If no path exists between the two nodes, then \( d(x, y) = \infty \).

Adjacency Matrix

A directed graph \( G = (V, E) \), with \( |V| = n \) vertices, can be represented in the form of an \( (n \times n) \), binary adjacency matrix, \( A \), defined as:

\[
A(i, j) = \begin{cases} 
1, & \text{if } v_i \text{ is adjacent to } v_j \\
0, & \text{Otherwise} 
\end{cases}
\]

If the graph \( G \) is a weighted graph, then we have an \( (n \times n) \) weighted adjacency matrix, \( A \), defined as:

\[
A(i, j) = \begin{cases} 
wij, & \text{if } v_i \text{ is adjacent to } v_j \\
0, & \text{Otherwise} 
\end{cases}
\]

Where: \( wij \) is the weight on edge \( (vi, vj) \in E \).
Eccentricity

The eccentricity ($e$) of a node $v_i$ is the maximum distance from $v_i$ to any other connected node in the graph (finite distance), defined as:

$$e(v_i) = \max_j \{d(v_i, v_j)\}$$

Radius

The radius of a connected graph ($r(G)$) is the minimum eccentricity of any node in the graph. Thus, the radius can be defined as follows:

$$r(G) = \min_i \{e(v_i)\}$$

Diameter

The diameter ($d(G)$) is the maximum eccentricity of any node in the graph

$$d(G) = \max_i \{e(v_i)\}$$

Centrality

The centrality ($c$) is defined as a function that induces a total order on $V$, $c: V \rightarrow \mathbb{R}$. For example, $v_i$ is more central than $v_j$ if $c(v_i) > c(v_j)$.

2.5.3 Centrality Analysis

Centrality is a fundamental concept in network analysis. It is used to rank the nodes of a graph, based on how central or important they are. In social networks, analysts studied the centrality of individuals in their social networks. Bavelas studied the formal properties of centrality in 1950 [7]. Since his seminal work, a number of competing concepts and many different notions of centrality have been proposed. In the following, some of these notions of centrality will be given in short.
Degree Centrality

Degree Centrality is the simplest notion, and it is the degree \(d(v_i)\) of a node \(v_i\). So, the higher the degree, the more important or central the node is. For directed graphs, we have the indegree centrality and outdegree centrality of a node.

Eccentricity Centrality

Eccentricity centrality is another centrality notion, and can be defined as the less eccentric a node is the more central it is. Eccentricity centrality is thus defined as follows:

\[
c(v_i) = \frac{1}{e(v_i)}
\]

A node that has the least eccentricity is called a center node, whereas a node that has the highest eccentricity is called a periphery node. For example, if we want to choose a facility location such as hospital or firefighting center, we should pick up the center node because it minimizes the maximum distance to any node in the graph.

Closeness Centrality

Closeness centrality is a centrality notion which ranks a node based on the sum of all the distances to other nodes. Closeness centrality is thus defined as follows:

\[
c(v_i) = \frac{1}{\sum_j d(v_i, v_j)}
\]

A node that has the highest closeness centrality is called the median node. For example, if we want to choose a facility location such as coffee shop or a mall, we should pick up the node that minimizes the total distance over all the other nodes, a median node.
**Betweenness Centrality**

Betweenness Centrality is a centrality notion that measures for a given node \((v_i)\) how many shortest paths between all pairs of vertices include \(v_i\). This measure gives an indication for the central monitoring a node is performing for various pairs of nodes.

**Eigenvector Centrality (Prestige)**

As a centrality, the notion of prestige, or the eigenvector centrality, of a node \(v_i\) in a directed graph \(p(v_i)\) is supposed to be a rank of how important node \(v_i\) is. The basic idea is that the more nodes that point to a given node, the higher its prestige. However, the prestige of a node does not simply depend on its indegree, but it also depends on the prestige of the nodes that point to it in a recursive way.

Let \(G = (V, E)\) be a directed graph, with \(|V| = n\). The adjacency matrix of \(G\) is an \(n \times n\) asymmetric matrix \(A\) given as follow:

\[
A(i, j) = \begin{cases} 
1, & (i, j) \in E \\
0, & \text{Otherwise}
\end{cases}
\]

Thus, the prestige, or the eigenvector centrality, is defined as follows:

\[
P(v) = \sum_u A(u, v) P(u)
\]

\[= \sum_u A^T(v, u) P(u)
\]

Where \(p\) is an \(n\)-dimensional column vector corresponding to the prestige scores for each node, and \(A^T\) is the transpose of the adjacency matrix \(A\) of graph \(G\).
We can obtain an updated prestige vector in an iterative manner. Thus, if \((p_{k-1})\) is the prestige vector across all the nodes at iteration \((k-1)\), then the updated prestige vector at iteration \((k)\) can be given as follows:

\[
P_k = A^T P_k - 1
\]
\[
P_k = A^T (A^T P_k - 2)
\]
\[
P_k = (A^T)^2 P_k - 2
\]
\[
= \ldots
\]
\[
P_k = (A^T)^k P_0
\]

where \(P_0\) is the initial prestige vector.

**PageRank**

PageRank is a method for computing the prestige or centrality of nodes in the context of the web. The web can be presented as a graph that consists of web pages (nodes) connected by hyperlinks (edges). Like prestige, the PageRank of a node \(v\), recursively can be determined by the PageRank of other nodes that point to it, its dependents. Hence, a page will have high PageRank if the sum of the PageRank of its dependents is high, because either this page has many dependents or it has a few dependents with extremely high PageRank. PageRank is considered as a model of user behavior, where a user (the random surfer) clicks on hyperlinks randomly with no regard towards content. The random surfer may choose one of the hyperlinks from the current page which derives from the page's PageRank. The probability that the random surfer clicks on one hyperlink is given by the number of hyperlinks on the visited page. Hence, the probability of visiting a web page is the sum of probabilities for visiting this web page from
all the hyperlinks to this web page. The random surfer may get bored sometimes with some very small probability, dumping factor (d), and jumps to a non-linked web page. So the probability is reduced by the damping factor d.

Brin and Page in [83] defined PageRank as follows:

“We assume page A has pages T1, Tn which point to it (i.e., are citations). The parameter d is a damping factor which can be set between 0 and 1. We usually set d to 0.85. There are more details about d in the next section. Also C (A) is defined as the number of links going out of page A. The PageRank of a page A is given as follows:

\[ PR (A) = (1-d) + d \left( \frac{PR (T_1)}{C (T_1)} + \ldots + \frac{PR (T_n)}{C (T_n)} \right) \]

Note that the PageRanks form a probability distribution over web pages, so the sum of all web pages’ PageRanks will be one.”
CHAPTER 3

Agile Software Development

3.1 Introduction

As stated by Lehman [49], a software system should be maintained or it will lose its value as times goes. It is acknowledged that the large software development process should be evolving continuously [49]. Systems developed under traditional development often are rigid, and the cost of change is very high. This rigor has led to the uprising of agile methodologies which embrace change. Although the cost of change may not be flat, as claimed by Kent Beck [43], Agile may help in reducing the cost of change throughout the software life cycle. Therefore, agile methodologies are claimed to be a superior choice for developing software when the user requirements are indefinite or rapidly changing. Xtreme Programming (XP), Agile Modeling (AM), and Scrum are examples of the agile methodologies [72].

3.2 Agile Software Development Values

As stated in [82], agile software development values are:

1. Individuals and interactions over processes and tools.
2. Working software over comprehensive documentation.
3. Customer collaboration over contract negotiation.
4. Responding to change over following a plan.

3.3 Agile Software Development vs. Traditional Software Development

First of all, agile methodologies try to decrease the cost of change, and hence reduce the overall development costs. The cost of change for agile software development in comparison
with traditional software development according to the project progress is shown in Figure 3.1 [26].

![Cost of Change for Agile and Traditional development methods](image)

**Figure 3.1: Cost of change for Agile and Traditional development methods**

Second, the prime aim of Agile Software Development is to quickly develop software that meets customers’ needs. Agile software development places a higher importance on code, and a lesser importance on documentation. In Agile methodologies, working software is the primary measure of progress [82]. Therefore, agile software development encourages the team to start immediately, and make decisions when they have to. The reason for this is that they will have more domain knowledge, and therefore be in a better position to make the decision. This is

---

1 this figure is borrowed from reference [26]
clearly different than the traditional development where almost all the decisions are made in the initial stage of the development life cycle. Third, Agile Software Development has a different life cycle than traditional software development. The major element in agile software development is the refactoring. Kent Beck in his book [43] defines refactoring as the change of source-code and the system design without altering the functionality or external behavior of software. So the refactoring cycle has very important role in agile software development toward software quality assurance. Also, agile is an incremental iterative development where the traditional lifecycle is replaced by a sequence of short iterations, which typically last from one to four weeks, and every iteration adds some fully functional feature to the system.

3.4 **Xtreme Programming (XP)**

One of the main components of the XP life cycle is what so-called “Iteration.” The iteration is a recurring event in which the current software is adjusted to add additional functionality, enhancing or deleting existing functionality, or correcting errors or bugs. Before finishing any iteration, the updated version is checked for an acceptance test. Usually, each iteration takes from one week to four weeks. This period is very short as compared to the traditional software development life cycle. After each iteration, the refactoring cycle begins to apply the changes to the software. Figure 3.2 shows the general characteristics of the XP life.
Figure 3.2: XP Programming Life Cycle

\[2\text{ This figure is brought from reference [43]}\]
CHAPTER 4

Object Oriented Design Principles for Agile

4.1 **Introduction**

Most software engineers manage to maintain a high quality software design at the first release. As the software evolves, it becomes harder to maintain the quality of the design due to the changing requirements. Martin stated that without carefully implementing these changes, software starts to rot. In rotted systems, even simple changes to the software require great effort, since applying these changes means a redesign for the system. As a consequence, redesigns have a high chance to fail. Martin mentioned four symptoms for rotted designs: Rigidity, Fragility, Immobility, and Viscosity. Rigidity is the tendency for software to be difficult to implement even simple changes. This difficulty of changes occurs because of the cascading of subsequent changes in dependent classes/packages or the so-called ripple effect. As a result, the system has lost its ability to evolve, and it gets obsolete and dies. The second symptom is Fragility. Fragility happens when the software tends to break in different places every time there is a change. The breakage occurs in classes/packages that have no conceptual relationship with the changed classes/packages. Fragility makes it impossible to maintain the software. The third symptom is Immobility which means the inability to reuse part of the software in the same application or in different applications. So the developers have to develop the software from scratch instead of reusing the current software or part of it. The last symptom is Viscosity. There are two type of viscosity: design viscosity and environment viscosity. Design viscosity happens when the design preserving changes are harder to employ than the hacks. Therefore, the developers feel it is hard
to do the right thing. However, they also feel it is easy to do the wrong thing. Environment viscosity happens when the development environment is slow and inefficient.

As stated by Martin [56], requirements are the most volatile element in the software project. Since the fact that the requirements always change, it is our responsibility to deal with it and design a system that continuously evolves in response to the changing requirements. However, the question is how do we design a system that can survive the changing requirements? The answer is to let the agile developers be aware of the fundamental design principles and patterns in a consistent, disciplined way. In the following, we will briefly describe the Object Oriented Design Patterns, and discuss in details the Object oriented Design (OOD) principles.

4.2 Design patterns

Expert designers usually reuse good design decisions and solutions that have worked well for them in their previous experiences. In object oriented programming, lots of recurring patterns of classes and their relationships that provide practical solutions to similar problems have been implemented in well-designed object oriented systems. So design patterns are proven and certified design solutions are usually tested and reviewed to prove their correctness and efficiency. Design patterns are sets of principles that guide the designers in designing object oriented systems. Design patterns are defined by the so-called "Gang of Four" GoF (Gamma, Helm, Johnson, & Vlissides) in [71] as “simple and elegant solutions to specific problems in object-oriented software design.” Design patterns became popular after the GoF published the previously mentioned book [71] in 1994. GoF divided and classified the twenty three design patterns into three categories: creational, structural, and behavioral. Creational design patterns
help the designer with how to build problematic objects. Abstract factory, Builder, Factory method, Lazy initialization, Prototype, and Singleton are the most popular creational patterns. Structural design patterns help the designer in building flexible structures by controlling how classes and objects can be combined to form larger structures. Adapter, Bridge, Composite, Decorator, Facade, Flyweight, and Proxy are the most popular structural patterns. Behavioral design patterns help the designer to build powerful behaviors by specifying how classes and objects interact and distribute responsibilities. Command, Chain of responsibility, Interpreter, Iterator, Mediator, Observer, State, Strategy, Template method, and Visitor are examples of the structural patterns.

If we come to an agreement that OOD principles are good ideas, then design patterns can be seen as the tools that are used to implement these good ideas. Although most of design patterns follow one or more of design principles, these principles can also help the designers when there is no applicable design patterns.

4.3 **Object Oriented Design (OOD) principles Classification**

Object Oriented Design (OOD) principles govern the structure of object-oriented software systems. OOD principles are at the heart of object oriented design [44] and show the system designers what to use and what to avoid. Not only do the following Object Oriented Design (OOD) principles guide the design of a system, but they also should be followed during the evolution of the system. Most of the principles discussed here in this chapter were proposed by R. C. Martin in his book [56]. OOD principles can be classified into class-level principles and package-level principles. The class-level principles deal with the management of the
relationships between classes, while the package-level principles control the relationships between packages.

4.3.1 Class-level principles

In the following, we will introduce the basic five class-level principles of object oriented design (OOD). These principles intend to make object oriented software systems easy to maintain and reuse over time. Moreover, as stated in [73], Michael Feathers came up with the acronym of SOLID design principles (Single Responsibility Principle (SRP), Open/Closed Principle (OCP), Liskov Substitution Principle (LSP), Interface Segregation Principle (ISP), and Dependency Inversion Principle (DIP)). Next, SOLID design principles will be presented.

The Single-Responsibility Principle (SRP)

This principle was described first by DeMarco [23] and later also by Page-Jones [62]. In regard to this principle, they have different meaning for cohesion than Martin in [56]. Martin related the meaning of cohesion to the forces that make a class to change. The SRP states that there should be only one reason for a class to change. Martin defined responsibility to be a reason for change. As a result, if a class has more than one responsibility, then this class will have more than one reason to change. These responsibilities the class has will be coupled in a way that if one responsibility is changed, the others will be affected. As a result, the design will be fragile. However, if the two responsibilities will not be changed at different times, then separating them would add complexity to the design without any real gain. Moreover, decoupling the two responsibilities could be done by only separating their interfaces. Martin states that SRP is one of the simplest principles, but is one of the hardest principles to get right.
The Open Closed Principle (OCP)

Bertrand Meyer has originated the term Open/Closed Principle in his book, Object-Oriented Software Construction [8]. Then, R. C. Martin took this approach and proposed a very seminal work in [57]. As stated in [44], all of the class principles are derived from OCP. OCP states that classes should be closed for modification, but open for extension [56]. In other words, we should be able to change the behavior of the existing classes without modifying them. But how could we have classes that are extensible without modifying their existing code? The answer is Abstraction, which is the key to the OCP. In addition, inheritance and polymorphism are the main techniques to conform to the OCP, and they are based on abstraction. In object oriented programming, the software engineers can benefit from the OCP by the use of abstract classes (or interface classes) with pure virtual methods. The abstract class can be extended without being modified by deriving a concrete class that has the new required implementation details. Rigidity symptoms usually happen because of not applying OCP [56].

The Liskov Substitution Principle. (LSP)

This principle is named after Barbara Liskov because of her work in [54]. The LSP states that subclasses should be substitutable for their base classes [56]. To adhere to LSP, the preconditions and post-conditions for each of the methods of abstract classes must be defined first. Then, the developers of the derived subclasses must adhere to these preconditions and post-conditions. A method’s precondition is a set of condition(s) that must be met before one can invoke this method. While a method’s post-condition is a set of condition(s) that must be met in order for this method to return. LSP is known as “Design by Contract” [8]. Based on design by contract, one can say that the derived class is substitutable for its base class if its preconditions
are no stronger than its base class method and its post-conditions are no weaker than its base class method. As a result, LSP implies that member functions with references to base classes must not be confused when a derived class is substituted for the base class. LSP also implies that the base classes’ virtual member functions must be included in the derived classes. Accordingly, one can say that LSP is an extension to OCP, and hence violations of LSP also are violations of OCP.

Dependency Inversion Principle (DIP)

R. C. Martin proposed the following two design principles: the Dependency Inversion Principle (DIP) and the Interface Segregation Principle (ISP) [56]. DIP and ISP set some very useful rules of interface design. The DIP states that dependency should be upon abstractions and not upon concretions. In procedural design, the dependency structure is designed in a way that the high level modules depend upon lower level modules as shown in Figure 4.1 [56].

![Figure 4.1: Typical dependency diagram of a procedural Architecture](image-url)
Unlike procedural designs, it is completely not recommended for a high level module in object oriented designs to depend upon a low level module. Both should depend on abstract/interface classes. The main dependency structure for object oriented architecture is inverted as compared to the procedural architecture and shown in Figure 4.2 [56].

![Diagram of OO Architecture](image)

**Figure 4.2: Typical dependency diagram of an OO Architecture**

Usually, Abstract and interface classes implement the high level policy. Low level implementation is done by concrete classes. DIP assumes that abstract classes are non-volatile and concrete classes are volatile. Therefore, DIP recommends depending upon interface or abstract classes, but not upon concrete classes. Since the abstract classes provide the flexibility to
their dependents, classes that depend upon abstract classes can be easily extendable and modifiable. As a result, adhering to the DIP gives reusable frameworks.

*Interface Segregation Principle (ISP)*

As mentioned before, the ISP is proposed by R. C. Martin [56]. ISP states that many clients’ specific interfaces are better than a fat interface, one general purpose interface. So, according to the ISP, it is not recommended to have classes with non-cohesive interfaces, a fat interface. A fat interface implements core services as well as a set of services for the common needs of its client. Moreover, a fat interface can be split into groups of services where each group provides its services to a different set of clients. First, a fat interface’s clients should be categorized based on their type. Then, a fat interface can be refactored and broken down into several client-specific interfaces. As a result, interfaces must be designed to be used by its every client.

### 4.3.2 Package-level principles

As software applications increase in size and complexity, a higher level of abstraction is needed. Package organization is needed to help the designer to organize these large complex systems. Packages can be seen as containers for set of related classes. Package-level principles are the principles that help to design a robust package dependency structure. R. C. Martin proposes some useful design principles which guide the design of package relationships and classes’ allocation to packages. In the following, six principles of package design are presented. The first three principles are about package cohesion which helps in allocating classes to packages. The last three principles are about package coupling which guides the relationships between packages.
The Release Reuse Equivalency Principle (REP)

One of the most important goals of object-oriented design is to improve the reusability of software components. The REP states that “the granule of reuse is the granule of release” [56]. Since a package is usually the granule of reuse, it is also the granule of release. In other words, the granule of release should not be larger than a package, the granule of reuse. The merit behind this is that anything we plan to effectively reuse we must also release and track it through a tracking system so that its clients would have the willingness and the trust to reuse it. It is not recommended for when the client of the reusable package uses only some of its classes. The packaging should be designed to be client centric. As a result, the way we do the packaging has a tremendous impact on reuse.

The Common Reuse Principle (CRP)

The CRP states that “the classes in a package are reused together. If you reuse one of the classes in a package, you reuse them all.” [56]. The classes that are usually reused together should belong to the same package. So the classes that belong to the same package should have lot of dependencies between each other. The CRP provides the guidance not only to what classes to allocate to a package, but also to what classes not to allocate to a package. Actually, the CRP tells us more about the latter than the former. Since depending on one package means depending on every class in this package, one needs to make sure that it is never possible to depend on some of the classes and not the others.

The Common Closure Principle (CCP)

The CCP states that classes that change together, belong together. Classes belonging to a package should all be closed against the same set of changes. A package is called closed against
a set of changes if this package won’t be affected when these changes have been applied to the system. Similarly, a package is open against a set of changes if this package will be affected when these changes have been applied to the system. The CCP suggests allocating in one package the classes that are most likely to change for the same set of reasons. In other words, the CCP is trying to group classes that are open to a certain set of changes. So view the number of packages that will be affected when the requirement changes. As a result, the CCP helps maintain a very high level of maintainability, which is more important than reusability in most applications [56]. Although the CCP is easy to understand, it is difficult to apply due to the difficulty of predicting the set of changes that might occur and their effect.

The Acyclic Dependencies Principle (ADP)

The ADP states that one should “allow no cycles in the package dependency graph” [56]. Therefore, the package dependency graph must be a directed acyclic graph (DAG). The package dependency structure is a directed graph where the packages are the nodes and the dependency relationships are the edges. The main problem with the package dependency cycle is that it makes all packages in the cycle depend on each other. Therefore, all of these packages can be considered to be one large package, which means that changes made to one package of the cycle will have an immediate effect on all the packages. Unit testing and releasing any package in a cycle involves applying the same for all the packages included in the cycle. As a result, the cost of maintaining the software will be very high due to the rigidity of the system. In addition, the understandability and the reusability of the system or part of it will be very low. The easiest way to solve the cyclic package dependency is to factor out the classes that cause the cyclic dependency structure. First, a new package is created. Then, the classes that both of the two
packages in the cycle depend upon are moved into the new package. (See Figure 4.3) Another solution to break the cycle is to apply the Dependency Inversion Principle (DIP).

Figure 4.3: The first solution to break a dependency cycle

The SDP states that dependency decreases in the direction of stability [56]. Since every design needs to be flexible to some extent, some packages are expected to change and are sensitive to some specific changes as per the already mentioned Common Closure Principle (CCP). The packages that expected to be volatile should not be depended upon by any other non-volatile packages, because we will end up with a rigid design where all the packages are difficult to change. The SDP ensures that instable packages, which require less work to be changed, are
not depended upon by more stable packages, which require more work to be changed. As stated before, Martin describes the stability of a package as the “amount of work required to make a change” [56], and not the frequency or possibilities of change. As described before in Chapter 2, the SDP says that the Martin Instability metric (I) should decrease in the direction of dependency. In other words, the I metric of a depended upon package should be less than of the depending package. By conforming to the SDP, one can make sure that all the dependency relations flow in the direction of decreasing I.

