SOFTWARE MAINTAINABILITY AND TESTABILITY PREDICTIONS USING PACKAGE COHESION

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

by

Waleed Saleh Albattah

December 2014

© Copyright
All rights reserved

Except for previously published materials
Dissertation written by

Waleed Albattah

B.S., King Saud University, KSA 2003
M.S., King Saud University, KSA, 2006
PhD, Kent State University, USA, 2014

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
Artem Zvavitch, 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
# TABLE OF CONTENTS

**LIST OF FIGURES** ........................................................................................................... VIII

**LIST OF TABLES** ............................................................................................................... IX

**DEDICATION** .................................................................................................................. XI

**ACKNOWLEDGEMENTS** ................................................................................................. XII

**CHAPTER 1** INTRODUCTION ....................................................................................... 1

1.1 Overview ......................................................................................................................... 1

1.2 Research Framework ....................................................................................................... 2

1.3 Organization .................................................................................................................... 4

**CHAPTER 2** BACKGROUND ......................................................................................... 5

2.1 Software Maintenance ..................................................................................................... 5

2.1.1 Staged Model ................................................................................................................. 8

2.2 Software Evolution ......................................................................................................... 11

2.2.1 SPE Classification ......................................................................................................... 12

2.3 Software Evolution Laws ............................................................................................... 16

2.3.1 Law one: Continuing Change ....................................................................................... 17

2.3.2 Law two: Increasing Complexity ............................................................................... 17

2.3.3 Law three: Self-Regulation ......................................................................................... 17

2.3.4 Law four: Conservation of Organizational Stability (invariant work rate) .... 18

2.3.5 Law Five: Conservation of Familiarity ...................................................................... 18

2.3.6 Law six: Continuing Growth ...................................................................................... 19

2.3.7 Law seven: Declining Quality ..................................................................................... 19

2.3.8 Law eight: Feedback System ...................................................................................... 20
2.4 FEAST .......................................................................................................................... 20
2.5 Software Package Maintenance .................................................................................. 21
2.6 Software Measurement .............................................................................................. 24
2.7 Package Cohesion ....................................................................................................... 27
  2.7.1 Cohesion Metrics .................................................................................................... 28
  2.7.2 Package Cohesion Metrics ..................................................................................... 31
  2.7.3 The General Example ............................................................................................ 44
  2.7.4 Comparison Framework ........................................................................................ 45
CHAPTER 3 THE PROPOSED PACKAGE COHESION METRIC ................. 48
3.1 Introduction .................................................................................................................... 48
3.2 Towards Cohesion Classification .................................................................................. 49
  3.2.1 The Reuse-Release Equivalence Principle (REP) ................................................... 50
  3.2.2 The Common Reuse Principle (CRP) ................................................................... 50
  3.2.3 The Common Closure Principle (CCP) ................................................................ 51
3.3 Cohesion Types ............................................................................................................ 56
  3.3.1 Common Reuse (CR) ............................................................................................ 56
  3.3.2 Common Closure (CC) ......................................................................................... 58
3.4 Working Example .......................................................................................................... 60
3.5 What Is Good in This Metric Suite? ........................................................................... 61
3.6 The Flexibility of The Proposed Metrics ..................................................................... 63
3.7 The Unified Metric ....................................................................................................... 64
3.8 Theoretical Validation ................................................................................................. 65
  3.8.1 Non negativity and Normalization: ....................................................................... 66
3.8.2 Null value and maximum value: .................................................................66
3.8.3 Monotonicity: ...............................................................................................67
3.8.4 Merging of Unconnected Packages: .............................................................67
3.9 Applications of The new Metric ......................................................................67

CHAPTER 4 MAINTAINABILITY PREDICTION USING PACKAGE COHESION
..................................................................................................................................69
4.1 Introduction ........................................................................................................69
4.2 Internal and External Attributes .......................................................................70
4.3 Related Work ......................................................................................................71
4.4 Descriptive Statistics ........................................................................................75
  4.4.1 The software systems ....................................................................................75
  4.4.2 Maintenance Data .........................................................................................77
  4.4.3 Package Cohesion Data ................................................................................79
4.5 Exploring the relationship between package cohesion and maintainability using correlation 79
  4.5.1 Hypotheses ..................................................................................................81
  4.5.2 Statistical Analysis .......................................................................................81
  4.5.3 Results and Discussion ................................................................................82
4.6 Exploring the relationship between package cohesion and maintainability using regression analysis .................................................................................................85
  4.6.1 Dependent Variables ....................................................................................87
  4.6.2 Statistical Analysis .......................................................................................88
  4.6.3 Results and Discussion ................................................................................89
# LIST OF FIGURES

- Figure 1.1 Exploring the relationships with package cohesion ........................................3
- Figure 2.1 ISO Product quality model [17] ........................................................................6
- Figure 2.2 The versioned staged model ............................................................................9
- Figure 2.3 Packages' changes example ...........................................................................23
- Figure 2.4 Software quality and software measurements ..................................................24
- Figure 2.5 Internal and external attributes .......................................................................26
- Figure 2.6 The general example .......................................................................................44
- Figure 3.1 Package design examples ................................................................................52
- Figure 3.2 Package design examples ................................................................................53
- Figure 3.3 The working example ......................................................................................60
- Figure 3.4 The Cartesian coordinate system of CR and CC ............................................65
- Figure 4.1 Repository log example ...................................................................................77
LIST OF TABLES

Table 2.1 The weights of links between classes ....................................................... 43
Table 2.2 Cohesion values of the general example .................................................... 45
Table 4.1 Maintenance history details ................................................................. 76
Table 4.2 Details of the studied systems ............................................................... 76
Table 4.3 Maintenance data .................................................................................. 78
Table 4.4 Means and Standard Deviations of the variables used in the maintainability analysis ................................................................. 82
Table 4.5 Spearman's Rho correlations for maintainability analysis ....................... 83
Table 4.6 The results of the null hypotheses ........................................................... 84
Table 4.7 Means and Standard Deviations ............................................................ 88
Table 4.8 Summary of the model predicting the log number of package revisions (N=477) ........................................................................................................ 89
Table 4.9 Summary of the model predicting the log number of package revisions (N=477) ........................................................................................................ 90
Table 4.10 Summary of the model predicting the log number of revised lines of code (N=477) ................................................................. 91
Table 4.11 Summary of the model predicting the log number of revised lines of code (N=477) ................................................................. 91
Table 5.1 Details of the studied systems ............................................................... 101
Table 5.2 Testing data ......................................................................................... 102
Table 5.3 Means and Standard Deviations of the variables used in testability analysis .105
Table 5.4 Spearman's Rho correlations for testability analysis ................................106
Table 5.5 The results of the null hypotheses .........................................................107
Table 5.6 Means and Standard Deviations ..............................................................110
Table 5.7 Summary of the model predicting the log number of testing lines of code (N=783) ........................................................................................................................................................................111
Table 5.8 Summary of the model predicting the log number of testing lines of code (N=783) ........................................................................................................................................................................112
DEDICATION

For my parents, my wife, and my sons ...
ACKNOWLEDGEMENTS

I would like to acknowledge and thank my advisor, Dr. Austin Melton, for his countless hours of discussing, reading, encouraging, and most of all patience throughout the entire process of this dissertation. Thank you Dr. Melton for your support and for your precious time. A special thanks to my committee members Dr. Cheng Chang Lu, Dr. Ye