The Stable Abstractions Principle (SAP)

The SAP principle states that a package should be as abstract as it is stable. This principle suggests the existence of a correlation between stability and abstractness. A stable package should be abstract so that it can be extended even though its stabile. Since there are no dependencies upon instable packages, there is no need to make the instable packages abstract. Instead, instable packages should be concrete. In other words, the SAP says that Martin’s Instability metric (I) should increase as Martin’s abstractness metric (A) decreases. Aiming to measure the SAP, Martin proposed a metric D that is built based on the instability metric (I) and the abstractness metric (A). The details of these metrics were presented in Chapter 2.
CHAPTER 5

The proposed Solution

5.1 Package Dependency Management

5.1.1 Package definition

In this research, the term “package” refers to a collection of classes which have a reason to locate in the same group. This collection of classes placed in a package is intended to carry out a common service(s). As stated by D’Souza in [21], that package is a named container for a basic development work unit that can be independently created, maintained, released, tested, and assigned to a team.

5.1.2 The Model

The model in this research is simple. It contains classes, packages, and dependencies. A package dependency relationship is based on the static dependency relationships between classes in this package and classes allocated in other packages in the system. The dependency between packages is directed. So the package dependency graph is a directed graph $G = (P, D)$, where:

- $P = \{1, \ldots, n\}$ is the set of nodes, representing packages.
- $D \subseteq P \times P$ is the set of edges, representing direct dependencies between packages.

At the package level in an object-oriented system and in any dependency relationship, one package provides the service (provider) and the other one asks for the service (client). Therefore, the dependency direction between packages defines client-provider relationships. The direct
dependency relationships between packages can be any one of the following types of dependencies:

1. Inheritance dependency: A subclass inherits behavior and state fields from its superclass and hence depends on it.

2. Reference and accesses dependency: There is an access dependency going from the client package to the provider package if one or more classes located in the provider package are explicitly used in any class located in the client package. An instance variable, class variable, and variable used as a parameter type or return type are examples of this type of dependency.

3. Method invocation. There is a method invocation that goes from the client package to the provider package if there is at least one class that resides in the client package invoking one method of any class located in the provider package.

The package dependency can be described as follows. A dependency relationship from the client package to the provider package is the union of all types of dependencies from classes in the client package to classes in the provider package. These dependencies can be automatically extracted from the source code in popular object-oriented programming languages such as Java and C++. In this study, we will focus only on the packages that have been produced by the application itself. Third-party packages are not included in the study because application developers have no control over these packages, as well as that these packages are expected to be tested and work perfectly.
5.1.3 Packaging

Packaging is a method of splitting a large software system into a number of collections of classes, packages, which makes this system and its components easier to understand, test, maintain, and reuse. Packaging is also set to manage the development and the distribution of the software systems. Moreover, releasing a software system in packages allows for smaller releases since the modified packages are the only packages that need to be tested and released. Since there are many distinct packages, the problem of how to package a system is arising. An appropriate choice for packaging of the software system ought to be set based on the package dependency and package stability. Martin states in [56] that a package is chosen such that each package is reusable and does not contain more than one reusable component as well as that the package dependency graph forms no cycle. Principles of package design were presented in Chapter 4.

However, as Martin stated, the package structure cannot be designed in a top-down approach [56]. In other words, package dependency diagrams shouldn’t be designed at the beginning of the development phases. Therefore, a package dependency structure is not proposed to represent the functions of the system. Actually, it is proposed to map the buildability of the application [56]. Designing the package dependency structure in a top-down approach will affect the maintainability and reusability in a negative way. As the number of classes grows in the early stages of implementation and design, package dependency management is needed. Actually, package dependency management solves “the morning-after syndrome,” and keeps the changes as contained as possible.
Martin stated that the morning-after syndrome happens in big development environments when the same source files are revised by many developers. What happens is that the developer worked all the day on his/her code, and at the end of the day he/she was able to make things done and went home. Then, the next morning the developer found that his/her code is not working because another developer made changes on the code he/she depended on. This circumstance in such an environment occurs almost every day. The result is that every developer fights to apply his/her changes and make them work with the last changes that other developers made. As a consequence, it is very hard to build a stable version of the project. To solve the morning-after syndrome, one needs to apply the Acyclic Dependency Principle (ADP). The ADP is discussed in detail in Chapter 4. Another resolution for morning-after syndrome is what so called, “the weekly build.” The weekly build technique allows the developers to work on their own code for the first four days of the week and ignore others’ code. They work in isolation without thinking of the work necessary for the integration. Then developers complete the integration and testing on the fifth day (Friday), and build the system. However, this method is not practical for large systems where the integration and test effort is very high and needs more than one day to make it done.

Keeping the changes as contained as possible is done by applying the single responsibility principle (SRP) and the common closure principle (CCP) to allocate the classes that are expected to change together in one package. As mentioned before, the CCP helps to maintain a very high level of maintainability, which is more important than reusability in most applications [56]. As the application grows, it reaches a point that the attention is moved to the reusability of the system components. Hence, the common reuse principle (CRP) starts to be the
main principle to apply for packaging the system. In this situation, the packages turn into units of work and, once tested, they are released for use by the other teams. Other teams can adopt the new release or stay working with the preceding release until they become prepared to adopt the new release. Managing the package’s dependency structure is the only approach to make this process works. Also, there must be no cycle in the package’s dependency structure. However, the Acyclic Dependency Principle (ADP) is applied anytime a package cycle emerges.

5.2 Instability and Abstractness

There has been much effort to discover the connection between different metrics and how they affect each other. One of the most promising studies has been proposed by Martin [56], which is the analysis of dependencies. As mentioned previously in Chapter 2, Martin in his seminal work suggests that to avoid the rigidity and fragility of OO design, dependencies must be inverted to achieve abstractness and stability of system design. He proposes a unique method for measuring abstractness and instability on the package level to deal with the increasing in size and complexity of the systems.

The stability metric proposed by Martin, known as the Instability metric (I), is defined as the ratio of efferent coupling (Ce) to total coupling (Ce / (Ca + Ce)) and has the range [0,1]. The closer to 1 the more the package is unstable and the closer to 0 the more the package is stable. The more stable a package is, the more reusable it is, and the more unstable a package is, the less reusable it is. The second metric is the abstractness metric (A), which is the ratio of the number of abstract classes (and interfaces) in a package to the total number of classes in that package, A = Na / Nc. The abstractness metric (A) is a measure of the rigidity of a package and ranges from [0, 1], where the closer to 1 the more the package is abstract and the closer to 0 the
more the category is concrete. Abstractness satisfies the already mentioned Open/Closed design principle. Martin created a main sequence graph, A on the vertical axis and I on the horizontal axis, which represents the relationship between these two metrics by the equation, A+I=1. The distance from the main sequence, \( D = \left| \frac{A+I-1}{\sqrt{2}} \right| \), represents how far a category is from ideal characteristics. D range is \([0, \sim 0.707]\) and it can be normalized to \( D_n = |A+I-1| \) to ranges from \([0, 1]\). These metrics help to measure the maintainability level of a package in a system [56].

5.3 The proposed method of calculating Instability: Dual Ranking

5.3.1 Introduction

As mentioned in the previous section, the afferent couplings (Ca), also called the import links, define the package’s responsibility, while the efferent couplings (Ce), also called export links, represents the dependency of the package. The more clients for a package, the more difficult the change is. A responsible package is difficult to change because a modification of it may lead to subsequent changes or even a refactoring. The responsibility of a package can be measured, based on how many services this package is providing for the other packages. On the other hand, any change applied to any provider package (server) may ripple up to its clients and cause them to change too. So, efferent coupling (Ce) of a package symbolizes the reasons for a change and is an indicator of the dependence of a package. Martin uses these two metrics to evaluate the instability of the package. As stated by Martin, the stability of the packages does not measure the frequency or possibilities of the change. However, Martin’s instability metric measures how difficult it is to change the package. We assume that the extra efforts that needed to be done when changing the package in order to maintain the integrity of the system and make
this change in consistent with other changes are included in the total efforts to change this package.

Martin’s coupling metrics, Ca and Ce, only measure the form of direct coupling that is, coupling between packages that have some direct dependency relationship. However, Ca and Ce don’t measure package indirect dependency, coupling between packages that have no direct dependency relationship. Package indirect coupling is a special form of coupling that establishes between two apparently unrelated packages through a buried dependency relationship. Without being able to measure the indirect dependency between packages, one won’t be able to predict the difficulty of a change. In large object-oriented software architectures, indirect dependency is very critical for maintaining high quality software systems. Ripple effect, a situation when one modification causes a cascading effect on the dependent entities (packages) along the dependency relationship path, can only be avoided if package indirect coupling is analyzed. As clearly known, the ripple effect is one of the most significant maintenance problems [84]. The larger the transitive dependencies would be, the more challenging to keep the changes in consistent with each other packages without mistakes. Another important aspect of the importance of the global dependency metric is that it is very likely that a local dependency metric such as Martin’s metric can yield a situation in which a more stable metric is depending on a less stable metric. Conforming to the package-level principles such as DAP and SAP requires the ability to have such a global metric that can measure the indirect dependency between packages.

Let’s take an example to illustrate the importance of considering indirect coupling. Item, Book, and BookType are classes declared in packages PackageA, PackageB, and PackageC,
respectively. The Java code for this example is shown in Figure 5.1. Figure 5.2 shows its corresponding package dependency diagram.

Figure 5.1: The Java code for the indirect dependency example
This example deals partially with issuing a book to a person. An Item corresponds to a single copy of a Book, which has an associated BookType: “adult” or “teen.” A borrower similarly has associated classes: adult, teen, or child. Method issue() in class Item issues an Item for a given book to a given borrower after checking whether the borrower is restricted from borrowing the item. For example, a child borrower is not allowed to borrow an adult or teen book. The borrowing policy is implemented by BookType. As shown in Figure 5.2, the PackageA package depends directly only on PackageB. PackageB also depends directly on PackageC. There is no direct dependency relationship between PackageA and PackageC. However, PackageA depends indirectly on PackageC. Any changes that will be applied to
PackageC, PackageA will be affected and needs to be changed, tested, and released too. For example, if the library management decides to change the borrowing policy. They will change the BookType class in PackageC. As a consequence, the PackageB (Book class) will be affected as well as the PackageA(Item class). This means that someone who wants to understand PackageA must also understand PackageB and PackageC. Which means there is coupling between PackageA and PackageC. The above problem can be generalized to any design scenario where there are indirect dependency relationships between packages. Therefore, local metrics, such as Martin’s instability, won’t be able to get the whole dependency relationships and hence the instability of a package won’t be as accurate as it should be.

There are many reasons why indirect, as well as direct, dependency makes changes more difficult to accomplish, and we will indicate the most important ones:

1. The more indirect dependency the package has, the more dependency overhead associated with a change is, which will add more overhead to the initial intended change.

2. The scope of the changes will be relatively larger because there are more direct and indirect dependent packages.

3. The dependency overhead implies more effort needed to apply the required changes to the indirect dependents packages to maintain the integrity of the system.

4. The span of the changes may affect different unrelated packages that are outside the local developers’ knowledge of the system, especially in large systems. Therefore, there is a need for adding more human resources and subject-matter
experts to apply this kind of changes associated with the indirect dependency overhead.

5. Since object oriented design principles encourage the encapsulation, polymorphism, and abstraction, there is more likelihood in indirect dependency relationships to modify the contract between the client and the provider packages by adjusting the interface or refactoring of these packages, which may affect some of their other clients and so on.

6. The more the dependency overhead the change has, the more hazardous it will be. Hence, there are more test efforts needed to be done in order to make sure that the changes are safe and won’t introduce errors or bugs to the indirect packages and ensure that these changes are compatible with each other.

One open question in Martin’s Instability metric is that depending on stable and responsible packages should be encouraged [32]. Martin counts every export link (Ce) in his instability metric equally, regardless how stable the dependent upon package is. As mentioned before, such interdependency is not always being bad, particularly if designed very cautiously. As undoubtedly known, dependency is not bad at all the time. As a matter of fact, Martin stated that some forms of dependency must be desirable and other forms must be undesirable. The author believes that the stability of the dependent package should be reduced when depending on an unstable package more than depending on a more stable package. By conforming to the Stable Dependencies Principle (SDP), we ensure that instable packages are not depended upon by more stable ones. Hence, we should let the instability metric distinguish between “good” and “bad” dependency and get the dependent package’s stability penalized when depending upon unstable
packages more than when depending upon stable packages. For example, the stability of the class or package won’t be reduced if it uses or inherits from a very stable entity such as Microsoft Foundation Class (MFC) Library classes, as in VC++, or java.lang classes, as in Java Development Environment.

In conclusion, the proposed metric will be shown to be a more accurate measure of instability than Martin’s instability metric, which simply counts the number of incoming and outgoing links, in five aspects:

1. It measures the direct dependency relationships as well as indirect dependency relationships which makes it a global measure that gives the whole picture of the interdependency between packages. It is very difficult to ensure conforming to many of the object oriented design package-level principles without being able to measure both kind of dependency: indirect and direct.

2. Being depended upon by a package that has few dependency relationships gives a larger contribution to its responsibility than being depended upon by a package with more dependency relationships.

3. Being depended upon by important packages gives more responsibility than being depended upon by less important packages.

4. Depending upon a responsible package contributes less to the instability than depending upon a package with lower responsibility.

5. Being depended upon through important dependency relationship gives more responsibility than being depended upon by the same package through a less important dependency relationship.
The last two points are satisfied when we apply the suggested customizations to the proposed metric, as will be described later in this chapter. The author believes that the already mentioned locality limitations decrease the accuracy of Martin’s metric suite and makes it misleading, because it is a purely a local measure and not a global measure. In the following, an approach that measures and analyzes all kind of dependency relations of all packages in an object-oriented software development is presented.

5.3.2 The proposed Dual Ranking Instability Metric Steps

The proposed dual ranking instability metric of the system’s packages is mainly composed into three steps. The main two steps are: The InRank and the OutRank rankings steps. In these two steps, the independency and responsibility of a package is measured by the help of the two new metrics, OutRank and InRank, respectively. After calculating these two rankings for all packages in the system, the instability of each package can be easily calculated in a straightforward way.

Step1: Responsibility Calculation (InRank)

The InRank of a package is a method for computing the package’s responsibility. Actually, InRank can also be seen as a ranking mechanism for the packages of a system based on the responsibility they have. The responsibility of a package reflects the prestige or importance of packages in the context of the whole system. InRank=1 indicates a maximally responsible package. InRank =0 indicates a maximally irresponsible package. A package will have a high InRank if the summation of the InRank of its dependents is high, either because of the many dependents packages this package has, or it has few number of dependent packages with
extremely high InRank. The InRank is very similar to the PageRank [83] which is described in Chapter 2. Grounded on the suggested model explained at the beginning of this chapter, we assume package (A) has (n) package(s) that depend upon it, specifically package (T1) to package (Tn). The parameter d is a damping factor (default value is 0.85) similar to the damping factor in PageRank. The InRank (A) is the InRank of package (A) and InRank (Ti) is the InRank of package (Ti) which depends on package (A). Also C (Ti) is the number of classes inside package Ti that depend upon classes inside package (A). D (Ti) is the total number of classes inside package Ti that depend upon classes outside package Ti. Then the InRank of a package (A) can be calculated in a recursive way as follows:

\[
\text{InRank} \ (A) = (1-d) + d \left( \text{InRank} \ (T1) \times \frac{C(T1)}{D(T1)} + \ldots + \text{InRank} \ (Tn) \times \frac{C(Tn)}{D(Tn)} \right)
\]

The InRank of a package (Ti) is always weighted with respect to the number of packages which package (Ti) depends upon. So, the more depending upon packages package (Ti) has, the less ranking value will package (A) gain from package (Ti). The weighted InRank of package (Ti) is then added up. The outcome of this is that an additional dependency relations to package (A) will always increase package (A)'s InRank. After all, the sum of the weighted InRank of all packages (T1: Tn) is multiplied with a damping factor (d). Hence, the extent that InRank of package (A) benefits from another depending upon packages is reduced.

Similar to PageRank, InRank can be considered as a model of user’s execution behavior, where a random user’s execution goes through dependency relationships at random with no regard towards content. The random user’s execution runs part of a package with a certain probability which derives from the package’s InRank. The probability that the random user’s execution goes through one dependency relationship is merely set by the number of dependency
relationships on that package. For that reason, a package's InRank is not completely passed on to a package it depends on, but is distributed by the number of dependency relationships on the package. So, the likelihood for random user’s execution reaching one package is the summation of likelihoods for the random user’s execution following dependency relations to this package. As mentioned before, this likelihood is reduced by the damping factor $d$.

Since the range of values of InRank varies widely and we need both metrics, InRank and OutRank, to contribute approximately proportionately to the final proposed instability metric, we need to normalize and adjust the InRank values for every package in the system that measured in this step to ranges between 0 and 1. This normalization is called Min-Max Normalization. In this kind of normalization, the minimum InRank value is set to 0, and the maximum InRank value is set to 1. The following formula is used for the normalization of InRank values:

$$\text{InRank}(A) = \frac{\text{InRank}_i(A) - \text{Min}}{\text{Max} - \text{Min}}.$$ 

Where $\text{InRank}(A)$ is the normalized value of the initial InRank value for package (A). Also, $\text{InRank}_i(A)$ is the non-normalized InRank value of package (A). Min and Max are the minimum and maximum non-normalized InRank values over the entire system, respectively.

**Step 2: Dependence Calculation (OutRank)**

The OutRank of a package is a method for computing the package’s dependence. Actually, OutRank can also be seen as a ranking mechanism for the packages of a system based on the dependency they have. As stated previously, change applied to any package may propagate to its dependents and cause them to change too. However, change is not the only thing that can ripple up through dependencies between packages. Failures can also propagate.
Therefore, the dependency of a package reflects the risk or vulnerability to the changes or incorrect execution of other packages in the context of the whole system. For example, a failure of one method causes failure of calling methods. This failure also causes the failure of the functions that call the calling functions, and so on. As a consequence, the more dependence a package has, the more likely it will fail. Briand et. al., stated that method invocation has shown to be a strong, stable indicator of fault proneness [14]. OutRank=1 indicates a maximally dependent package. InRank =0 indicates a maximally independence package. A package will have a high OutRank if the summation of the OutRank of its depending packages is high either because of the many depending packages this package has or that it has a few number of depending packages with an extremely high OutRank. Grounded on the suggested model explained at the beginning of this chapter, we assume that package (A) depends on (n) package(s), specifically package (T1) to package (Tn). The parameter d is a damping factor (default value is 0.85), similar to the damping factor in InRank, which means the probability of this package being vulnerable to changes that not because of its depended upon packages. The OutRank (A) is the OutRank of package (A), and OutRank (Ti) is the OutRank of package (Ti) which package (A) depends on. Then, the OutRank of a package (A) can be calculated in a recursive way as follows:

$$\text{OutRank} (A) = (1-d) + d (\text{OutRank} (P1) + ... + \text{OutRank} (Pn))$$

The OutRank of a package (A) is always computed by the dependency relationships (export links) on package (A), directly or indirectly. Thereby, the more dependency relationships a package (A) has, the more it will be dependent and thus vulnerable to change. The OutRank of
packages (T1: Tn) is then added up. Any extra dependency relationships (export links) for package (A) will always increase package (A)'s OutRank. After all, the sum of the InRank of all packages (T1: Tn) is multiplied by a damping factor (d). Therefore, the extent that OutRank of package (A) benefits from another depended upon packages is reduced.

OutRank can be considered as a model of change propagation, where a package’s vulnerability to changes is measured by the probability of either this change happened because of changed depended upon packages or because of this change is applied directly on this package. The probability of a package being vulnerable to change is merely given by the number of dependency links this package has. This is the reason that a depended upon package's OutRank is not divided by the number of packages depending on it (import links) as compared to the InRank method. Hence, the likelihood for the change reaching one package is the summation of likelihoods for the change reaching any of its depended upon packages. In this context, a change reaches a package means that this package needs to be changed or simply tested accordingly in order for this change to be verified and tested and for this package to be released. Similar to InRank, this probability is reduced by the damping factor d. The justification is that not all the changes on the depended upon packages will affect the correctness of their entire dependents package.

Since the range of values of OutRank varies widely and we need both metrics, InRank and OutRank, to contribute approximately proportionately to the final proposed instability metric, we need to normalize and adjust the OutRank values measured in this step to the range [0, 1] in such a way that the minimum OutRank value is set to 0 and the maximum OutRank value is set to 1. The following formula is used for the normalization of OutRank values:
\textbf{OutRank (A) = (OutRanki (A) - Min) / (Max - Min).}

Where: OutRank (A) is the normalized value of the initial OutRank value for package (A); and, OutRanki (A) is the non-normalized OutRank value of package (A); Also, Min and Max are the minimum and maximum non-normalized OutRank values over the entire system, respectively.

\textit{Step 3: Instability Calculation}

As Martin relates the stability of a package to not be easily moved [56], the stability is a measure of the difficulty in changing a package. As stated by Martin, stability can be achieved by the two characteristics: Independence and responsibility. Specifically, he stated that there are two forces that could make packages change. The first force comes from the dependency characteristic where the depending upon packages get changed and this change will ripple up to the depended upon packages and cause them to change. Hence, an independent package is not affected by this kind of force which increases its stability. The second force comes from the responsibility characteristic where there is a great deal of force from the dependent packages preventing the developer from changing their depended upon package and hence increasing its stability. Hence, a responsible package is affected by this kind of force which increases its stability. As a result, Independence and responsibility are the two main related factors that influence the stability of a package.

As stated before in Chapter 2, Martin has chosen to measure these previously mentioned two characteristics, i.e., independence and responsibility, by the two coupling metrics, Ca and Ce. The Ca and Ce metrics are calculated by counting the number of classes in packages that
have direct dependencies with the package in question. The efferent coupling (Ce) of a package is an indicator of the independence of a package. In contrast, the afferent coupling (Ca) of a package is an indicator of the package’s responsibility. Martin derives from these two coupling measures, Ca and Ce, a package stability metric which is built based on his Stable Abstraction Principle (SAP) and Stable Dependencies Principle (SDP). However, these two coupling metrics are a purely local measure and not a global measure. Conforming to SAP and SDP, one needs a global metric instead of a local metric. For that reason, the author suggests using the already introduced global coupling metrics, InRank and OutRank, that measure responsibility and dependence, respectively. Additionally, the author derives from these two metrics a new stability metric for the system’s package. The proposed metric is called a dual ranking instability metric. The equation for computing the new instability metric looks like:

\[ I(A) = \frac{\text{OutRank}(A)}{\text{OutRank}(A) + \text{InRank}(A)} \]

Where: \( I(A) \) is the instability metric of package (A). Also, OutRank (A) and InRank (A) are the normalized OutRank value and the normalized InRank value for package (A), respectively. This new metric has the range \([0, 1]\). \( I=1 \) indicates a maximally instable package. \( I=0 \) indicates a maximally stable package. Of course, the new stability metric of a package measures the package’s resilience to change, and not the likelihood of the change. Martin describes the stability of a package as the “amount of work required to make a change.”

Instability Calculation can also be done by another method that also relates between Instability to the new two coupling metrics, InRank and OutRank. We can create a dependency-responsibility graph with OutRank on the horizontal axis and InRank on the vertical axis. Every
package can be represented as a point on this graph such as (OutRank, InRank). The maximal stable package can be found at the upper left at (0, 1). This package is the most independent package on the system, OutRank = 0. Also, this package is the most responsible package on the system, InRank =1. In contrast, the maximal instable package can be found at the lower right at (1, 0). This package is the most dependent package on the system, OutRank = 1. Also, this package is the most irresponsible package on the system, InRank =0. Figure 5.3 shows the dependency-responsibility graph.

![Dependency-Responsibility Graph](image)

**Figure 5.3: The Dependency-Responsibility diagram**

Of course, not all packages can sit at one of these two spots, because packages have degrees of dependence and responsibility. Consider a package at the point (0, 0) as shown in Figure 5.4. This package is an independent and irresponsible package. Since its OutRank =0,
such a package is very independent, and hence its stability will be increased. Also, this package is very irresponsible, InRank = 0, and hence its stability will be decreased. This package is as far from the stability position as it is from the instability position. So, its instability should be 0.5.

Figure 5.4: Package sit on (0, 0) and its distance from stability and Instability points

Consider again a package at the point (0.5, 0.5) as shown in Figure 5.5. This package is a partially independent and partially responsible package. Since its OutRank = 0.5, such a package is half independent and hence is partially instable, or partially stable. Moreover, this package is halved responsible, InRank = 0.5, and hence is partially instable, or partially stable. Also, this
package is as far from the stability position as it is from the instability position, i.e. 0.7071 from point (0, 1) and (1, 0). So, its instability should be 0.5.

Figure 5.5: Package sit on (0.5, 0.5) and its distance from stability and Instability points

Thus, we can create an instability metric which measures how far away a package is from these two points, the stability point and the instability point, i.e. (0,1) and (1,0). Our hypothesis is that the further away a package is from the most stable point as compared to the sum of the distance from this package to the most stable point and the distance from this package to the unstable point, the worse it is from the stability point of view. In other words, as close as a package is to the stability point as compared to the sum of its distance to stability point and its
distance to the instability point the more stable it will be. Therefore, we want to measure the
distance from the package to the stability and instability points. Its stability characteristic can be
known by measuring how far it is from the stability point as compared to the sum of the distance
from this package to the most stable point and the distance from this package to the instable
point. So the basic formula for instability ($I'$) will be as follows:

$$I'(A) = \frac{\text{Distance(Package (A), Stability Point )}}{\text{Distance(Package (A), Stability Point )} + \text{Distance(Package (A), Instability Point (1,0))}}$$

$$I'(A) = \frac{\text{Distance(Package (A), point (0,1) )}}{\text{Distance(Package (A), point (0,1) )} + \text{Distance(Package (A), point (1,0))}}$$

Based on the Pythagoras' Theorem, the basic formula for the distance between two points
given their coordinates is as follows:

$$\text{Distance( (x1,y1), (x2,y2) )} = \sqrt{dx^2 + dy^2}$$

$$\text{Distance( (x1,y1), (x2,y2) )} = \sqrt{(x2 - x1)^2 + (y2 - y1)^2}$$

Then, the distance between the package and the stability point can be computed as
follows:

$$\text{Distance(Package (A), point (0,1)} = \sqrt{(\text{OutRank(A)} - 0)^2 + (\text{InRank(A)} - 1)^2}$$

$$\text{Distance(Package (A),point (0,1)} = \sqrt{\text{OutRank(A)}^2 + (1 - \text{InRank(A)})^2}$$

Also, the distance between the package and the instability point can be computed as
follows:
Then, the equation for computing the new instability metric looks like:

\[
\text{Distance}(\text{Package}(A), \text{point}(1,0)) = \sqrt{(1 - \text{OutRank}(A)^2) + (\text{InRank}(A) - 0)^2}
\]

\[
\text{Distance}(\text{Package}(A), \text{point}(1,0)) = \sqrt{(1 - \text{OutRank}(A)^2) + (\text{InRank}(A))^2}
\]

Then, the equation for computing the new instability metric looks like:

\[
I'(A) = \frac{\sqrt{\text{OutRank}(A)^2 + (1 - \text{InRank}(A))^2}}{\sqrt{\text{OutRank}(A)^2 + (1 - \text{InRank}(A))^2 + \sqrt{(1 - \text{OutRank}(A)^2) + (\text{InRank}(A))^2}}}
\]

Where: \(I'(A)\) is another instability metric of package \(A\); Also, OutRank \(A\) and InRank \(A\) are the normalized OutRank values and the normalized InRank values for package \(A\), respectively. As clearly shown, the derived metric, the new instability \(I\) metric, is defined as the ratio of the distance from the package to the most stable point to the total distance from this package to both the most stable point and most instable point. This new metric has the range \([0, 1]\). \(I=1\) indicates a maximally instable package. \(I=0\) indicates a maximally stable package. In this study, we will use the first method for calculation of Instability \(I\).

5.3.3 The proposed method of calculating Package’s Abstraction

The Stable Abstraction Principle (SAP) implies that stable packages should be highly abstract. An abstract class is a superclass that is used to state or define general characteristics. Also, an abstract class cannot be instantiated. The abstract classes provide the flexibility to the design; therefore, the design should be very stable. To provide flexibility, abstract classes typically act as an interface rather than provide the implementation details. On the other hand, concrete classes usually provide the implementation details, which make them unstable due to its rigidity. Martin defines the Abstractness \(A\) as the ratio of the number of abstract classes (and
interfaces) in a package to the total number of classes in that package, $A = \frac{N_a}{N_c}$. As a consequence, abstractness is a measure of the rigidity of a package and has the range $[0,1]$. However, there will be a design problem if there is little use of the abstract classes in a package as compared to the use of the concrete classes which means few number of dependents classes for the abstract classes as compared to of the concrete classes. Hence, the responsibility of the abstract classes in a package is an important factor that defines the package’s abstractness.

Martin’s Abstractness metric counts the number of abstract classes a package has in order to define the abstractness of this package. As stated before, not all the abstract classes have the same level of responsibility. If the design of the system is not very carefully managed, then we may end up having some abstract classes that have limited responsibility and less reachability as compared to the responsibility level of a typical abstract class, or even worse, as compared to the responsibility level of a typical concrete class. To calculate the new abstractness of package, we need first to calculate the InRank at class level. The InRank of a class is calculated by the same way the InRank of a package is calculated in the previous section but in a lower model abstraction, class dependency diagram. Grounded on the class dependency diagram, which is similar to the model explained at the beginning of this chapter but in class level, we assume class $(A)$ has $(n)$ class(es) that depend upon it, specifically class $(T_1)$ to class $(T_n)$. The parameter $d$ is a damping factor (default value is 0.85) similar to the damping factor in InRank of a package. The InRank $(A)$ is the InRank of class $(A)$ and InRank $(T_i)$ is the InRank of class $(T_i)$ which depends on class $(A)$. Also $C(T_i)$ is the number of class which class $(T_i)$ depends upon. Then, the InRank of a class $(A)$ can be calculated in a recursive way as follows:

$$\text{InRank} (A) = (1-d) + d \left( \frac{\text{InRank} (T_1)}{C(T_1)} + \ldots + \frac{\text{InRank} (T_n)}{C(T_n)} \right)$$
We assume package (Pi) contain set (A) of abstract classes and set (C) of all classes, abstract and concrete classes. The InRank (Aj) is the InRank of the abstract class (Aj) from the set of abstract classes (A) in Package (Pi). Similarly, the InRank (Cj) is the InRank of class (Cj) from the set of all classes (C) in Package (Pi). Then, the proposed method for calculating the new abstractness of package (Pi) can be calculated as follows:

\[
Abs(P_i) = \frac{\sum_{A_j \in A} \left( \text{InRank}(A_j) \right)}{\sum_{C_j \in C} \left( \text{InRank}(C_j) \right)}
\]

The proposed abstractness metric measures the rigidity of a package and has the range [0, 1]. A = 0 indicates a completely concrete package and has no abstract package. A = 1 indicates a completely abstract package.

5.3.4 The proposed method of calculating a Package’s Normalized Distance

To define the link between instability (I) and abstractness (A), Martin introduced a graph with A and I on the vertical and the horizontal axis, respectively. The package can be represented as point of this format: (Instability, Abstractness). Packages at the location (0, 1) are the most stable and abstract packages. Similarly, Packages at the location (1, 0) are the most instable and concrete ones. Figure 5.6 shows the instability-abstractness diagram.
As mentioned before in Chapter 2, a stable package should be abstract because they are responsible packages and thus, they are hard to change. As stated before, we assume that the extra efforts that needed to be done when changing the package in order to maintain the integrity of the system and make this change in consistent with other changes are included in the total efforts to change this package. Abstractness gives a package the ability to be extended as per the Open Close Principle (OCP). On the contrary, unstable packages should be concrete because they are irresponsible packages and thus, they are easy to change. Such packages don’t need to

**Figure 5.6: the Instability-Abstractness diagram and the zones of Pain and Uselessness**

As mentioned before in Chapter 2, a stable package should be abstract because they are responsible packages and thus, they are hard to change. As stated before, we assume that the extra efforts that needed to be done when changing the package in order to maintain the integrity of the system and make this change in consistent with other changes are included in the total efforts to change this package. Abstractness gives a package the ability to be extended as per the Open Close Principle (OCP). On the contrary, unstable packages should be concrete because they are irresponsible packages and thus, they are easy to change. Such packages don’t need to
be abstract. In real systems, packages have different degrees of abstraction and instability; therefore, not all packages are located in the two extreme points, i.e., (0, 1) and (1, 0). As mentioned before, the line connecting these two points is called the main sequence. According to Martin, the package that sits on the main sequence has the right abstractness in inverse proportionality to its instability (I). Since the most desired situation for a package is to sit on the main sequence, Martin came up with a new metric, D, that measure the distance from this package to the main sequence, \( D = \frac{|A + I - 1|}{\sqrt{2}} \). His hypothesis is that the further away a package is from a point on the main sequence, the worse it is from the dependency point of view. This metric shows how far a package is from the ideal characteristic. The distance from the main sequence (D) has the range \([0, \sim 0.707]\). Martin suggests a new metric measure the normalized distance from the main sequence, \( D_n = |A + I - 1| \). \( D_n \) indicates the package’s balance between abstractness and stability and its range is \([0, 1]\).

Additionally, the coordinates (0, 0) and (1, 1) in the previous instability-abstractness figure represent bad design. The area near the lower left corner at (0, 0) is called “the Zone of Pain” because it contains packages that are very rigid. They are hard to change because of their responsibilities and difficult to extend because they have very low abstractness. Also, the area near the upper right corner at (1, 1) is called “the zone of uselessness” because it contains packages that are impractical. They are abstract but they do not have much responsibility.

### 5.4 Motivational Example

In the following, we will show some motivational examples to illustrate the proposed dual ranking instability metric and compare it with Martin’s instability metric. For simplicity, we assume that the number of classes inside any package that depend upon classes inside other
package is one, \( C(T_i) = 1 \). Figure 5.7 represents the simplified package dependency graph of the first motivational example.

![Package Dependency Graph](image)

**Figure 5.7: The package dependency graph for the first motivational example**

As mentioned before, each node in the system model represents a package of that system. It is clear from this figure that the levels of dependency and responsibility of these packages are not the same. For instance, the most responsible package as per Martin’s \( C_a \) metric is package (A) because it has more dependent packages than any other packages in the system. However, in reality Package (H) is more responsible than Package (A), even though Package (H) has fewer direct dependents. Actually, Package (A) depends upon Package (H). As a consequence, it is very difficult and needs lots of work to make any change to package (H) due to the added effort necessary to apply the subsequent changes on all of the dependent packages of package (H), namely Packages (A, B, C, D, E, and F), to ensure that they work in the expected manner after that change. The Package (A) has less dependent packages, namely Packages (B, C, D, E, and F), which make it relatively less hard to change than Package (H). This figure clearly shows that the
cost of changing a responsible package, such as package (H) and (A), is higher than the cost of changing a less responsible package or an irresponsible package such as package (C) and package (D) in this example. Hence, the responsible packages are necessary to be stable. On the other hand, packages (C) and package (D) are irresponsible because there is no package that depends upon them. Therefore, it is easier to make changes to such irresponsible packages because any change to them doesn’t necessarily affect and propagate to other packages. Independent packages are the packages that don’t depend on other packages. These packages have less reason to change because they don’t have depended upon packages. Two of the nine packages in this example are independent, namely, package (H) and package (I). All the other seven packages are dependent with different levels of dependency. Martin’s instability metric assigned the maximal instable value to any irresponsible packages, e.g., Package (C) and Package (D), regardless the level of their dependencies. However, the proposed instability metric takes this in consideration and assigns different instability values to such irresponsible packages. For example, the instability value of Package (C) is 1 while the instability value of Package (D) is 0.56 due to its dependency, i.e. its OutRank = 0.2. On the other hand, Martin’s instability metric assigned the minimal instable value to any independent packages, e.g., Package (H) and Package (I), regardless of the level of their responsibilities. However, the proposed instability metric takes this into consideration and assigns different instability values to them. For example, the instability value of Package (H) is 0, while the instability value of Package (I) is 0.38 due to its responsibility, i.e. its InRank = 0.34. Table 5.1 shows Martin’s instability metric and the proposed instability metric based on the dual ranking method for all the packages in the first example.
Table 5.1. Comparison between Martin’s instability and the proposed instability metric for the packages in the first motivational example

<table>
<thead>
<tr>
<th>Package</th>
<th>Martin’s Instability</th>
<th>The new Instability</th>
<th>Difference</th>
<th>Difference %</th>
<th>InRank</th>
<th>OutRank</th>
<th>Martin’s Instability Ranking</th>
<th>The new Instability Ranking</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0.2</td>
<td>0.1</td>
<td>0.1</td>
<td>50%</td>
<td>0.92</td>
<td>0.11</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>B</td>
<td>0.5</td>
<td>0.54</td>
<td>-0.04</td>
<td>-8%</td>
<td>0.07</td>
<td>0.20</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0%</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>1</td>
<td>0.56</td>
<td>0.44</td>
<td>44%</td>
<td>0</td>
<td>0.20</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>E</td>
<td>0.67</td>
<td>0.67</td>
<td>0</td>
<td>0%</td>
<td>0.07</td>
<td>0.48</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>F</td>
<td>0.5</td>
<td>0.52</td>
<td>-0.02</td>
<td>-4%</td>
<td>0.14</td>
<td>0.20</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>G</td>
<td>0.5</td>
<td>0.49</td>
<td>0.01</td>
<td>2%</td>
<td>0.14</td>
<td>0.11</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>H</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0%</td>
<td>1</td>
<td>0</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>I</td>
<td>0</td>
<td>0.38</td>
<td>NA</td>
<td>NA</td>
<td>0.34</td>
<td>0</td>
<td>8</td>
<td>7</td>
</tr>
</tbody>
</table>
Figure 5.8: The package dependency graph for the second motivational example
Figure 5.8 represents the package dependency diagram of the second motivational example. Based on Martin’s metrics, the most responsible package is package (E) because it has more direct dependent packages than any other packages in the system. Also, package (E) depends only on one package. As a consequence, it is very stable and its I value is 0.17. However, there are four packages that have more responsibility than Package (E) if we consider the indirect dependency as in the proposed responsibility metric, i.e., InRank. Package (E) has InRank value of 0.52. Packages J, K, N, and O have an InRank value of 0.56, 0.77, 0.97 and 0.94, respectively. Also, as per Martin’s metrics, package (N) and package (O) have very low responsibility, i.e., Ca =1. However, they are the second and the third responsible packages in this system, respectively. Another example that shows the lack of only direct dependency measuring as of Martin’s metrics is that Package (G) and package (N) have the same direct dependency value, i.e., Ce =1. However, when we also consider the indirect dependency as well as direct dependency, then their dependency values as per the proposed dependency metric, OutRank, are different. The OutRank of Package (G) is 0.97 while the OutRank of Package (N) is 0.12. In addition, the instability of package (E) based on the proposed I metric is 0.59, which is relatively 247% more instable than of Martin’s instability metric, i.e., 0.17. Likewise, the instability of package (G) based on the proposed I metric is 0.88 which is relatively 166% more instable than of Martin’s instability metric, i.e., 0.33. Similarly, the instability of package (N) based on the proposed I metric is 0.09, which is relatively 82% less instable than of Martin’s instability metric, i.e., 0.5. As mentioned in the first example, Martin’s instability metric assigned the maximal instable value to any irresponsible packages, e.g., Packages (A), (B), (C), (D), (H), and (I), regardless of the level of their dependencies. However, the proposed instability
metric takes this into consideration and assigns different instability values to such irresponsible packages. For example, the instability value of Packages (H) and package (I) is 0.96, due to their dependency, i.e. their OutRank = 0.94, while the instability value of Packages (A), (B), (C), and (D) is 0.82, due to their dependency, i.e. their OutRank = 0.72. Similarly to what mentioned in the previous example, Martin’s instability metric assigned the minimal instable value to any independent packages, e.g., Package (O), regardless the level of their responsibilities. However, the proposed instability metric takes this into consideration and assigns different instability values to them. For example, the instability value of Package (O) is 0.04 due to its responsibility, i.e. its InRank = 0.94. Table 5.2 shows the Martin’s instability metric and the proposed instability metric based on the dual ranking method for all of the packages in the second example.
Table 5.2. Comparison between Martin’s instability and the proposed instability metric for the packages in the second motivational example

<table>
<thead>
<tr>
<th>Package</th>
<th>Martin’s Instability</th>
<th>The new Instability</th>
<th>Difference</th>
<th>Difference %</th>
<th>InRank</th>
<th>OutRank</th>
<th>Martin’s Instability Ranking</th>
<th>The new Instability Ranking</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>0.82</td>
<td>0.18</td>
<td>18%</td>
<td>0</td>
<td>0.72</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td>0.82</td>
<td>0.18</td>
<td>18%</td>
<td>0</td>
<td>0.72</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
<td>0.82</td>
<td>0.18</td>
<td>18%</td>
<td>0</td>
<td>0.72</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>D</td>
<td>1</td>
<td>0.82</td>
<td>0.18</td>
<td>18%</td>
<td>0</td>
<td>0.72</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>E</td>
<td>0.17</td>
<td>0.59</td>
<td>-0.42</td>
<td>-247%</td>
<td>0.52</td>
<td>0.71</td>
<td>14</td>
<td>9</td>
</tr>
<tr>
<td>F</td>
<td>0.67</td>
<td>0.82</td>
<td>-0.15</td>
<td>-22%</td>
<td>0.27</td>
<td>1</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>G</td>
<td>0.33</td>
<td>0.88</td>
<td>-0.55</td>
<td>-166%</td>
<td>0.18</td>
<td>0.97</td>
<td>12</td>
<td>3</td>
</tr>
<tr>
<td>H</td>
<td>1</td>
<td>0.96</td>
<td>0.04</td>
<td>4%</td>
<td>0</td>
<td>0.94</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>I</td>
<td>1</td>
<td>0.96</td>
<td>0.04</td>
<td>4%</td>
<td>0</td>
<td>0.94</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>J</td>
<td>0.5</td>
<td>0.56</td>
<td>-0.06</td>
<td>-12%</td>
<td>0.56</td>
<td>0.70</td>
<td>8</td>
<td>10</td>
</tr>
<tr>
<td>K</td>
<td>0.5</td>
<td>0.46</td>
<td>0.04</td>
<td>8%</td>
<td>0.77</td>
<td>0.68</td>
<td>8</td>
<td>11</td>
</tr>
<tr>
<td>L</td>
<td>0.5</td>
<td>0.46</td>
<td>0.04</td>
<td>8%</td>
<td>0.38</td>
<td>0.30</td>
<td>8</td>
<td>11</td>
</tr>
<tr>
<td>M</td>
<td>0.25</td>
<td>0.15</td>
<td>0.1</td>
<td>40%</td>
<td>1</td>
<td>0.22</td>
<td>13</td>
<td>13</td>
</tr>
<tr>
<td>N</td>
<td>0.5</td>
<td>0.09</td>
<td>0.41</td>
<td>82%</td>
<td>0.97</td>
<td>0.12</td>
<td>8</td>
<td>14</td>
</tr>
<tr>
<td>O</td>
<td>0</td>
<td>0.04</td>
<td>-0.04</td>
<td>NA</td>
<td>0.94</td>
<td>0</td>
<td>15</td>
<td>15</td>
</tr>
</tbody>
</table>
Figure 5.9: The package dependency graph for the third motivational example\(^3\)

Figure 5.9 shows the third motivational example, and represents a solution architecture diagram of Microsoft Dynamics CRM 4.0, depicting all of its basic components and their interdependencies. Based on Martin’s metrics, the most responsible package is package (G) because it has more direct dependent packages than any other packages in the system. Also, package (G) depends on only two packages. As a consequence, it is relatively stable, and its I value is 0.33. However, if we consider the indirect dependency as in the proposed responsibility metric, i.e., InRank, there are three packages having more responsibility than Package (G). Package (G) has an InRank value of 0.39. Packages A, E, and I have an InRank value of 1, 0.42, and 0.77, respectively. Also, as per Martin’s metrics, package (C) and package (D) have low dependency, i.e., Ce =2. However, they are the second dependent packages in this system, respectively. Another example that shows the lack of only direct dependency measuring as of Martin’s metrics is that Package (C) has less direct dependency value, i.e., Ce=2, than package (H), i.e., Ce=3. However, when we also consider the indirect dependency as well as direct dependency, then their dependency values as per the proposed dependency metric, OutRank, are not in the same order. The OutRank of Package (C) is 0.96 while the OutRank of Package (H) is 0.53. In addition, the instability of package (G) based on the proposed I metric is 0.65, which is relatively 97% more instable than of Martin’s instability metric, i.e., 0.33. Package (G) is the tenth instable package based on Martin’s instability metric. However, it is the fifth instable package as per the proposed instability metric. Likewise, the instability of package (F) based on the proposed I metric is 0.44 which is relatively 12% less instable than of Martin’s instability metric, i.e., 0.5. Package (F) is the sixth instable package based on Martin’s instability metric, and it is the tenth instable package as per the proposed instability metric. Martin’s instability
metric assigned the maximal instable value to any irresponsible packages, e.g., Packages (C), (D), (J), and (K), regardless the level of their dependencies. However, the proposed instability metric takes this into consideration and assigns different instability values to such irresponsible packages. For example, the instability value of Package (J) is 1, while the instability value of Packages (C) and package (D) is 0.97, due to their dependency, i.e. their OutRank = 0.96. Also, the instability value of Package (K) is 0.84, due to its dependency, i.e. its OutRank = 0.76.

Similarly to what is mentioned in the previous examples, Martin’s instability metric assigned the minimal instable value to any independent packages, e.g., Packages (A), (I), (L), and (M), regardless of the level of their responsibilities. However, the proposed instability metric takes this into consideration and assigns different instability values to them. For example, the instability value of Package (A) is 0, while the instability value of Package (M) is 0.49 due to its responsibility, i.e. its InRank = 0.06. Also, because the InRank of package (I) is 0.77 and the InRank of package (L) is 0.11, the instability values of Package (I) and Package (L) are 0.15 and 0.47, respectively. Table 5.3 shows Martin’s instability metric and the proposed instability metric based on the dual ranking method for all of the packages in the third example.
Table 5.3. Comparison between Martin’s instability and the proposed instability metric for the packages in the third motivational example

<table>
<thead>
<tr>
<th>Package</th>
<th>Martin’s Instability</th>
<th>The new Instability</th>
<th>Difference</th>
<th>Difference %</th>
<th>InRank</th>
<th>OutRank</th>
<th>Martin’s Instability Ranking</th>
<th>The new Instability Ranking</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0%</td>
<td>1</td>
<td>0</td>
<td>12</td>
<td>14</td>
</tr>
<tr>
<td>B</td>
<td>0.4</td>
<td>0.44</td>
<td>-0.04</td>
<td>-10%</td>
<td>0.37</td>
<td>0.24</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
<td>0.97</td>
<td>0.03</td>
<td>3%</td>
<td>0</td>
<td>0.96</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>D</td>
<td>1</td>
<td>0.97</td>
<td>0.03</td>
<td>3%</td>
<td>0</td>
<td>0.96</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>E</td>
<td>0.33</td>
<td>0.37</td>
<td>-0.04</td>
<td>-12%</td>
<td>0.42</td>
<td>0.08</td>
<td>10</td>
<td>12</td>
</tr>
<tr>
<td>F</td>
<td>0.5</td>
<td>0.44</td>
<td>0.06</td>
<td>12%</td>
<td>0.27</td>
<td>0.08</td>
<td>6</td>
<td>10</td>
</tr>
<tr>
<td>G</td>
<td>0.33</td>
<td>0.65</td>
<td>-0.32</td>
<td>-97%</td>
<td>0.39</td>
<td>0.69</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>H</td>
<td>0.75</td>
<td>0.62</td>
<td>0.13</td>
<td>17%</td>
<td>0.27</td>
<td>0.53</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>I</td>
<td>0</td>
<td>0.15</td>
<td>-0.15</td>
<td>NA</td>
<td>0.77</td>
<td>0</td>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td>J</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0%</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>K</td>
<td>1</td>
<td>0.84</td>
<td>0.16</td>
<td>16%</td>
<td>0</td>
<td>0.76</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>L</td>
<td>0</td>
<td>0.47</td>
<td>-0.47</td>
<td>NA</td>
<td>0.11</td>
<td>0</td>
<td>12</td>
<td>9</td>
</tr>
<tr>
<td>M</td>
<td>0</td>
<td>0.49</td>
<td>-0.49</td>
<td>NA</td>
<td>0.06</td>
<td>0</td>
<td>12</td>
<td>8</td>
</tr>
<tr>
<td>N</td>
<td>0.5</td>
<td>0.51</td>
<td>-0.01</td>
<td>--2%</td>
<td>0.06</td>
<td>0.08</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>
CHAPTER 6
Software Maintainability and Testability

6.1 Introduction

Many studies have been conducted to correlate software metrics with the external quality attributes of open source software systems such as maintainability and testability. Similarly, research studies have been conducted on package-level design metrics. For example, Aggarwal et al. [121] proposed a maintainability index using package metrics, and Briand et al. [15] defined ratio-scale metrics for cohesion and coupling and their usefulness to the quality of software, especially the maintainability.

These approaches are similar to ours in the sense that they use statistical methods to predict external quality attributes from internal quality attributes. However, most of these studies did not focus exclusively on design metrics. In this study, we begin with Martin's principles and use them to modify his coupling, instability, and abstractness metrics based on indirect dependency. The enhancements are based on a global responsibility factor that has not been previously used in definitions of these two metrics.

6.2 Maintainability

Software maintenance is one of the most challenging and costly parts in the software life-cycle [94]. Clearly, the maintenance of a software project is a major contributor to its overall cost. Maintainability is defined in [95] as “The ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or adapt to a changed
environment.” Accordingly, package maintainability is the probability that a package can be easily modified. During the development of the software, it is difficult to know for sure when and how a package will be modified. However, at that early stage, this probability can be estimated using some internal quality attributes such as coupling, cohesion, and size. Hence, to reduce the future package maintenance cost, we need to construct a prediction model for package maintainability.

Based on the previously mentioned definition of maintainability, a system is easier to maintain if it is simpler to make the required changes. Ease of maintenance refers to the effort that is needed to perform these changes. Some of the efforts done to maintain a package are easily measured, such as Revised Lines of Code (RLOC) [52]. However, many others efforts are harder to measure, such as the effort needed to understand the system or to locate the bug source. The maintenance effort of a package can be measured in different ways. In this study, it is measured as the number of changes made to code during a specific maintenance period, and as the number of line changes (added, modified, or deleted) made in a package within a period of time.

6.3 Testability

With the introduction of agile development methods arose the technique of Test-Driven Development (TDD). TDD is a software development technique that combines program design, coding, and testing (in the form of writing unit tests) in cycles of micro-iterations. Usually, unit tests are created using a unit testing framework, and are fully automated with the help of a Unit Testing Framework (UTF). A system cannot be appropriately maintained without a set of automated tests. These test cases must be executed on a daily basis with every change to the
system. The selected open source systems are unit tested at the package level using the JUnit [97] testing framework. In these systems, test cases in JUnit are written in Java. A typical usage of JUnit is to test each class by developing a corresponding test class dedicated for test purposes only.

Software testability is one of the important aspects for controlling software development activities. To have a high quality software system, it is important to monitor and measure the testability activities from early in the development in order to be able to control testability of the system. By understanding the nature of the relationship of software external quality attributes and package metrics, testability will be more strengthened. We conducted our study in order to achieve a better understanding of what adds to package testability. Testability is defined in [95] as “the degree to which a system or component facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met,” and as “the degree to which a requirement as stated in terms that permit establishment of test criteria and performance of tests to determine whether those criteria have been met.” Also, it is defined in [96] as “attributes of software that bear on the effort needed for validating the modified software.”

Based on the previous definitions of testability, a package is more testable if it is easier to make the required test cases needed to test the package. Ease of test refers to the effort that is needed to perform the test. The test effort of a package can be measured in several different ways. To indicate the testing effort required for a package, we use the size of the corresponding test suite. Software size is used to measure the development cost in many popular cost models such as Boehm’s COCOMO [99] and Putnam’s SLIM model [100]. Therefore, the size of the
corresponding test suite is a good indicator of the testing effort. As a result, the testing effort of a package in this study is measured as the number of lines of code of its corresponding test package.
CHAPTER 7
Theoretical Validation

7.1 Introduction

A major goal when defining object-oriented metrics is to establish a theoretical basis for the metrics. One of the first attempts to provide criteria for complexity metrics was the study made by Weyuker [88]. Weyuker proposed nine properties to evaluate software complexity measures that were written in procedural languages. Since Weyuker’s criteria were proposed before an object-oriented paradigm, they are not appropriate for our study. However, there are two main approaches for a theoretical analysis within software measurement literature to evaluate object-oriented systems. The first approach uses frameworks based on desirable properties (axioms) such as Briand et al.’s property-based measurement framework [10]. The second approach uses frameworks directly based on measurement theory principles such as the DISTANCE framework [87]. These approaches provide frameworks to show that the newly developed metrics using these guidelines are valid and may be used as measurement instruments.

7.2 Property-based measurement framework

Briand, Morasca, and Basili presented a set of mandatory properties for different kinds of metrics to provide a proper framework for the theoretical validation of metrics. The proposed coupling measures were validated theoretically by analyzing their mathematical properties. For example, there are five mathematical properties that are required for a coupling metric to have in
order to prove the usefulness of this measure. They are non-negativity, null value, monotonicity, merging of connected systems, and merging of unconnected systems.

InRank and OutRank are clearly coupling measures. Package (P) in an object-oriented system can be represented as:

\[ S(P) = (E_{in} \cup E_{out}, R) \text{, where } E_{in} \cap E_{out} = \emptyset. \]

Where:

- \( E_{in} \) is the set of classes belonging to a package \( P \).
- \( E_{out} \) contains all the other classes in other packages.
- \( R \) is a Relation which is a subset of \( E_{in} \times E_{out} \).

1) **Non-negativity:**

This property states that the value of the coupling metric of package \( P \) is always non-negative.

\[ \text{CouplingMetric}(P) \geq 0 \]

Clearly, InRank and OutRank fulfil the property, since those both measure:

\[ |\{e_{2} \in E_{out} \mid \exists e_{1} \in E_{in}: (e_{1}, e_{2}) \in R\}|. \]

Also, based on the definition of the newly developed metrics, the value of the InRank and OutRank of a package in an object-oriented system will always be non-negative. Thus, InRank and OutRank satisfy Property 1.
2) *Null value:*

This property states that the value of the coupling metric of package \( P = (E_P \cup E_{out}, R_P) \) is null when \( E_P \) is empty.

\[ E_P = \emptyset \Rightarrow \text{CouplingMetric}(P) = 0 \]

If there is no class in a package or there is no relationship between the classes of a package and any outside classes then InRank and OutRank for this package will be null. So, it can be stated that if \( E_P \) is null then the value of InRank and OutRank are 0. Therefore, InRank and OutRank satisfy property 2.

3) *Monotonicity:*

Let assume that: \( P' = (E_{in} \cup E_{out}, R') \) is identical with package \( P = (E_{in} \cup E_{out}, R) \) except that \( R \subseteq R' \). That is we may be adding some relationships to \( P \), then monotonicity property states that:

\[ \text{CouplingMetric}(P) \leq \text{CouplingMetric}(P') \]

\( R \subseteq R' \) for the InRank metric represents the increase of classes which are dependent on this package. Also, \( R \subseteq R' \) for the OutRank metric represents the increase of classes on which the package depends upon. If an additional relationship is added to any classes of a package, then according to this property the InRank and OutRank of this package must increase or remain the same. It cannot decrease in any circumstances. It is clear that InRank and OutRank fulfil property 3.
4) **Merging of connected systems:**

Let $P'$ be the union of the packages $P_1 = (E_{1in} \cup E_{1out}, R_1)$ and $P_2 = (E_{2in} \cup E_{2out}, R_2)$ such that $P_1$ and $P_2$ are connected in the sense of the relation (there are relationships that exist between $P_1$ and $P_2$), then:

$$\text{CouplingMetric}(P_1) + \text{CouplingMetric}(P_2) \geq \text{CouplingMetric}(P')$$

This property states that the merging of two classes of a package must not increase the coupling metric value because some of the relationships may disappear because of the merging. Because $P' = (E'_{in} \cup E'_{out}, R')$ such that:

- $E'_{in} = E_{1in} \cup E_{2in}$ and $E'_{out} = E_{1out} \cup E_{2out} - E'_{in}$

Since $P_1$ and $P_2$ are connected, then for $\text{InRank}$ and $\text{OutRank}$ either:

- $\exists (e_i, e_0) \in R_1 \mid e_0 \in E_{2in}$
- $\exists (e_i, e_0) \in R_2 \mid e_0 \in E_{1in}$

As the resulting relation $R'$ is a union of $R_1$ and $R_2$ with the elements defined by both equations removed, then clearly the property 4 holds for $\text{InRank}$ and $\text{OutRank}$.

5) **Merging of unconnected systems:**

Let $P'$ be the union of the package $P_1 = (E_{1in} \cup E_{1out}, R_1)$ and $P_2 = (E_{2in} \cup E_{2out}, R_2)$ such that $P_1$ and $P_2$ are not connected in the sense of the relation (no relationships exist between $P_1$ and $P_2$), then:

$$\text{CouplingMetric}(P_1) + \text{CouplingMetric}(P_2) = \text{CouplingMetric}(P')$$
This property states that the merging of two unconnected packages must not increase the coupling metric value of that package. InRank and OutRank cannot increase because the unrelated packages are being captured together in a single package. Since there are no connections, then \( R' = R_1 \cup R_2 \) and \( R_1 \cap R_2 = \emptyset \). As InRank and OutRank measure the size of the right-hand of the image of the relation, then clearly by \( R_1 \cap R_2 = \emptyset \) the property holds for InRank and OutRank.

### 7.3 DISTANCE framework

Frameworks such as DISTANCE [87] ensure that the metrics developed based on their guidelines are proven to be valid distance measures, and that they may be used as ratio scale measurement instruments. It works based on the concepts of distance and dissimilarity. The distance is conceptual, and not physical. Software attributes are modelled as conceptual distances between the software entities they characterize and other software entities that serve as reference points or norms for measurement. In brief, the DISTANCE framework presents a set of mandatory properties, i.e., non-negativity, identity, triangular inequality, and symmetry that need to be satisfied by any metric in order for it to be considered an acceptable measurement-theoretic metric. The construct validity of newly developed metrics is guaranteed by this framework. A metric on a set \( X \) is a distance function \( d: X \times X \to \mathbb{R} \). \( \mathbb{R} \) is the set of real numbers. For all \( x, y, z \) in \( X \), this function is required to satisfy the following conditions:

- Non-negativity: \( d(x, y) \geq 0 \).
- Identity: \( d(x, y) = 0 \) if and only if \( x = y \).
- Symmetry: \( d(x, y) = d(y, x) \).
subadditivity / triangle inequality: \( d(x, z) \leq d(x, y) + d(y, z) \).

<table>
<thead>
<tr>
<th></th>
<th>InRank</th>
<th>OutRank</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-negativity</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Null value</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Monotonicity</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Merging of connected systems</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Merging of unconnected systems</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Distance non-negativity</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>identity</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>symmetry</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>subadditivity / triangle inequality</td>
<td>✔</td>
<td>✔</td>
</tr>
</tbody>
</table>

The non-negativity and identity axioms require the difference of at least two equivalence packages: one containing pairs of abstractions having no distance between them and one containing pairs of abstractions with a positive distance between them. Table 7.1 summarizes the theoretical properties of the newly developed metrics obtained by following the steps proposed in [87]. Hence, the four mandatory properties required by DISTANCE framework hold for InRank and OutRank.
Given the introduction of a set of theoretically valid object-oriented software metrics, a case study that shows their usefulness will be shown in the next section. However, we need to empirically validate their usefulness in our future work. Some initial results that have been obtained from real well-known open source systems encourage completing the experimental study of the newly developed metrics.
CHAPTER 8

Experimental Design

8.1 Introduction

Research has been done to estimate the maintainability and testability of open source software systems. Much of this research has been applied at the end of the coding stage. The measurement of maintainability and testability during early stages of the software lifecycle, e.g., during the design phase, could help in decreasing the maintenance and testing effort to a great extent. One promising approach for early identification of costly packages is to construct prediction models using object-oriented design metrics. The usefulness of these metrics could be verified using historical data. These models could be used to identify potentially high maintenance and expensive-to-test packages in systems before their release. The usage of design metrics would allow the developers and system engineers to take immediate actions, and accordingly potentially avoid unnecessary costly refactoring.

In specific, this research work set out to study the relationship between the newly developed metrics of package coupling and the maintenance and test efforts. Also, this research will assess the capability of these metrics to predict the two software quality characteristics, maintainability and testability. Thus, we are assessing whether these metrics can predict the required amount of effort needed for maintaining and testing a package. This experiment also aims to evaluate the new metrics and compare them to Martin’s metrics. It was believed necessary to include a number of real-world large systems in the analysis to see if the results that
have been obtained from the test cases are scalable to large systems. For this purpose, the newly developed metrics were applied to seven open source software systems by using our customized tool. The performed empirical validation demonstrates the relationships between the new metrics and package maintainability and testability. Also, we did a comparative analysis between the new metrics and Martin’s metrics. In general, the expectation is that loosely coupled packages in a system require less effort to maintain and test. Thus, one set of analyses will examine the relationship between software package coupling, instability, abstractness, and distance as measured by the new package metrics and package maintainability. Four systems, Camel 2.0.0 [112], Tomcat 7.0.6 [91], JHotDraw 7.5.1 [92], and JEdit 4.5.0 [113], provided four data sets for the analyses of software maintainability. The analysis for each data set mirrored that of the other systems. The fact is that the data sets have similar characteristics and the need to have a larger sample size to regression analysis run on the integrated data from the four systems. Regression analysis includes package size (number of classes in a package) as part of the prediction model to control for effect and improve model prediction. The other set of analysis looks into the relation between software package coupling, instability, abstractness, and distance as measured by the new package metrics and package testability (TLOC). Five systems, Camel 2.0.0 [112], Tomcat 7.0.6 [91], Hadoop 2.2.0 [93], Synapse 2.1.0 [114], and Ant 1.92 [115], provide five data sets for the analyses of software testability. Similar to package maintainability analysis, the analysis for testability mirrored each one of the data sets provided by the five systems, Camel 2.0.0, Tomcat 7.0.6, Hadoop 2.2.0, Synapse 2.1.0, and Ant 1.92. Similarly, regression analysis was run on the integrated data from these five systems. Regression analysis includes package size (number of
classes in a package) as part of the prediction model to control for effect and improve model prediction.

8.2 Data set

Table 8.1 summarizes the seven open-source systems under study. For the maintainability study, we investigated the change history over multiple years of the development of the four popular open source systems: Camel 2.0.0 [112], Tomcat 7.0.6 [91], JHotDraw 7.5.1 [92], and JEdit 4.5.0 [113]. The details of the studied systems in the maintainability experiment are given in Table 8.2.

<table>
<thead>
<tr>
<th>System</th>
<th>Domain</th>
<th># studied Packages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Camel 2.0.0</td>
<td>rule-based routing and mediation engine</td>
<td>264</td>
</tr>
<tr>
<td>Tomcat 7.0.6</td>
<td>Java web server</td>
<td>113</td>
</tr>
<tr>
<td>JHotDraw 7.5.1</td>
<td>2-D graphics framework</td>
<td>65</td>
</tr>
<tr>
<td>JEdit 4.5.0</td>
<td>Java text editor</td>
<td>35</td>
</tr>
<tr>
<td>Hadoop 2.2.0</td>
<td>Big data distributed computing framework</td>
<td>221</td>
</tr>
<tr>
<td>Synapse 2.1.0</td>
<td>lightweight open source Enterprise Service Bus (ESB)</td>
<td>117</td>
</tr>
<tr>
<td>Ant 1.92</td>
<td>Java library and command-line tool</td>
<td>67</td>
</tr>
</tbody>
</table>

Table 8.1: The descriptions of the studied systems
Table 8.2: Maintainability Studied Systems details

<table>
<thead>
<tr>
<th></th>
<th>Camel</th>
<th>Tomcat</th>
<th>JHotDraw</th>
<th>JEdit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base Release</td>
<td>2.0.0</td>
<td>7.0.6</td>
<td>7.5</td>
<td>4.5.0</td>
</tr>
<tr>
<td>Base Release Date</td>
<td>08/24/2009</td>
<td>01/14/2011</td>
<td>07/29/2010</td>
<td>01/31/2012</td>
</tr>
<tr>
<td>End Release</td>
<td>2.2.0</td>
<td>7.5</td>
<td>7.6</td>
<td>5.1.0</td>
</tr>
<tr>
<td>Revisions</td>
<td>1614</td>
<td>636</td>
<td>354</td>
<td>323</td>
</tr>
<tr>
<td>RLOC</td>
<td>60,688</td>
<td>22,027</td>
<td>21,857</td>
<td>9,981</td>
</tr>
</tbody>
</table>

Table 8.3: Testability Studied Systems details

<table>
<thead>
<tr>
<th></th>
<th>Camel</th>
<th>Tomcat</th>
<th>Hadoop</th>
<th>Synapse</th>
<th>Ant</th>
</tr>
</thead>
<tbody>
<tr>
<td>Release</td>
<td>2.0.0</td>
<td>7.0.6</td>
<td>2.2.0</td>
<td>2.1.0</td>
<td>1.92</td>
</tr>
<tr>
<td>Release Date</td>
<td>08/24/2009</td>
<td>01/14/2011</td>
<td>10/15/2013</td>
<td>01/06/2012</td>
<td>07/12/2013</td>
</tr>
<tr>
<td>TLOC</td>
<td>31,494</td>
<td>10,732</td>
<td>181,007</td>
<td>13,177</td>
<td>23,170</td>
</tr>
</tbody>
</table>

On the other side, the testability study was also performed on five large and mature open source systems: Camel 2.0.0, Tomcat 7.0.6, Hadoop 2.2.0, Synapse 2.1.0, and Ant 1.92. Detailed information about the studied systems in the testability study is given in Table 8.3.
Our case studies are large Java projects in each study, and all of them are mature and successful with years of development and are well-known in the open-source and software-engineering-research communities. Since open source systems are freely available for download and are very useful for the purpose of research, we chose open source projects to avoid the constraints associated with obtaining permission to study the commercially developed projects. In addition, these systems were also chosen because they offer a sufficient repository that tracks the evolution of the system. The studied systems give free access to their repository, which includes the historical data about many of their artifacts. In addition, Camel, Tomcat, Hadoop, Synapse, and Ant provide the source code of their test classes which are used for unit testing. Also, their size is sufficient to obtain adequate sample sizes (measured by the number of packages). Camel 2.0.0 consists of 264 packages, the Tomcat 7.0.6 system consists of 113 packages, the JHotDraw 7.5.1 system consists of 65 packages, and JEdit 4.5.0 consists of 35 packages. Also, the Hadoop 2.2.0 system consists of 221 packages, Synapse 2.1.0 consists of 117 packages, and Ant 1.92 consists of 67 packages. All systems were implemented in Java. Java is a pure object-oriented language that is at present one of the most common programming languages with large community support. The reason for selecting the Java system is that most of the tools available for measuring the related metrics operate on a Java system.

8.3 Tools

Design documents of the systems under study were not readily available; thus, we had to extract many different design diagrams, e.g. class diagrams, package diagrams, and quality metrics, by reverse engineering the source code of the studied instances. We’ve also implemented a tool that calculates coupling metrics, instability, and the abstractness of both
metric suites. The first tool that has been used is the ObjectAid UML Explorer [105]. This tool has been used to extract the UML diagrams from the Java source code. The JHawk tool [104] is a general-purpose metrics collection tool that calculates a variety of metrics from OO systems. We’ve used the JHawk tool to extract size and complexity-based metrics from each system, and to validate the other tools’ calculations. JDepend [106] has been used to generate design quality metrics for each package in the system. We used JDepend to verify the relations between the packages. The third tool we have used is Classycle [107], which analyses the static class and package dependencies in Java applications. A customized tool has been implemented to integrate and analyze the data from the previous tools and compute the new coupling, instability, and abstractness metrics for every package in the system. In addition, this tool computes Martin’s instability and abstractness metrics and does the comparative study between the new metrics and Martin’s metrics.

For collecting maintainability and testability information, we have downloaded and accessed the systems’ SVN[108] repositories by using a subversion client tool, TortoiseSVN[109]. For each system in the maintainability study, we studied every log of the SVN repository during the studied maintenance period, which included many revisions. Every revision has a revision number and a list of the classes and objects that were affected in this revision. Every class listed in a revision has detailed information of the changes that have been applied to it. For each system in the testability study, we downloaded every test class in the system and assigned it to its source class. After having the maintainability and testability data ready at the class level, we assigned every class’s data to its package in order to have package-level data for our study.
For more than three months, two full-time Computer Science PhD students were assigned to conduct the data collection part. The first author was one of them. They had to manually collect the maintainability and testability data. We followed a manual collection of the data from the studied systems’ repositories to ensure more accurate results. Then we performed a validation of the collected data by randomly double checking the collected data. This process enhanced the validity of the experiment.

8.4 Maintenance and Test metrics

We needed to quantify the maintenance and testing efforts. Several measures have been suggested to measure different aspects of maintenance, such as number of revisions [101], the Revised Lines of Code (RLOC) [102], and pieces of code potentially affected by the revised code [116, 117]. Also, some researchers studied the correlation between these maintenance measures and maintenance effort. In this study, we followed the recommendation of Morasca [85] of using probabilities to estimate the two external attributes, namely, package maintainability and package testability. These probabilistic models use the newly developed metrics as independent variables, and estimate the probability that a package will be frequently modified (REVISION) or will require costly modifications (RLOC). Building these probability estimation models involves the collection of historical data of actual revisions performed on these systems’ packages during the maintenance phase. We thus used the number of revisions (REVISIONS) [101] that are made to a package and the number of Revised Lines of Code (RLOC) during the studied maintenance period to indicate the maintenance effort required for a package. RLOC was found to correlate with both maintenance cost [102] and maintenance effort.
The package maintainability metric was measured with the RLOC metric by counting the number of revised lines of code in the package code. RLOC was defined as the number of line changes (added, modified or deleted) made in a package within a period of time. We followed Li and Henry’s suggestion in [52] to count a change in line content as a deletion and an addition. Also, the maintainability of a package was measured with the REVISIONS metric by counting the number of changes made to its source code, or the number of revisions of this package during a maintenance period. A lower number of package revisions (REVISIONS) and a smaller number of revised lines of codes (RLOC) during the package maintenance history indicate less package maintenance effort which means having high maintainability. The Revised Lines of Code (RLOC) and the number of revisions (REVISIONS) are measured on the main trunk branch of the systems’ repositories.

We followed Singh’s [118] and Alvi’s suggestions [119] to use the size of the corresponding test suite to indicate the testing effort required for a package. The authors believe that the size of the corresponding test suite is a good indicator of the testing effort. As a result, the proposed test metrics for a package in our experiments is TLOC, number of lines of code of its test package. This metric is calculated from the JUnit test classes of the test packages of the five systems. A smaller number of lines of code of test package (TLOC) indicate less package test effort which means having a high testability.

8.5 **Research Hypotheses and Questions**

Maintenance and testing effort measurements are the dependent variables which will be used in this experiment and were defined precisely before in this section.
The research hypotheses will be tested through quantitative research data in the form of case studies that measures the accuracy of the proposed metrics through the history of theses evolving systems. In addition, this research seeks to answer some of the following questions:

1- Is there any relationship between the proposed responsibility metric, called InRank, and how often package change?

2- Is there any relationship between the proposed independence metric, called OutRank, and how often package change?

3- Is there any relationship between the proposed responsibility metric, called InRank, and stability of the system?

4- Is there any relationship between the proposed independence metric, called OutRank, and stability of the system?

5- Is there any relationship between the proposed responsibility metric, called InRank, and testability?

6- Is there any relationship between the proposed independence metric, called OutRank, and testability?

7- Is there any relationship between the proposed instability metric and maintainability?

8- Is there any relationship between the proposed instability metric and testability?

9- Is there any relationship between the proposed distance metric and maintainability?

10- Is there any relationship between the proposed distance metric and testability?

Based on these questions, we drew up the following hypotheses that would be tested by this experiment in order to establish the relationship between the new developed metrics and maintenance and test effort of a package:
• Hypothesis 1:

\(H_0(\text{InRank, REVISION})\): Coupling of package (InRank) does not contribute to its maintenance cost (REVISION) and requires no more maintenance cost than its peer.

\(H_1(\text{InRank, REVISION})\): A package that is highly coupled (InRank) is more costly to maintain (REVISION) than its peer.

• Hypothesis 2:

\(H_0(\text{InRank, RLOC})\): Coupling of package (InRank) does not contribute to its maintenance cost (RLOC) and requires no more maintenance cost than its peer.

\(H_1(\text{InRank, RLOC})\): A package that is highly coupled (InRank) is more costly to maintain (RLOC) than its peer.

• Hypothesis 3:

\(H_0(\text{OutRank, REVISION})\): Coupling of package (OutRank) does not contribute to its maintenance cost (REVISION) and requires no more maintenance cost than its peer.

\(H_1(\text{OutRank, REVISION})\): A package that is highly coupled (OutRank) is more costly to maintain (REVISION) than its peer.
• Hypothesis 4:

\[ H_0(\text{OutRank, RLOC}) : \text{Coupling of package (OutRank) does not contribute to its maintenance cost (RLOC) and requires no more maintenance cost than its peer.} \]

\[ H_1(\text{OutRank, RLOC}) : \text{A package that is highly coupled (OutRank) is more costly to maintain (RLOC) than its peer.} \]

• Hypothesis 5:

\[ H_0(\text{Dual_Instability, REVISION}) : \text{Instability of package does not contribute to its maintenance cost (REVISION) and requires no more maintenance cost than its peer.} \]

\[ H_1(\text{Dual_Instability, REVISION}) : \text{A package that is highly stable is more costly to maintain (REVISION) than its peer.} \]

• Hypothesis 6:

\[ H_0(\text{Dual_Instability, RLOC}) : \text{Instability of package does not contribute to its maintenance cost (RLOC) and requires no more maintenance cost than its peer.} \]

\[ H_1(\text{Dual_Instability, RLOC}) : \text{A package that is highly stable is more costly to maintain (RLOC) than its peer.} \]
• Hypothesis 7:

\( H_0(\text{Dual Abstractness, REVISION}) \): Abstractness of package does not contribute to its maintenance cost (REVISION) and requires no more maintenance cost than its peer.

\( H_1(\text{Dual Abstractness, REVISION}) \): A package that is highly abstract is more costly to maintain (REVISION) than its peer.

• Hypothesis 8:

\( H_0(\text{Dual Abstractness, RLOC}) \): Abstractness of package (OutRank) does not contribute to its maintenance cost (RLOC) and requires no more maintenance cost than its peer.

\( H_1(\text{Dual Abstractness, RLOC}) \): A package that is highly abstract is more costly to maintain (RLOC) than its peer.

• Hypothesis 9:

\( H_0(\text{Dual Distance, REVISION}) \): Distance of package from the main sequence does not contribute to its maintenance cost (REVISION) and requires no more maintenance cost than its peer.

\( H_1(\text{Dual Distance, REVISION}) \): A package that is far from the main sequence is more costly to maintain (REVISION) than its peer.
• Hypothesis 10:

\[ H_0(\text{Dual\_Distance, RLOC}): \text{Distance of package from the main sequence does not contribute to its maintenance cost (RLOC) and requires no more maintenance cost than its peer.} \]

\[ H_1(\text{Dual\_Distance, RLOC}): \text{A package that is far from the main sequence is more costly to maintain (RLOC) than its peer.} \]

• Hypothesis 11:

\[ H_0(\text{InRank, TLOC}): \text{Coupling of package (InRank) does not contribute to its test cost (TLOC) and requires no more test cost than its peer.} \]

\[ H_1(\text{InRank, TLOC}): \text{A package that is highly coupled (InRank) is more costly to test (TLOC) than its peer.} \]

• Hypothesis 12:

\[ H_0(\text{OutRank, TLOC}): \text{Coupling of package (OutRank) does not contribute to its test cost (TLOC) and requires no more test cost than its peer.} \]

\[ H_1(\text{OutRank, TLOC}): \text{A package that is highly coupled (OutRank) is more costly to test (TLOC) than its peer.} \]

• Hypothesis 13:

\[ H_0(\text{Dual\_Instability, TLOC}): \text{Instability of package does not contribute to its test cost (TLOC) and requires no more maintenance cost than its peer.} \]
H1(Dual_Instability, TLOC): A package that is highly stable is more costly to test (TLOC) than its peer.

- Hypothesis 14:

  H0(Dual_Abstractness, TLOC): Abstractness of package (OutRank) does not contribute to its test cost (TLOC) and requires no more test cost than its peer.

  H1(Dual_Abstractness, TLOC): A package that is highly abstract is more costly to test (TLOC) than its peer.

- Hypothesis 15:

  H0(Dual_Distance, RLOC): Distance of package from the main sequence does not contribute to its maintenance cost (RLOC) and requires no more maintenance cost than its peer.

  H1(Dual_Distance, RLOC): A package that is far from the main sequence is more costly to maintain (RLOC) than its peer.

8.6 Expected Result

Developing a metric can be a relatively easy target to achieve. The major difficulty is to understand its results in a meaningful way and then be able to prove its validity. A metric is considered valid if it numerically captures the behavior we observe in the empirical world. In general, the expectation is that tightly coupled systems require more effort to maintain, and are
more expensive to test. In the following, we are setting our expectations for the new metrics that are supported by the theory.

Briand et al. [111] have provided a theoretical basis for developing quantitative models relating object-oriented metrics and external quality metrics. It is assumed that the structural properties, internal attributes, of a software component, i.e. coupling, have an impact on its cognitive complexity. Higher cognitive complexity leads to a decrease in the external quality attributes of those components such as external qualities, reduced maintainability, and testability.

In addition, Ammann and Offutt Offutt in [110] describe how graphs can be used as abstractions to represent software code, “Accordingly, a graph-based coverage criterion evaluates a test set for an artifact in terms of how the paths corresponding to the test cases ‘cover’ the artifact’s graph abstraction,” hence, if the number of relationships increases in a package, it means that the number of test cases also increase. As a result, we are expecting a positive relationship between the studied metrics and maintenance and test cost. This means that higher values for these coupling metrics represent structural properties that increase the probability that a package will have a high maintenance and test cost.

Managing the change is a very important factor in reducing the maintenance cost. One important reason for reducing the coupling between packages is the certainty that it will decrease the probability that change to one package will ripple up to another. As a result, we concluded that it is expected to have a positive relationship between coupling metrics and maintenance and testing effort.

It is also expected that the level of coupling (InRank and OutRank) defines the required maintenance effort. A change that is contained within a small number of packages is likely to be
less costly to fix because less code is altered, reducing the risk of introducing new defects. Also, it is easier for the developer to understand the related packages in order to perform the change. So reduced coupling (Low InRank and OutRank) can ease maintenance by making the system easier to maintain and to understand. Packages with a high number of dependencies, i.e. high OutRank, subsequently increase the coupling, decreasing the system overall quality. Due to the high number of external dependencies it has, a package with high OutRank is more often changed when any other package is changed. Since it has many sources for change, it is very likely to change. These kinds of packages indicate a lack of modularity in the system, and hence needs to be avoided. Also, packages with a high number of dependents, i.e. high InRank, subsequently increase the coupling, decreasing the system’s overall quality. Hence, the number of dependents of a package gives an indication of how many packages in the system are affected if this package is touched. So if this package is changed, a significant impact will be made on many packages in the system. Identifying these two kinds of packages helps to locate packages that need to be considered as the first candidates for refactoring.

Martin defined the word stability as “not easily moved.”[56]. Martin claims that stability is not a measure of the likelihood of change; rather it is a measure of how difficult it is to change a package. So packages that are more difficult to change, i.e. stable, are going to be less volatile. In addition, the stability metric is an indicator of the package's resilience to change. Therefore, we expect to get a positive correlation between the highly instable packages and the maintenance effort. Therefore, for the instability metric to be valid, it must provide a reasonable measurements of volatility in known situations. So the volatility of each package must be estimated, and then compared to the instability; if there is a directional correlation between the two, we can determine
that the metric is reasonable. Therefore, based on Martin’s assumptions, we expect to find volatile packages tend to have high instability values, and nonvolatile packages tend to have low stability values. In other word, packages which change less tend to have more dependent packages than depended upon packages. Also, packages which change more frequent tend to have more depended upon packages than dependent packages. It is important in this study to quantify the volatility of packages. Having accepted that volatile packages are the packages which change often, the volatility is measured in terms of the number of actual change during the period of the study.

It's important that the packages that are most likely to undergo changes in requirements are the packages into which we've put the most flexibility. On the other hand, a stable package has a lot of other packages that depend on it. Since there is huge impact on its dependent packages, it is hard to change it. It needs to be flexible enough to be extended without modification by the help of the abstract classes [56]. Therefore, packages with high abstractness values are easily extensible. So in good systems, we expect to find a positive correlation between the high abstract packages and the maintenance effort.

Packages with a high value of Distance (Dual_D) are either in the zone of pain or in the zone of uselessness. If a package is in the zone of pain, it is highly stable and totally concrete. Although such packages are rigid and cannot be extended as they are not abstract, they are also difficult to change, due to their stability. If a package is in the zone of uselessness, it is highly instable and totally abstract. Although such packages are very flexible and can be easily extended as they are abstract, they are also easy to change, due to their instability. Due to these two different situations, we are expecting that we may have a weak positive correlation between the distance (Dual_D)
and the maintenance effort. The author is considering modifying the distance (Dual_D) in such a way that it can differentiate between these two different zones.
CHAPTER 9
Empirical Analysis and Results

9.1 Introduction

In this chapter, we describe the methodology used to analyze the newly developed metrics and maintainability and testability data collected for the studied systems. The analysis procedures for maintainability and testability studies involve analysis of the descriptive statistics, statistical analyses, principal component analysis, and linear multivariate regression analysis against the maintainability and testability effort. We now present these techniques for the maintainability and testability measures separately in some detail.

9.2 Maintainability

The goals of the maintainability studies are to identify any relationship between the newly developed metrics and maintenance effort as measured by the number of revisions (REVISION) and number of revised lines of code (RLOC), and to find whether there is a compact model containing some of these metrics that can predict the system maintainability.

Data screening and evaluation of linearity for assessing correlations and regression analysis led to the natural log transformation of the following variables: number of revisions (REVISION), package size (number of classes in a package), and number of revised lines of code (RLOC). Martin’s package metric suite is included in the list of variables for the purpose of comparison.
9.2.1 Descriptive statistics

Table 9.1 provides descriptive statistics (mean and standard deviation) for the variables used in analyzing package maintainability across the four systems: Camel 2.0.0, JHotDraw 7.5.1, JEdit 4.5.0, and Tomcat 7.0.6.

Table 9.1. Means and Standard Deviation of the Variables Used in Maintainability Analyses

<table>
<thead>
<tr>
<th>Variable</th>
<th>Camel 2.0.0 N=264</th>
<th>JHotDraw 7.5.1 N=65</th>
<th>JEdit 4.5.0 N=35</th>
<th>Tomcat 7.0.6 N=113</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mean</td>
<td>S.D.</td>
<td>Mean</td>
<td>S.D.</td>
</tr>
<tr>
<td>InRank</td>
<td>0.01</td>
<td>0.07</td>
<td>0.1</td>
<td>0.2</td>
</tr>
<tr>
<td>Ca</td>
<td>17.6</td>
<td>127</td>
<td>20.6</td>
<td>39.2</td>
</tr>
<tr>
<td>OutRank</td>
<td>0.06</td>
<td>0.1</td>
<td>0.15</td>
<td>0.24</td>
</tr>
<tr>
<td>Ce</td>
<td>9.9</td>
<td>13.9</td>
<td>19.1</td>
<td>23.6</td>
</tr>
<tr>
<td>Dual Instability</td>
<td>0.93</td>
<td>0.19</td>
<td>0.44</td>
<td>0.41</td>
</tr>
<tr>
<td>Instability</td>
<td>0.86</td>
<td>0.27</td>
<td>0.47</td>
<td>0.36</td>
</tr>
<tr>
<td>Dual Abstr.</td>
<td>0.17</td>
<td>0.32</td>
<td>0.27</td>
<td>0.36</td>
</tr>
<tr>
<td>Abstractness</td>
<td>0.08</td>
<td>0.18</td>
<td>0.10</td>
<td>0.17</td>
</tr>
<tr>
<td>Dual Distance</td>
<td>0.17</td>
<td>0.29</td>
<td>0.46</td>
<td>0.38</td>
</tr>
<tr>
<td>Distance</td>
<td>0.13</td>
<td>0.21</td>
<td>0.43</td>
<td>0.35</td>
</tr>
<tr>
<td>Classes</td>
<td>13.7</td>
<td>29.7</td>
<td>16.3</td>
<td>18.3</td>
</tr>
<tr>
<td>LogClasses</td>
<td>2.0</td>
<td>1.0</td>
<td>2.36</td>
<td>1.00</td>
</tr>
</tbody>
</table>
### 9.2.2 Statistical Analysis

The main hypotheses are tested in this section. The first set of hypotheses states that there is a relationship between the newly developed metrics and the maintenance effort as measured by the number of revisions (REVISION) and number of revised lines of code (RLOC). In order to evaluate the above hypotheses, we applied correlation analysis to compute the coefficient of correlation(r) between the two metrics suites, the new metrics and Martin metrics, and the maintenance effort per package. This is a commonly used statistical measure to determine the dependency of a dependent variable on explanatory variables. We decided to use Spearman’s rank-order correlation coefficient (Spearman Rho) over Pearson’s correlation coefficient due to the fact that the three dependent variables are not normally distributed and the nonparametric nature of the studied package metrics in this study. Spearman’s rank-order correlation coefficient is the proper measure of a bivariate relationship when normality and linearity conditions for the Pearson’s product moment correlation do not hold.

For this study, the Spearman Rho correlation provided a measure of association between the new package metrics, Martin’s package metric suite, package size (classes), and the two measures of package maintainability, number of package revisions (REVISION) and the number of revised lines of code (RLOC), within each of the four data sets. The value of \( r_s \) ranges from -1 to 1.
to 1 where the value of $r$, +1, indicates a perfect positive correlation while the value of $r$, -1, indicated a perfect negative correlation. A value of 0 indicates that there is no correlation; that is the two variants are independent of each other. Tables 9.2-9.5 provide the list of these correlations for the four sets of data. In addition, we estimated the statistical significance of $r$s by calculating the t statistic. This test provided a significance $p$. The significance $p$ indicates the probability that the observed value is a chance event based on the number of pairs in the data set. The significance $p$ allows us to reject the null hypothesis ($H_0(m;n)$) and accept the alternative hypothesis ($H_1(m;n)$) with a certain level of confidence.

Tables 9.2-9.5 present the correlations of the newly developed metrics for the four systems. We calculated the value of $r$s from the arithmetic average of all the correlation matrixes. This enabled us to specify the followings observations:

- There are medium correlations between InRank and all the metrics.
- There is no strong correlation between OutRank and any other metric. This means that there is no linear association between OutRank and the other metrics.
- There is a medium correlation among dual abstractness and dual instability.
- There is a correlation among dual distance and dual instability.

From these observations, it follows that these metrics, as expected from their definitions, are not totally independent and may represent redundant information. However, there are no strong correlations between the newly developed metrics across the studied systems and hence they are not highly correlated which indicate that there is no multicollinearity. Multicollinearity will be checked later on in this chapter.
Table 9.2 Correlation of the studied Metrics For Camel2.0.0

<table>
<thead>
<tr>
<th>Metrics</th>
<th>InRank</th>
<th>Ca</th>
<th>OutRank</th>
<th>Ce</th>
<th>Dual Instability</th>
<th>Instability</th>
<th>Dual Abstractness</th>
<th>Abstractness</th>
<th>Dual Distance</th>
<th>Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>InRank</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(sig.)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ca</td>
<td>.99**</td>
<td></td>
<td>.99**</td>
<td>.000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(sig.)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OutRank</td>
<td>.23**</td>
<td>.22**</td>
<td>.23**</td>
<td>.000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(sig.)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ce</td>
<td>.27**</td>
<td>.26**</td>
<td>.27**</td>
<td>.000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(sig.)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dual Instability</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(sig.)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instability</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(sig.)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dual Abstractness</td>
<td>.46**</td>
<td>.47**</td>
<td>.46**</td>
<td>.000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>.84**</td>
</tr>
<tr>
<td>(sig.)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Abstractness</td>
<td>.46**</td>
<td>.46**</td>
<td>.46**</td>
<td>.000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>.84**</td>
</tr>
<tr>
<td>(sig.)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dual Distance</td>
<td>.65**</td>
<td>.66**</td>
<td>.65**</td>
<td>.000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>.84**</td>
</tr>
<tr>
<td>(sig.)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Distance</td>
<td>.72**</td>
<td>.73**</td>
<td>.72**</td>
<td>.000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>.83**</td>
</tr>
<tr>
<td>(sig.)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 9.3 Correlation of the studied Metrics For JEdit 4.5.0

<table>
<thead>
<tr>
<th>Metrics</th>
<th>InRank</th>
<th>Ca</th>
<th>OutRank</th>
<th>Ce</th>
<th>Dual Instability</th>
<th>Instability</th>
<th>Dual Abstractness</th>
<th>Abstractness</th>
<th>Dual Distance</th>
<th>Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>InRank</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(sig.)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ca</td>
<td>.96**</td>
<td>.62**</td>
<td>.45**</td>
<td>.47**</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(sig.)</td>
<td>.000</td>
<td>.000</td>
<td>.007</td>
<td>.004</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OutRank</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(sig.)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ce</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(sig.)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dual Instability</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(sig.)</td>
<td>.82**</td>
<td>.68**</td>
<td>.000</td>
<td>.000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instability</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(sig.)</td>
<td>.90**</td>
<td>.78**</td>
<td>.000</td>
<td>.000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dual Abstractness</td>
<td>.46**</td>
<td>.44**</td>
<td>.34*</td>
<td>.37*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(sig.)</td>
<td>.006</td>
<td>.008</td>
<td>.050</td>
<td>.027</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Abstractness</td>
<td>.63**</td>
<td>.51**</td>
<td>.01</td>
<td>.07</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(sig.)</td>
<td>.000</td>
<td>.002</td>
<td>.935</td>
<td>.707</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dual Distance</td>
<td>.54**</td>
<td>.42*</td>
<td>.04</td>
<td>.08</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(sig.)</td>
<td>.001</td>
<td>.011</td>
<td>.806</td>
<td>.656</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Distance</td>
<td>.86**</td>
<td>.76**</td>
<td>.14</td>
<td>.14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(sig.)</td>
<td>.000</td>
<td>.000</td>
<td>.431</td>
<td>.412</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 9.4 Correlation of the studied Metrics For JHotDraw 7.5.1

<table>
<thead>
<tr>
<th>Metrics</th>
<th>InRank</th>
<th>Ca</th>
<th>OutRank</th>
<th>Ce</th>
<th>Instability</th>
<th>Abstractness</th>
<th>Distance</th>
<th>Dual Distance</th>
<th>Abstractness</th>
<th>Dual Abstractness</th>
<th>Instability</th>
<th>Dual Instability</th>
<th>Dual Distance</th>
<th>Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>InRank</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(sig.)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Ca</strong></td>
<td>.90**</td>
<td></td>
<td></td>
<td>.95**</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(sig.)</td>
<td>.000</td>
<td></td>
<td></td>
<td>.000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>OutRank</strong></td>
<td>-.27*</td>
<td>-.073</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(sig.)</td>
<td>.028</td>
<td>.565</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Ce</strong></td>
<td>-.20</td>
<td>-.010</td>
<td></td>
<td>.95**</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(sig.)</td>
<td>.114</td>
<td>.935</td>
<td></td>
<td>.000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Dual Instability</strong></td>
<td>-</td>
<td>-</td>
<td>.87**</td>
<td>.78**</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(sig.)</td>
<td>.000</td>
<td>.001</td>
<td>.000</td>
<td>.000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Instability</strong></td>
<td>-</td>
<td>-</td>
<td>.75**</td>
<td>.73**</td>
<td>.91**</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(sig.)</td>
<td>.000</td>
<td>.000</td>
<td>.000</td>
<td>.000</td>
<td>.000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Dual Abstractness</strong></td>
<td>.61**</td>
<td>.62**</td>
<td>.21</td>
<td>.24</td>
<td>-.04</td>
<td>-.17</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(sig.)</td>
<td>.000</td>
<td>.000</td>
<td>.087</td>
<td>.054</td>
<td>.761</td>
<td>.188</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Abstractness</strong></td>
<td>.58**</td>
<td>.54**</td>
<td>.17</td>
<td>.19</td>
<td>-.07</td>
<td>-.15</td>
<td>.93**</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(sig.)</td>
<td>.000</td>
<td>.000</td>
<td>.171</td>
<td>.139</td>
<td>.602</td>
<td>.234</td>
<td>.000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Dual Distance</strong></td>
<td>.23</td>
<td>.11</td>
<td>-.04**</td>
<td>-.07**</td>
<td>-.19</td>
<td>-.18</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(sig.)</td>
<td>.068</td>
<td>.398</td>
<td>.000</td>
<td>.000</td>
<td>.000</td>
<td>.121</td>
<td>.142</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Distance</strong></td>
<td>.47**</td>
<td>.42**</td>
<td>-.05</td>
<td>-.11</td>
<td>.71**</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(sig.)</td>
<td>.000</td>
<td>.000</td>
<td>.000</td>
<td>.000</td>
<td>.000</td>
<td>.714</td>
<td>.395</td>
<td>.000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Notes:**
- `**` indicates significance at the .01 level.
- `*` indicates significance at the .05 level.
- All correlations are two-tailed.
<table>
<thead>
<tr>
<th>Metrics</th>
<th>InRank</th>
<th>Ca</th>
<th>OutRank</th>
<th>Ce</th>
<th>Dual Instability</th>
<th>Instability</th>
<th>Dual Abstractness</th>
<th>Abstractness</th>
<th>Dual Distance</th>
<th>Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>InRank</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(sig.)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ca</td>
<td>.94**</td>
<td>.00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(sig.)</td>
<td>.000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OutRank</td>
<td>.18</td>
<td>.27**</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(sig.)</td>
<td>.061</td>
<td>.003</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ce</td>
<td>.14</td>
<td>.22*</td>
<td>.87**</td>
<td></td>
<td></td>
<td>.14</td>
<td>.22*</td>
<td>.24**</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(sig.)</td>
<td>.150</td>
<td>.021</td>
<td>.000</td>
<td></td>
<td></td>
<td>.000</td>
<td></td>
<td>.000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dual Instability</td>
<td>*</td>
<td>*</td>
<td>.21*</td>
<td>.24**</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(sig.)</td>
<td>.80**</td>
<td>.71**</td>
<td>.026</td>
<td>.010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instability</td>
<td>*</td>
<td>*</td>
<td>.11</td>
<td>.22*</td>
<td>.94**</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(sig.)</td>
<td>.82**</td>
<td>.80**</td>
<td>.247</td>
<td>.018</td>
<td>.000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dual Abstractness</td>
<td>.40**</td>
<td>.40**</td>
<td>.18</td>
<td>.19*</td>
<td>.33**</td>
<td>.37**</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(sig.)</td>
<td>.000</td>
<td>.000</td>
<td>.063</td>
<td>.047</td>
<td>.000</td>
<td>.000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Abstractness</td>
<td>.47**</td>
<td>.45**</td>
<td>.99</td>
<td>.06</td>
<td>.41**</td>
<td>.45**</td>
<td></td>
<td>.88**</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(sig.)</td>
<td>.000</td>
<td>.000</td>
<td>.365</td>
<td>.508</td>
<td>.000</td>
<td>.000</td>
<td></td>
<td>.000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dual Distance</td>
<td>.56**</td>
<td>.55**</td>
<td>.14</td>
<td>.12</td>
<td>.60**</td>
<td>.57**</td>
<td></td>
<td>.16</td>
<td>.20</td>
<td></td>
</tr>
<tr>
<td>(sig.)</td>
<td>.000</td>
<td>.000</td>
<td>.147</td>
<td>.205</td>
<td>.000</td>
<td>.000</td>
<td></td>
<td>.091</td>
<td>.039</td>
<td></td>
</tr>
<tr>
<td>Distance</td>
<td>.67**</td>
<td>.68**</td>
<td>.07</td>
<td>-.05</td>
<td>.71**</td>
<td>.77**</td>
<td></td>
<td>.11</td>
<td>.20*</td>
<td>.77**</td>
</tr>
<tr>
<td>(sig.)</td>
<td>.000</td>
<td>.000</td>
<td>.492</td>
<td>.637</td>
<td>.000</td>
<td>.000</td>
<td></td>
<td>.232</td>
<td>.038</td>
<td>.000</td>
</tr>
</tbody>
</table>

Table 9.5 Correlation of the studied Metrics For Tomcat 7.0.6
The correlation matrices for maintainability are presented in Table 9.6. The results show a strong correlation between maintainability and some of the metrics. As mentioned before in this section, the t statistic is calculated to provide correlations significance p. Correlations at the 0.05 level are marked with (*), and the ones significant at 0.01 are marked with (**). A detailed discussion of the correlation between the newly developed metrics and maintainability metrics is given below. In addition, there is a high correlation amongst maintainability metrics, REVISION and RLOC, as shown in Table 9.7.
Table 9.6 Spearman’s Rho Correlations for Maintainability Analysis

<table>
<thead>
<tr>
<th>Metrics</th>
<th>Camel 2.0.0</th>
<th>JEdit 4.5.0</th>
<th>JHotDraw 7.5.1</th>
<th>Tomcat 7.0.6</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Revisions</td>
<td>RLOC</td>
<td>Revisions</td>
<td>RLOC</td>
</tr>
<tr>
<td>InRank</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sig.</td>
<td>.000</td>
<td>.000</td>
<td>.000</td>
<td>.074</td>
</tr>
<tr>
<td>Ca</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sig.</td>
<td>.000</td>
<td>.000</td>
<td>.000</td>
<td>.019</td>
</tr>
<tr>
<td>OutRank</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sig.</td>
<td>.000</td>
<td>.000</td>
<td>.000</td>
<td>.035</td>
</tr>
<tr>
<td>Ce</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sig.</td>
<td>.000</td>
<td>.000</td>
<td>.229</td>
<td>.701</td>
</tr>
<tr>
<td>Dual Instability</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sig.</td>
<td>.000</td>
<td>.000</td>
<td>.909</td>
<td>.339</td>
</tr>
<tr>
<td>Instability</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sig.</td>
<td>.000</td>
<td>.000</td>
<td>.900</td>
<td>.339</td>
</tr>
<tr>
<td>Dual Abstractness</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sig.</td>
<td>.000</td>
<td>.000</td>
<td>.014</td>
<td>.150</td>
</tr>
<tr>
<td>Abstractness</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sig.</td>
<td>.000</td>
<td>.000</td>
<td>.171</td>
<td>.226</td>
</tr>
<tr>
<td>Dual Distance</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sig.</td>
<td>.000</td>
<td>.000</td>
<td>.170</td>
<td>.304</td>
</tr>
<tr>
<td>Distance</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sig.</td>
<td>.000</td>
<td>.000</td>
<td>.088</td>
<td>.750</td>
</tr>
</tbody>
</table>
Table 9.7 Correlations of Maintainability Metrics

<table>
<thead>
<tr>
<th>Metrics</th>
<th>Camel 2.0.0</th>
<th>JEdit 4.5.0</th>
<th>JHotDraw 7.5.1</th>
<th>Tomcat 7.0.6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Revisions</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Revisions</td>
<td>1</td>
<td>.96**</td>
<td>1 .98**</td>
<td>1 .79**</td>
</tr>
<tr>
<td>RLOC</td>
<td>.96**</td>
<td>1 .98**</td>
<td>1 .79**</td>
<td>1 .97**</td>
</tr>
</tbody>
</table>

Table 9.6 reveals that the newly developed measures of package coupling (InRank, OutRank) consistently have a positive moderate to a high correlation with the two measures of package maintainability, number of package revisions (Revisions), and the number of revised lines of code (RLOC), across the data sets. The correlation values between the new measure of package afferent coupling (InRank) and number of revisions (Revisions) across the four data sets range from 0.22 with no significance (for the JHotDraw system data set) to 0.56 (for the JEdit system data set). Similarly, correlation values between the new measure of package afferent coupling (InRank) and the number of revised lines of code (RLOC) across the four data sets range from 0.29 (for the Tomcat system data set) to 0.58 (for the JEdit system data set). Likewise, the correlation values between the new measure of package efferent coupling (OutRank) and number of revisions (Revisions) across the four data sets range from 0.26 (for the JHotDraw system data set) to 0.69 (for the JEdit system data set). Similarly, correlation values between the new measure of package efferent coupling (OutRank) and the number of revised lines of code (RLOC) across the four data sets range from 0.44 (for the JHotDraw system data set) to 0.58 (for the JEdit system data set).
set) to 0.69 (for the JEdit system data set). The statistically significant correlations confirm the expectation of a low coupling software package being easy to maintain.

The high correlation values between the new measure of package efferent coupling (OutRank) and the maintenance effort indicate that the packages having a high efferent coupling (OutRank) are likely to also have high maintenance cost in the future and vice versa. The same can be said with less extent to the afferent coupling (InRank) and abstractness. The correlations with the coupling and abstractness of the new metrics are positive and significant; the more coupled and abstract a package, the higher maintenance cost it will have.

Therefore, the hypothesis that the packages with more InRank value have less maintenance effort is not supported in almost all four systems. The only exception is in JHotDraw with REVISION where there is correlation of (0.22) with no significance (0.074) which is higher than 0.05 by a non-significant amount. This evidence suggests that we must consider the alternative hypotheses H1 (InRank, REVISION) and H1 (InRank, RLOC). Likewise, the hypothesis that the packages with more OutRank value will have less maintenance effort is not supported in all four systems. This evidence suggests that we must consider the alternative hypotheses H1(OutRank, REVISION) and H1(OutRank, RLOC). For dual abstractness, the hypothesis that the packages with more Dual Abstractness value have less maintenance effort is not supported in almost all four systems. The only exception is in JHotDraw with REVISION where there is a correlation of (0.18) with no significance (0.150) which is higher than (0.05) by a non-significant amount. This evidence suggests that we must consider the alternative hypotheses H1(Dual abstractness, REVISION) and H1(Dual abstractness, RLOC).
For both Camel and Tomcat, we have some evidence that Dual distance led to an increase in maintenance effort (RLOC and Revision). On Camel only, we have some evidence that Dual instability led to an increase in maintenance effort (RLOC and Revision). There is no evidence found on the other systems for both dual instability and dual distance. Hence, the results are inconclusive for Dual instability and Dual distance. As a result, we can’t reject the following null hypotheses: H0 (Dual instability, REVISION), H0 (Dual instability, RLOC), H0 (Dual distance, REVISION) and H0 (Dual distance, RLOC). Table 9.8 provides a summary of the assessments of all maintainability propositions.
<table>
<thead>
<tr>
<th>Proposition</th>
<th>Proposition description</th>
<th>suggestion</th>
</tr>
</thead>
<tbody>
<tr>
<td>$H_0$(InRank, REVISION)</td>
<td>Coupling of package (InRank) does not contribute to its maintenance cost (REVISION) and require no more maintenance cost than its peer.</td>
<td>We have evidence to support rejecting it.</td>
</tr>
<tr>
<td>$H_0$(InRank, RLOC)</td>
<td>Coupling of package (InRank) does not contribute to its maintenance cost (RLOC) and require no more maintenance cost than its peer.</td>
<td>We have evidence to support rejecting it.</td>
</tr>
<tr>
<td>$H_0$(OutRank, REVISION)</td>
<td>Coupling of package (OutRank) does not contribute to its maintenance cost (REVISION) and require no more maintenance cost than its peer.</td>
<td>We have strong evidence to support rejecting it.</td>
</tr>
<tr>
<td>$H_0$(OutRank, RLOC)</td>
<td>Coupling of package (OutRank) does not contribute to its maintenance cost (RLOC) and require no more maintenance cost than its peer.</td>
<td>We have strong evidence to support rejecting it.</td>
</tr>
<tr>
<td>$H_0$(Dual_Instability, REVISION)</td>
<td>Instability of package does not contribute to its maintenance cost (REVISION) and require no more maintenance cost than its peer.</td>
<td>Evidence does not support rejecting it.</td>
</tr>
<tr>
<td>$H_0$(Dual_Instability, RLOC)</td>
<td>Instability of package does not contribute to its maintenance cost (RLOC) and require no more maintenance cost than its peer.</td>
<td>Evidence does not support rejecting it.</td>
</tr>
<tr>
<td>$H_0$(Dual_Abstractness, REVISION)</td>
<td>Abstractness of package does not contribute to its maintenance cost (REVISION) and require no more maintenance cost than its peer.</td>
<td>We have evidence to support rejecting it.</td>
</tr>
<tr>
<td>$H_0$(Dual_Abstractness, RLOC)</td>
<td>Abstractness of package (OutRank) does not contribute to its maintenance cost (RLOC) and require no more maintenance cost than its peer.</td>
<td>We have evidence to support rejecting it.</td>
</tr>
<tr>
<td>$H_0$(Dual_Distance, REVISION)</td>
<td>Distance of package from the main sequence does not contribute to its maintenance cost (REVISION) and require no more maintenance cost than its peer.</td>
<td>Evidence does not support rejecting it.</td>
</tr>
<tr>
<td>$H_0$(Dual_Distance, RLOC)</td>
<td>Distance of package from the main sequence does not contribute to its maintenance cost (RLOC) and require no more maintenance cost than its peer.</td>
<td>Evidence does not support rejecting it.</td>
</tr>
</tbody>
</table>
9.2.3 Regression Analysis

The coefficient of correlation that was obtained from the previous section cannot provide the prediction of the maintenance and test effort. However, correlations are useful since they can indicate a predictive relationship that can be exploited. Therefore, we’ve applied linear regression analysis to determine the ability of the newly developed metrics that correlate with maintainability to predict package maintainability. We performed multivariate regression analysis to find out the relationship between the newly developed package metrics and revision count. Building the maintainability prediction models are the outcomes of this analysis. We used linear regression analysis to assess the validity of the new metrics as predictors of package maintainability. As before, correlations significant at the 0.05 level are marked with (*), and the ones significant at 0.01 are marked with (**).

We’ve applied a multivariate linear regression technique. Multivariate regression analyses establish the relationship between dependent variables (RLOC, REVISION, and TLOC) and explanatory variables (the new metrics). Data from the four systems are integrated to form one data set for the regression analysis.

We know from the previous section that the newly developed metrics are not totally independent and, hence, must capture redundant information. Thus, not all of them are required in multivariate analyses. However, a standard procedure called stepwise selection was also used to select the necessary variables for multivariate analysis. We constructed our multivariate models in a stepwise approach, starting from all of the independent variables that were found to
be statistically significant and decreasing their number based on their statistical significance in the multivariate model.

Table 9.9 presents the means and standard deviations for the variables on the combined data used in a maintainability regression analysis.

**Table 9.9  Means and Standard Deviations of the Data Used in Maintainability Regression Analysis**

<table>
<thead>
<tr>
<th>Variable</th>
<th>Combined Data N=475</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mean</td>
</tr>
<tr>
<td>InRank</td>
<td>0.034</td>
</tr>
<tr>
<td>OutRank</td>
<td>0.087</td>
</tr>
<tr>
<td>Dual Abstractness</td>
<td>0.201</td>
</tr>
<tr>
<td>Classes</td>
<td>16.23</td>
</tr>
<tr>
<td>LogClasses</td>
<td>2.171</td>
</tr>
<tr>
<td>Revisions</td>
<td>6.128</td>
</tr>
<tr>
<td>LogRevision</td>
<td>1.167</td>
</tr>
<tr>
<td>RLOC</td>
<td>240.2</td>
</tr>
<tr>
<td>LogRLOC</td>
<td>2.99</td>
</tr>
</tbody>
</table>

For linear regression, we want to make sure that the relationship between a dependent variable (RLOC and REVISION) and explanatory variables (the new metrics) is approximately
linear. Also, it is required that the data follows a normal data distribution to make sure that the computed statistics are not misleading. However, this is not the case in many variables in our study. There are a number of transformations that can be used to approximate data to make them normally distributed such as logarithmic, square roots, and inverse transforms. Therefore, data screening and evaluation of the linearity condition for the regression analyses led to the natural log transformation of the variables number of revisions (REVISION), package size (classes), and the number of revised lines of code on the package (RLOC). Table 9.10 provides Pearson correlations among the variables used in the maintainability regression analyses for the combined data.

**Table 9.10 Pearson Correlations among the Variables Used In the Maintainability Regression Analysis**

<table>
<thead>
<tr>
<th></th>
<th>OutRank</th>
<th>Dual Abstractness</th>
<th>LogClasses</th>
<th>logRevision</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Log Classes</strong></td>
<td>0.543**</td>
<td>0.426**</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>LogRevision</strong></td>
<td>0.483**</td>
<td>0.322**</td>
<td>0.638**</td>
<td></td>
</tr>
<tr>
<td><strong>LogRLOC</strong></td>
<td>0.439**</td>
<td>0.328**</td>
<td>0.615**</td>
<td>0.954**</td>
</tr>
</tbody>
</table>

Two regression analyses were run to predict package maintainability by the newly developed measures (OutRank, Dual Abstractness) controlling for the natural log transformed package size (LogClasses). The first regression analysis is for predicting the natural log of the number of revisions (logRevision) as the first measure for package maintainability. The second regression analysis is for predicting the natural log of the number of revised lines of code.
(logRLOC) as the second measure for package maintainability. Table 9.11 shows the results of a multivariate linear regression analysis for maintenance cost measured by the log of the number of package revisions (logRevision), and Table 9.12 shows the results for a multivariate linear regression analysis for maintenance cost measured by the log of the number of revised lines of code (logRLOC).

Regression results for the number of revisions (logRevision) indicate that the overall model of the new measures (OutRank, Dual Abstractness) and package size (LogClasses) significantly predict the natural log of the number of package revisions \([R^2=0.446, \text{R}^2_{\text{adj}} = 0.442, F(3, 472) =126.642, p = 0.000]\). R-square (\(R^2\)) and Adjusted R-square (\(R^2_{\text{adj}}\)) are regression quality indicators that measures the quality of predictions; and are defined as the percentage of the variance in the dependent variable (logRevision and logRLOC) accounted for by the independent variables in a regression model based on the sample data and in the population, respectively. So, the prediction model accounts for 44.6% of the variance in the log of the number of package revisions. Table 9.12 presents a summary of the regression model coefficients.

Table 9.11 Summary of the Model Predicting The Log Of Package Revision (LogRevision)

<table>
<thead>
<tr>
<th>Predictor</th>
<th>B</th>
<th>Beta</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>OutRank</td>
<td>0.520</td>
<td>0.598</td>
<td>2.017**</td>
</tr>
<tr>
<td>Dual Abstractness</td>
<td>0.250</td>
<td>0.076</td>
<td>2.139**</td>
</tr>
<tr>
<td>LogClasses</td>
<td>0.586</td>
<td>0.081</td>
<td>14.342**</td>
</tr>
</tbody>
</table>
Similarly, regression results for the log of the number of revised lines of code (LogLOC) indicate that the overall model of the new measures of package (OutRank, Dual Abstractness) and package size (LogClasses) significantly predict the natural log of the number of package revised lines of Code \( [R^2=0.368, \text{R}^2_{\text{adj}} = 0.364, F(3, 472) = 91.764, p = 0.000] \). The prediction model accounts for about 37% of the variance in the log of the number of revised lines of code. Table 9.12 presents a summary of the regression model coefficients.

**Table 9.12 Summary of the Model Predicting The Log Of The Number of Revised LOC (LogRLOC)**

<table>
<thead>
<tr>
<th>Predictor</th>
<th>B</th>
<th>Beta</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>OutRank</td>
<td>0.842</td>
<td>0.056</td>
<td>1.303**</td>
</tr>
<tr>
<td>Dual Abstractness</td>
<td>0.662</td>
<td>0.086</td>
<td>2.27**</td>
</tr>
<tr>
<td>LogClasses</td>
<td>1.259</td>
<td>0.547</td>
<td>12.30**</td>
</tr>
</tbody>
</table>

Our results demonstrate that packages with lower newly developed coupling and abstractness metrics values have better maintainability than those with higher newly developed coupling and abstractness metrics values. It appears that the newly developed efferent coupling metric (OutRank) and the new Abstractness are good predictors of our maintainability indicators.
9.3 **Testability**

The goals of the testability studies are to identify any relationship between the newly developed metrics and test effort which are measured by the number of lines of code of test package (TLOC) and to find whether there is a compact model containing some of these metrics that can predict the system’s testability.

Data screening and evaluation of linearity for assessing correlations and regression analysis led to the natural log transformation of the following variables: number of lines of code of test package (TLOC), and package size (number of classes in a package). Martin’s package metric suite is included in the list of variables for the purpose of comparison.

### 9.3.1 Descriptive statistics

Table 9.13 provides descriptive statistics (mean and standard deviation) for the variables used in analyzing package testability across the four systems: Camel 2.0.0, Tomcat 7.0.6, Hadoop 2.2.0, Synapse 2.1.0, and Ant 1.92.
Table 9.13 Means and Standard Deviation of the Variables Used in Testability Analyses

<table>
<thead>
<tr>
<th>Variable</th>
<th>Camel 2.0.0 N=264</th>
<th>Tomcat 7.0.6 N=113</th>
<th>Hadoop 2.2.0 N=221</th>
<th>Synapse 2.1.0 N=117</th>
<th>Ant 1.92 N=67</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mean</td>
<td>S.D.</td>
<td>Mean</td>
<td>S.D.</td>
<td>Mean</td>
</tr>
<tr>
<td>InRank</td>
<td>.01</td>
<td>.07</td>
<td>.03</td>
<td>.10</td>
<td>.012</td>
</tr>
<tr>
<td>Ca</td>
<td>17.6</td>
<td>127</td>
<td>15.7</td>
<td>33.9</td>
<td>17.2</td>
</tr>
<tr>
<td>OutRank</td>
<td>.06</td>
<td>.1</td>
<td>.07</td>
<td>.17</td>
<td>.052</td>
</tr>
<tr>
<td>Ce</td>
<td>9.9</td>
<td>13.9</td>
<td>16.5</td>
<td>25.0</td>
<td>13.4</td>
</tr>
<tr>
<td>Dual Instability</td>
<td>.93</td>
<td>.19</td>
<td>.61</td>
<td>.42</td>
<td>.870</td>
</tr>
<tr>
<td>Instability</td>
<td>.86</td>
<td>.27</td>
<td>.64</td>
<td>.38</td>
<td>.504</td>
</tr>
<tr>
<td>Dual Abstr.</td>
<td>.17</td>
<td>.32</td>
<td>.21</td>
<td>.33</td>
<td>.682</td>
</tr>
<tr>
<td>Abstractness</td>
<td>.08</td>
<td>.18</td>
<td>.13</td>
<td>.24</td>
<td>.278</td>
</tr>
<tr>
<td>Dual Distance</td>
<td>.17</td>
<td>.29</td>
<td>.34</td>
<td>.36</td>
<td>.154</td>
</tr>
<tr>
<td>Distance</td>
<td>.13</td>
<td>.21</td>
<td>.27</td>
<td>.31</td>
<td>.264</td>
</tr>
<tr>
<td>TLOC</td>
<td>119.3</td>
<td>689</td>
<td>76.6</td>
<td>207</td>
<td>.384</td>
</tr>
<tr>
<td>LogTOLC</td>
<td>.91</td>
<td>2.14</td>
<td>1.28</td>
<td>2.30</td>
<td>.267</td>
</tr>
</tbody>
</table>
9.3.2 Statistical Analysis

The main hypotheses are tested in this section. The first set of hypothesis states that there is a relationship between the newly developed metrics and the test effort which is measured by the number of lines of code of test package (TLOC). In order to evaluate the above hypotheses, we applied correlation analysis to compute the coefficient of correlation (r) between the two metrics suites, the new metrics and Martin metrics, and the test effort per package. This is a commonly used statistical measure to determine the dependency of a dependent variable on explanatory variables. As mentioned earlier, we decided to use Spearman’s rank-order correlation coefficient (Spearman Rho) over Pearson’s correlation coefficient due to the fact that the dependent variable, the number of lines of code of test package (TLOC), is not normally distributed, and the nonparametric nature of the studied package metrics in this study. Spearman’s rank-order correlation coefficient is the proper measure of a bivariate relationship when normality and linearity conditions for the Pearson’s product moment correlation do not hold.

For this study, the Spearman Rho correlation provides a measure of association between the new package metrics, Martin’s package metric suite, package size (Classes), and the measures of package testability, the number of lines of code of test package (TLOC), within each of the four data sets. The value of rs ranges from -1 to 1 where the value of r, +1, indicates a perfect positive correlation while the value of r, -1, indicates a perfect negative correlation. A value of 0 indicates that there is no correlation; that is the two variants are independent of each other. Tables 9.14- 9.18 provide the list of these correlations for the five sets of data. In addition, we estimate the statistical significance of rs by calculating the t statistic. This test provides a
significance p. The significance p indicates the probability that the observed value is a chance event based on the number of pairs in the data set. The significance p allows us to reject the null hypothesis (H₀(m;n)) and accept the alternative hypothesis (H₁(m;n)) with a certain level of confidence.

Tables 9.14-9.18 present the correlations of the newly developed metrics for the five systems. We calculate the value of rs from the arithmetic average of all the correlation matrixes. This enabled us to specify the followings observations:

- There is no strong correlation between InRank and any other metric. In some of the testability study’s systems, there are positive medium correlations between InRank and some of the metrics.

- There is no strong correlation between OutRank and any other metric. This means that there is no linear association between OutRank and the other metrics.

- In four systems out of five systems, there is a medium correlation among dual abstractness and dual instability.

From these observations, it follows that these metrics, as expected from their definitions, are not totally independent and may represent redundant information. However, there are no strong correlations between the newly developed metrics across the studied systems and hence they are not highly correlated which indicate that there is no multicollinearity. Multicollinearity will be checked in the next section.
## Table 9.14 Correlation of the studied Metrics For Camel 2.0.0

<table>
<thead>
<tr>
<th>Metrics</th>
<th>InRank</th>
<th>Ca</th>
<th>OutRank</th>
<th>Ce</th>
<th>Dual Instability</th>
<th>Instability</th>
<th>Dual Abstractness</th>
<th>Abstractness</th>
<th>Dual Distance</th>
<th>Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>InRank (sig.)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ca (sig.)</td>
<td>.99**</td>
<td>.000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OutRank (sig.)</td>
<td>.23**</td>
<td>.22**</td>
<td>.000</td>
<td></td>
<td></td>
<td>.000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ce (sig.)</td>
<td>.27**</td>
<td>.26**</td>
<td>.95**</td>
<td></td>
<td></td>
<td>.95**</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dual Instability (sig.)</td>
<td>.98**</td>
<td>.98**</td>
<td>-.12</td>
<td>.17**</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instability (sig.)</td>
<td>.97**</td>
<td>.98**</td>
<td>-.13*</td>
<td>-.13*</td>
<td>.16**</td>
<td>.99**</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dual Abstractness (sig.)</td>
<td>.46**</td>
<td>.47**</td>
<td>.20**</td>
<td>.23**</td>
<td>.23**</td>
<td>.44**</td>
<td>.45**</td>
<td>.84**</td>
<td>.72**</td>
<td></td>
</tr>
<tr>
<td>Abstractness (sig.)</td>
<td>.46**</td>
<td>.46**</td>
<td>.20**</td>
<td>.23**</td>
<td>.44**</td>
<td>.45**</td>
<td></td>
<td>.84**</td>
<td>.72**</td>
<td></td>
</tr>
<tr>
<td>Dual Distance (sig.)</td>
<td>.65**</td>
<td>.66**</td>
<td>.15**</td>
<td>.19**</td>
<td>.67**</td>
<td>.71**</td>
<td>.84**</td>
<td>.72**</td>
<td>.70**</td>
<td>.83**</td>
</tr>
<tr>
<td>Distance (sig.)</td>
<td>.72**</td>
<td>.73**</td>
<td>.13**</td>
<td>.16**</td>
<td>.74**</td>
<td>.75**</td>
<td>.60**</td>
<td>.70**</td>
<td>.70**</td>
<td>.83**</td>
</tr>
</tbody>
</table>
# Table 9.15 Correlation of the studied Metrics For Tomcat 7.0.6

<table>
<thead>
<tr>
<th>Metrics</th>
<th>InRank</th>
<th>Ca</th>
<th>OutRank</th>
<th>Ce</th>
<th>Dual Instability</th>
<th>Instability</th>
<th>Dual Abstractness</th>
<th>Abstractness</th>
<th>Dual Distance</th>
<th>Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>InRank</strong> (sig.)</td>
<td><strong>.94</strong></td>
<td><strong>.94</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Ca</strong> (sig.)</td>
<td>.18</td>
<td>.27</td>
<td></td>
<td>.14</td>
<td></td>
<td>.18</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>OutRank</strong> (sig.)</td>
<td>.61</td>
<td>.03</td>
<td></td>
<td>.02</td>
<td></td>
<td>.061</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Ce</strong> (sig.)</td>
<td>.15</td>
<td>.22</td>
<td>.87</td>
<td>.22</td>
<td>.24</td>
<td>.87</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Dual Instability</strong> (sig.)</td>
<td>.00</td>
<td>.02</td>
<td>.026</td>
<td>.00</td>
<td>.010</td>
<td>.00</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Instability</strong> (sig.)</td>
<td>.82</td>
<td>.82</td>
<td>.11</td>
<td>.22</td>
<td>.94</td>
<td>.94</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Dual Abstractness</strong> (sig.)</td>
<td>.00</td>
<td>.00</td>
<td>.247</td>
<td>.018</td>
<td>.00</td>
<td>.00</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Abstractness</strong> (sig.)</td>
<td>.47</td>
<td>.45</td>
<td>.09</td>
<td>.06</td>
<td></td>
<td>.06</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Dual Distance</strong> (sig.)</td>
<td>.56</td>
<td>.55</td>
<td>.14</td>
<td>.12</td>
<td></td>
<td>.12</td>
<td></td>
<td></td>
<td></td>
<td>.16</td>
</tr>
<tr>
<td><strong>Distance</strong> (sig.)</td>
<td>.67</td>
<td>.68</td>
<td>.07</td>
<td>.05</td>
<td></td>
<td>.05</td>
<td></td>
<td></td>
<td>.11</td>
<td></td>
</tr>
</tbody>
</table>
Table 9.16 Correlation of the studied Metrics For Hadoop 2.2.0

<table>
<thead>
<tr>
<th>Metrics</th>
<th>InRank</th>
<th>Ca</th>
<th>OutRank</th>
<th>Ce</th>
<th>Dual Instability</th>
<th>Instability</th>
<th>Dual Abstractness</th>
<th>Abstractness</th>
<th>Dual Distance</th>
<th>Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>InRank</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(sig.)</td>
<td>.23**</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ca</td>
<td></td>
<td>.21**</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(sig.)</td>
<td>0.001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OutRank</td>
<td>-0.03</td>
<td></td>
<td>.21**</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(sig.)</td>
<td>0.612</td>
<td></td>
<td>0.002</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ce</td>
<td>-0.04</td>
<td>.23**</td>
<td>.88**</td>
<td></td>
<td></td>
<td>.199**</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(sig.)</td>
<td>0.552</td>
<td></td>
<td>0.001</td>
<td>0</td>
<td></td>
<td></td>
<td>.003</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dual Instability</td>
<td>-.45**</td>
<td>-.34**</td>
<td>.17*</td>
<td>.199*</td>
<td>.003</td>
<td>.003</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(sig.)</td>
<td>0</td>
<td>0</td>
<td>0.01</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instability</td>
<td>-.26**</td>
<td>-.45**</td>
<td>0.026</td>
<td>0.029</td>
<td>.70**</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(sig.)</td>
<td>0</td>
<td>0</td>
<td>0.695</td>
<td>0.668</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dual Abstractness</td>
<td>0.02</td>
<td>0.027</td>
<td>0.024</td>
<td>0.003</td>
<td>-.19**</td>
<td>-.29**</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(sig.)</td>
<td>0.773</td>
<td>0.685</td>
<td>0.724</td>
<td>0.963</td>
<td>0.005</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Abstractness</td>
<td>0.012</td>
<td>0.052</td>
<td>-0.09</td>
<td>-.13*</td>
<td>-.30**</td>
<td>-.28**</td>
<td>-.67**</td>
<td>.005</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(sig.)</td>
<td>0.856</td>
<td>0.443</td>
<td>0.176</td>
<td>0.048</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dual Distance</td>
<td>.153*</td>
<td>.077</td>
<td>.016</td>
<td>-.02</td>
<td>.24**</td>
<td>-.22**</td>
<td>.76**</td>
<td>.40**</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(sig.)</td>
<td>0.023</td>
<td>0.252</td>
<td>0.815</td>
<td>0.747</td>
<td>0</td>
<td>0.001</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Distance</td>
<td>.27**</td>
<td>.42**</td>
<td>-.03</td>
<td>-.03</td>
<td>-.5**</td>
<td>-.79**</td>
<td>0.09</td>
<td>0.117</td>
<td>.22**</td>
<td>.001</td>
</tr>
<tr>
<td>(sig.)</td>
<td>0</td>
<td>0</td>
<td>0.671</td>
<td>0.666</td>
<td>0</td>
<td>0</td>
<td>0.192</td>
<td>0.083</td>
<td>.001</td>
<td></td>
</tr>
</tbody>
</table>
Table 9.17 Correlation of the studied Metrics for Synapse 2.1.0

<table>
<thead>
<tr>
<th>Metrics</th>
<th>InRank</th>
<th>Ca</th>
<th>OutRank</th>
<th>Ce</th>
<th>Dual Instability</th>
<th>Instability</th>
<th>Dual Abstractness</th>
<th>Abstractness</th>
<th>Dual Distance</th>
<th>Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>InRank</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(sig.)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ca</td>
<td>0.101</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(sig.)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.28</td>
</tr>
<tr>
<td>OutRank</td>
<td>-0.07</td>
<td>.24**</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(sig.)</td>
<td>0.448</td>
<td>0.008</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ce</td>
<td>-0.06</td>
<td>.40**</td>
<td>.91**</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(sig.)</td>
<td>0.529</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dual Instability</td>
<td>-0.42**</td>
<td>-0.1</td>
<td>.21*</td>
<td>.21*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(sig.)</td>
<td>0</td>
<td>0.311</td>
<td>0.026</td>
<td>0.022</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instability</td>
<td>-.19*</td>
<td>-.61**</td>
<td>0.02</td>
<td>-.09</td>
<td>.55**</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(sig.)</td>
<td>0.043</td>
<td>0</td>
<td>0.836</td>
<td>0.347</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dual Abstractness</td>
<td>0.153</td>
<td>.25**</td>
<td>0.07</td>
<td>0.13</td>
<td>-.23*</td>
<td>-.40**</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(sig.)</td>
<td>0.099</td>
<td>0.008</td>
<td>0.48</td>
<td>0.158</td>
<td>0.014</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Abstractness</td>
<td>-.02</td>
<td>.26**</td>
<td>-0.1</td>
<td>-0.04</td>
<td>-.23*</td>
<td>-.41**</td>
<td>.6**</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(sig.)</td>
<td>0.848</td>
<td>0.006</td>
<td>0.281</td>
<td>0.687</td>
<td>0.013</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dual Distance</td>
<td>0.049</td>
<td>.33**</td>
<td>0.06</td>
<td>0.12</td>
<td>-.24*</td>
<td>-.47**</td>
<td>.75**</td>
<td>.38**</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(sig.)</td>
<td>0.601</td>
<td>0</td>
<td>0.522</td>
<td>0.188</td>
<td>0.011</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Distance</td>
<td>.20*</td>
<td>.45**</td>
<td>-0.08</td>
<td>0.01</td>
<td>-.42**</td>
<td>-.74**</td>
<td>0.16</td>
<td>.38**</td>
<td>.40**</td>
<td></td>
</tr>
<tr>
<td>(sig.)</td>
<td>0.027</td>
<td>0</td>
<td>0.417</td>
<td>0.949</td>
<td>0</td>
<td>0</td>
<td>0.081</td>
<td>0</td>
<td>.000</td>
<td></td>
</tr>
<tr>
<td>Metrics</td>
<td>InRank</td>
<td>Ca</td>
<td>OutRank</td>
<td>Ce</td>
<td>Dual Instability</td>
<td>Instability</td>
<td>Dual Abstractness</td>
<td>Abstractness</td>
<td>Dual Distance</td>
<td>Distance</td>
</tr>
<tr>
<td>-----------------</td>
<td>--------</td>
<td>-----</td>
<td>---------</td>
<td>-----</td>
<td>------------------</td>
<td>-------------</td>
<td>-------------------</td>
<td>--------------</td>
<td>---------------</td>
<td>----------</td>
</tr>
<tr>
<td><strong>InRank</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(sig.)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Ca</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(sig.)</td>
<td>1.0**</td>
<td>-0.1</td>
<td>-0.1</td>
<td>0.1</td>
<td></td>
<td>0.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>OutRank</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(sig.)</td>
<td>0.371</td>
<td>0.371</td>
<td>0.371</td>
<td>0.371</td>
<td></td>
<td>0.371</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Ce</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(sig.)</td>
<td>-0.1</td>
<td>-0.1</td>
<td>1.0**</td>
<td>0.1</td>
<td></td>
<td>0.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Dual Instability</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(sig.)</td>
<td>-1.0**</td>
<td>-1.0**</td>
<td>0.1</td>
<td>0.1</td>
<td></td>
<td>0.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Instability</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(sig.)</td>
<td>-1.0**</td>
<td>-1.0**</td>
<td>0.1</td>
<td>0.1</td>
<td></td>
<td>1.0**</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Dual Abstractness</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(sig.)</td>
<td>-0.07</td>
<td>-0.07</td>
<td>0.1</td>
<td>0.1</td>
<td>0.07</td>
<td>0.07</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Abstractness</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(sig.)</td>
<td>0.14</td>
<td>0.14</td>
<td>0.1</td>
<td>0.1</td>
<td>-0.14</td>
<td>-0.14</td>
<td></td>
<td></td>
<td></td>
<td>.62**</td>
</tr>
<tr>
<td><strong>Dual Distance</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(sig.)</td>
<td>0.258</td>
<td>0.258</td>
<td>0.493</td>
<td>0.493</td>
<td>0.258</td>
<td>0.258</td>
<td></td>
<td></td>
<td></td>
<td>.99**</td>
</tr>
<tr>
<td><strong>Distance</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>.64**</td>
</tr>
<tr>
<td>(sig.)</td>
<td>0.004</td>
<td>0.004</td>
<td>0.399</td>
<td>0.399</td>
<td>0.004</td>
<td>0.004</td>
<td></td>
<td></td>
<td></td>
<td>.98**</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>.63**</td>
</tr>
</tbody>
</table>

Table 9.18 Correlation of the studied Metrics for Ant 1.92
The correlation matrices for testability are presented in Table 9.19. The results show a strong correlation between testability and some of the metrics. As mentioned before in this section, the t statistic is calculated to provide correlations significance p. Correlations at the 0.05 level are marked with (*), and the ones significant at 0.01 are marked with (**). A detailed discussion of the correlation between the newly developed metrics and testability metrics is given below.
Table 9.19  Spearman’s Rho Correlations for Maintainability Analysis

<table>
<thead>
<tr>
<th>Metrics</th>
<th>Camel 2.0.0 N=264</th>
<th>Tomcat 7.0.6 N=113</th>
<th>Hadoop 2.2.0 N=221</th>
<th>Synapse 2.1.0 N=117</th>
<th>Ant 1.92 N=67</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>TLOC</td>
<td>TLOC</td>
<td>TLOC</td>
<td>TLOC</td>
<td>TLOC</td>
</tr>
<tr>
<td>InRank</td>
<td>rs</td>
<td>Sig.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>.170**</td>
<td>0.01</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ca</td>
<td>rs</td>
<td>Sig.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>.193**</td>
<td>0.00</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OutRank</td>
<td>rs</td>
<td>Sig.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>.301**</td>
<td>0.00</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ce</td>
<td>rs</td>
<td>Sig.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>.436**</td>
<td>0.00</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dual Instability</td>
<td>rs</td>
<td>Sig.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>-.07</td>
<td>0.24</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Instability</td>
<td>rs</td>
<td>Sig.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>-.229**</td>
<td>0.00</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dual Abstr.</td>
<td>rs</td>
<td>Sig.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0.04</td>
<td>0.49</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Abstractness</td>
<td>rs</td>
<td>Sig.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>-.03</td>
<td>0.66</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dual Distance</td>
<td>rs</td>
<td>Sig.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0.00</td>
<td>1.00</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Distance</td>
<td>rs</td>
<td>Sig.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>.263**</td>
<td>0.00</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 9.19 also reveals that the new measure of package efferent coupling (OutRank) consistently has a positive moderate to a high correlation with the measure of package testability (TLOC) across all the five data sets. The correlation values between the new measure of package efferent coupling (OutRank) and the number of lines of code of test package (TLOC) across the five data sets range from 0.301 (for the Camel system data set) to 0.739 (for the Ant system data set). The high correlation values (0.301 for Camel, 0.502 for Tomcat, 0.725 for Hadoop, 0.506 for Synapse and 0.739 for Ant) between the new measure of package efferent coupling and the test effort indicate that the packages having high efferent coupling are likely to also have high test cost in the future and vice versa. The statistically significant correlations confirm the expectation of low efferent coupling software package requiring less effort to test.

Therefore, the hypothesis that the packages with more OutRank value have less test effort is not supported in all five systems. This evidence suggests that we must consider the alternative hypotheses H1(OutRank, TLOC). For the new measure of package efferent coupling (InRank), there is only weak evidence in Camel and Ant that InRank indicates that there is an increase in the test effort (TLOC). There is no evidence found on the other systems for InRank. Therefore, results are inconclusive for InRank. As a result, we can’t reject the following null hypotheses: H0 (InRank, TLOC). On Tomcat only, there is evidence that Dual instability reveals that there is an increase in the test effort (TLOC). However, On Ant, we have some evidence that Dual instability indicates that there is a decrease in test effort (TLOC). There is no evidence found on the other systems for both dual abstractness and dual distance. Hence, results are inconclusive for Dual instability, Dual abstractness, and Dual distance. As a result, we can’t reject the following null hypotheses: H0 (Dual instability, TLOC), H0 (Dual abstractness, TLOC), and H0 (Dual
distance, TLOC). Table 9.20 provides a summary of the assessments of all testability propositions.

**Table 9.20 Testability propositions summary**

<table>
<thead>
<tr>
<th>Proposition</th>
<th>Proposition description</th>
<th>suggestion</th>
</tr>
</thead>
<tbody>
<tr>
<td>$H_0(\text{InRank, TLOC})$</td>
<td>Coupling of package (InRank) does not contribute to its test cost (TLOC) and require no more test cost than its peer.</td>
<td>Evidence does not support rejecting it.</td>
</tr>
<tr>
<td>$H_0(\text{OutRank, TLOC})$</td>
<td>Coupling of package (OutRank) does not contribute to its test cost (TLOC) and require no more test cost than its peer.</td>
<td>We have strong evidence to support rejecting it.</td>
</tr>
<tr>
<td>$H_0(\text{Dual Instability, TLOC})$</td>
<td>Instability of package does not contribute to its test cost (TLOC) and require no more maintenance cost than its peer.</td>
<td>Evidence does not support rejecting it.</td>
</tr>
<tr>
<td>$H_0(\text{Dual Abstractness, TLOC})$</td>
<td>Abstractness of package (OutRank) does not contribute to its test cost (TLOC) and require no more test cost than its peer.</td>
<td>Evidence does not support rejecting it.</td>
</tr>
<tr>
<td>$H_0(\text{Dual Distance, TLOC})$</td>
<td>Distance of package from the main sequence does not contribute to its maintenance cost (RLOC) and require no more maintenance cost than its peer.</td>
<td>Evidence does not support rejecting it.</td>
</tr>
</tbody>
</table>
9.3.3 Regression Analysis

The coefficient of correlation that was obtained from the previous section cannot provide the prediction of the test effort. However, correlations are useful since they can indicate a predictive relationship that can be exploited. Therefore, we’ve applied linear regression analysis to determine the ability of the newly developed metrics that correlate with testability to predict package testability. We performed multivariate regression analysis to find out the relationship between the newly developed package metrics and the number of lines of code of the test package (TLOC). Building the testability prediction models are the outcomes of this analysis. We used linear regression analysis to assess the validity of the new metrics as predictors of package testability. As before, correlations significant at the 0.05 level are marked with (*), and the ones significant at 0.01 are marked with (**).

We’ve applied a multivariate linear regression technique. Multivariate regression analyses establish the relationship between a dependent variable (TLOC) and explanatory variables (the new metrics). Data from the five systems are integrated to form one data set for the regression analysis.

We know from previous sections that the newly developed metrics are not totally independent and, hence, must capture redundant information. Thus, not all of them are required in multivariate analyses. However, a standard procedure called stepwise selection was also used to select the necessary variables for multivariate analysis.

Table 9.21 presents the means and standard deviations for the variables on the combined data used in testability regression analysis.
Table 9.21 Means and Standard Deviations of the Data Used in Testability Regression Analysis

<table>
<thead>
<tr>
<th>Variable</th>
<th>Combined Data N=782</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mean</td>
</tr>
<tr>
<td>OutRank</td>
<td>0.072</td>
</tr>
<tr>
<td>Classes</td>
<td>15.44</td>
</tr>
<tr>
<td>LogClasses</td>
<td>2.16</td>
</tr>
<tr>
<td>TLOC</td>
<td>330.03</td>
</tr>
<tr>
<td>LogTLOC</td>
<td>1.99</td>
</tr>
</tbody>
</table>

For linear regression, we want to make sure that the relationship between the dependent variable (TLOC) and explanatory variables (the new metrics) is approximately linear. Also, it is required that the data follows a normal data distribution to make sure that the computed statistics are not misleading. However, this is not the case in many variables in our study. There are a number of transformations that can be used to approximate data to make them normally distributed such as logarithmic, square roots, and inverse transforms. Therefore, data screening and evaluation of the linearity condition for the regression analyses led to the natural log transformation of the variables the number of lines of code of test package (TLOC), and package size (classes). Table 9.22 provides Pearson correlations among the variables used in the testability regression analyses for the combined data.
Table 9.22 Pearson Correlations among the Variables Used In the Testability Regression Analysis

<table>
<thead>
<tr>
<th></th>
<th>OutRank</th>
<th>LogClasses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Log Classes</td>
<td>0.464**</td>
<td></td>
</tr>
<tr>
<td>LogTLOC</td>
<td>0.293**</td>
<td>0.519**</td>
</tr>
</tbody>
</table>

The regression analysis was run to predict package testability by the newly developed measures (OutRank) controlling for the natural log transformed package size (LogClasses). The regression analysis is for predicting the natural log of the number of lines of code of test package (logTLOC) which is the measure for package maintainability. Table 9.23 shows the results of a multivariate linear regression analysis for testability cost measured by the log of the number of lines of code of test package (LogTLOC). Regression results for (LogTLOC) indicate that the overall model of the new measures of package efferent coupling (OutRank) and package size (LogClasses) significantly predict the natural log of the number of package revisions \[R^2=0.273, \ R^2_{adj} = 0.271, \ F(2, 473) =146.104, \ p = 0.000\]. The prediction model accounts for 27.1% of the variance in the log of the number of lines of code of test package. Table 9.23 presents a summary of the regression model coefficients.
Our results demonstrate that packages with lower values of the newly developed efferent coupling metric (OutRank) have better testability than those with higher newly developed efferent coupling metric values. It is noted that the newly developed efferent coupling metric (OutRank) is shown to be a good predictor of our testability indicator.
CHAPTER 10

Threats to Validity

10.1 Introduction

In this section, we examine the threats to the validity of this study. Validity of any empirical study can be threatened in different ways. In general, the types of threats to the validity of any empirical study are: internal, external, construct, and content threat validity. Internal validity imposes demands on the study itself, and concerns the degree to which the dependent variable was influenced by the independent variable and not by some other variable. External validity refers to the ability to not only statistically but also analytically generalize the results of the study obtained to other software systems. Hence, threats to external validity regard the generalization of the study’s findings. Construct validity refers to the meaningfulness of measurements. Hence, threats to construct validity concern the relationship between theory and observation, and check if the measured variables may not actually measure the conceptual variable. It is required by this type of validation to show that the measurements are consistent with an empirical relation system. Content validity refers to the representativeness of sampling adequacy of the instrument’s content [120].

In the following, we analyze these four types of threats which may restrict the generality and limit the interpretation of our results.
10.2 **Internal Validity**

The collected maintenance and test data were collected from the subversion repositories of each particular release of systems. The maintenance and test data, which were available online for the selected systems, were collected under the assumption that all revisions performed during the maintenance history and all the test classes that used for unit testing of the packages were reported in the subversion repositories.

In addition, these collected maintenance and test data may depend on the ages of the systems. The likelihood for a package to be revised is expected to be higher with time. However, we follow a specific procedure for systems selection where the selected systems must be active over a reasonable maintenance history. In addition, we selected systems of different ages and found that their results led to the same conclusions. This gave us assurance that the collected maintenance and test data are reliable.

In the data extraction, we excluded all logs folders that were not directly related to the functionality of the system itself. For each system in the maintainability study, we extracted every log in the repository during the studied maintenance period which includes many revisions. For each system in the testability study, we extracted every test class in the system and assigned it to its source class. We do not claim that the logs, revisions, and test classes collected from the repositories are complete, in that it is possible that not all of the changes that have been applied during the chosen maintenance period of the analyzed systems were logged. Also, it is possible that not all the test classes that were used for testing these systems during the chosen maintenance period of the analyzed systems were documented in order to be collected. Furthermore, after having the maintainability and testability data ready at the class level, we
assigned every class’s data to its package in order to have package-level data for our study. We think that the assignment was done to our best level but we don’t claim that it is error free.

10.3 **External Validity**

The experimental results are the outcome of the analysis of four open-source systems and five open-source systems for the maintainability and testability study, respectively. The results were compatible with the expectations from the previous software engineering research studies of the characteristics of the higher quality systems. However, each system represents a different problem domain, and they may not be representative in terms of the numbers and sizes of packages. To ensure external validity, we have paid special attention to selection of these systems. Of course, the studied systems are not artificial cases and were used in similar empirical studies. The considered systems are of different domains, ages, and sizes, and the number of these systems and their sizes and the number of their packages are also comparable with those considered in similar empirical studies. We further selected only well-known open-source systems that have at least thirty non-third-party packages. However, additional studies conducted in different environments are necessary to determine the generality of the relationships.

The second external threat to validity is that all systems we considered in this study are implemented in Java. Other object-oriented programming languages, e.g., C++, have features that differ from those in Java. Therefore, we claim that our findings are valid for Java open-source systems. So, we do not recommend generalizing our results to non-Java open-source systems until this study is also conducted on case studies that are implemented in different object-oriented programming languages.
The third external threat to validity is that all nine of the studied systems are open-source systems, which may not be representative of industrial systems. There are certainly differences between open-source and industrial development. Therefore, we don’t recommend generalizing our results to close-source systems. However, the use of open-source systems in empirical studies is a common practice in the research community.

The fourth external threat to validity is using our own developed tool to examine Java packages and calculate the newly developed metrics and Martin’s metrics. Using an existing tool might require more development time than developing a new tool, since it requires reverse engineering the tool and extending its functionality to calculate the newly developed metrics. However, we cross-validated the new tool by comparing its results with other existing tools such as JHawk, JDepend, and Classycle. Also, we did a reverse engineering of the studied systems, and then manually verified our tool’s results. We found most of the corresponding values alike, which gave us assurance about the accuracy of our tool.

10.4 Construct Validity

The main construct validity threat in any study of external quality software attribute usually lies in how to quantify this attribute. As for the dependent variables used in our study, we have selected indicators whose construct validity has already been investigated in the literature. In this study, a package maintenance effort is measured as the number of changes made to code (Revision) and as the Revised Lines of Code (RLOC) made in a package within a period of time. Likewise, a package test effort is measured as the number of lines of code of its test package (TLOC). These indicators are a good approximate estimate for maintainability and testability, and have been used in a number of previous studies as explained in details in a previous section.
In this study, we follow the recommendation of Morasca [85] of using probabilities to estimate the two external attribute, namely, package maintainability and package testability. These probabilistic models use packet size, coupling measures as independent variables, and estimate the probability that a package will be frequently modified (REVISION) or will require costly modifications (RLOC). However, this study can be extended by considering some more accurate aspects about maintainability and testability such as recording the actual time and cost of a package maintenance and test to measure its testability and testability.

It is known that different lines of code can have different maintenance or test efforts; however, since it is difficult to measure the exact maintenance effort for each modified line of code, we assume that every modified lines of code is equal. This is true also for the number of revisions and the number of lines of code of test package (TLOC). In addition, the systems under the testability study are assumed to be tested using the functional unit test on the JUnit framework. The testability of a package would be different if other kind of test or test framework is used. As mentioned before, we use in this research the number of revised lines of code, number of revisions, and the number of lines of code of its test package to estimate the maintenance and test effort and not to measure them.

One important issue that may affect the construct validity is that during our study of the systems’ historical maintenance data there were some operations that we couldn’t track between the different releases of the studied systems such as changing class or package name or moving classes or packages from place to place. Therefore, after such a modification, the class or package is interpreted as a new class or package, respectively.
10.5 **Content validity**

To have content validity, our new developed metrics and the additional metrics must capture the notion of maintainability and testability. The RLOC, Revision, and TLOC quantify only some aspect of maintenance and test effort in this study. Moreover, the coupling metrics quantify various aspects and not all the aspect of the system. The development of measures that totally quantify the quality of object-oriented software is still in an ongoing research process. Our compact models which contain some of the new developed metrics were sufficient to some extent to predict package maintainability and testability. However, in order to enhance the content validity of this study and to validate new developed metrics, we should recruit software engineers and developers of different skill levels and use qualitative analysis techniques to examine the predictability and correlations between the new developed metrics and maintainability and testability.
CHAPTER 11

Conclusion and Future Work

In this research, a practical enhanced approach that manages packaging in object-oriented software development by analyzing all kind of dependency relations of packages is proposed. The newly developed coupling, instability, and abstractness metrics, which are based on Martin’s principles and on indirect responsibility, are tested and validated. We give an explicit formula for each metric. The results produced by these metrics are highly correlated to maintainability and testability.

Many metrics have been subject to serious criticisms, including the lack of a theoretical base or missing some appropriate measurement properties. Hence, two main frameworks, a property-based measurement framework and a DISTANCE framework, were used to axiomatically evaluate the newly developed metrics to demonstrate that they were valid. For validation purposes, the new metrics have been applied to seven well-known open source systems: Camel 2.0.0, Tomcat 7.0.6, JHotDraw 7.5.1, JEdit 4.5.0, Hadoop 2.2.0, Synapse 2.1.0, and Ant 1.92. The experimental study results support the conclusion that the new metrics produce better measurements of coupling, instability and abstractness, and produce good predictions of maintainability and testability.

The results of the maintainability statistical analysis clearly indicate that the null hypothesis pertaining to InRank, OutRank, and Dual Abstractness can be rejected. The results related to Dual Instability and Dual Distance show no significance in predicting a package
maintenance effort. Therefore, we can reject the null hypotheses and accept the alternate hypotheses. The study indicates a high significance of OutRank and Abstractness in predicting a package maintenance effort. Likewise, the results of the testability statistical analysis clearly indicate that the null hypothesis pertaining to OutRank can be rejected. The results related to InRank, Dual Instability, and Dual Distance show no significance in predicting a package testing effort. Therefore, we can reject the null hypotheses and accept the alternate hypotheses. The study indicates a high significance of OutRank in predicting the package testability effort.

In addition, our experimental study supports the claim that our new metrics can in early design help predict the maintainability and testability of software systems. Thus, our metrics should be useful in reducing costs by revealing potential maintainability and testability problems during early design. In specific, the newly developed metrics can be integrated with an IDE to offer guidance to the designer, the developer, and the tester. In addition, our metrics can be seen as a supporting tool to the design, develop, and test processes by identifying packages that are hard to maintain and test. This feature allows an IDE to issue an on-the-fly warning immediately after the recent decision that may affect package maintainability and testability. Further, even in later design stages, our metrics may help maintenance programmers to modify programs while respecting the existing design; this could reduce additional maintainability and testability issues. In practice, the prediction models can be automated and integrated with IDEs to estimate package maintainability and testability after the system is developed. That is, the modules in our tool that are responsible for calculating the variables that are involved in the constructed prediction models can be integrated with a Java editor. For each package in a system under development, the modified IDE can get the values of the independent variables and apply the
equations of the constructed prediction models and show a warning if the likelihood that the package will have maintainability or testability issues is high. By the help of this feature, software developers can review the code of the packages with low maintainability or testability. Also, testers can focus on and test such packages more than others to find all possible faults which will reduce the chances of discovering faults in these packages in the future. Finally, packages with low maintainability or testability ought to be fully documented, and hence minimize the time needed in the maintenance phase to comprehend the code and perform the required changes.

In this study, we introduced and tested responsibility with respect to indirect dependency. In future work, we would like to investigate whether indirect responsibility has additional application areas. Also, we would like to see if the distance metric [56] may be useful in determining maintainability threshold levels. Additionally, we wonder if there are useful relationships between our new metrics and the change history of a system.

We have shown that our indirect coupling metrics are useful in developing prediction models for maintainability and testability. However, we know that coupling is certainly not the only factor contributing to system maintainability and testability. In our future work, we want to improve our prediction models by including other factors such as cohesion and various complexity measurements. We believe, as we better understand maintainability and testability, that we will be able to significantly improve our prediction models.

This study conducts only quantitative analyses. The authors will extend this work by performing quantitative analysis such as measuring the dependent variables based on the
judgment of the experts and picking up top/bottom packages based on the newly developed metrics and asking expert developers and designers about whether these packages are actually high/low maintainability or testability.

Our experimental study should be extended. Although the systems studied were large, it is necessary to verify our outcomes on a larger number of open source and proprietary software systems using different development methodologies and developed by different programming languages.

This research, of course, is not the end. After completing this work, the author plans to continue doing his research in many related areas. In the following, the most important future works are shown:

First, the cohesion of the system as well as the package’s cohesion is a very important aspect to study along with the proposed metrics. The relational cohesion of a package can give useful insight about the quality of the design. The author is considering adding the relational cohesion to the proposed metric to come up with a broader decision about the quality of the system.

Second, while there is no silver bullet or magic bullet in software engineering, improvement is still possible. The author believes that we can have a predictive measurement of likelihood of change for packages. Most of the discussion in this research was about the internal factor the influence the change of packages. However, there are many external factors that influence the frequency of change for packages such as number of customers and suppliers of the system, type of customers and the system, and size and complexity of a package and the system. Each of these factors will contribute to the likelihood of change to varying degrees. It would be
motivating for future work to try to evaluate the contribution of each, and from these characteristics provide a new measurement predicting change.

Third, the field of software quality measurement is still a fresh discipline. Based on the proposed metrics, we will start the research on how to define and implement a software quality assessment tool that can measure the quality of the software during its lifetime. This assessment tool should predict any changes from a different level that may affect the quality of the software. The ability to detect the aging of the software is one of the most important features the author is willing to include in his research. The ultimate goal here is that we want to create a common platform for all design quality metrics as plug-ins for IDE to maintain the scalability and maintainability of the software.

Last, but not least, the ranking mechanisms proposed in this study are prominent and can be applied in different domain such as in business sector, political relationships, international trade, and the economy. The author believes that dependency relationships exist in many situations. For example, trades between countries, e.g. import and export trade, can be studied to rank the country based on their influence in the international trade, and measure the financial stability of each country. These types of study which depend on the global dependency can open a new era for many fields in different subjects.
REFERENCES


174


[109] TortoiseSVN. http://tortoisesvn.net/


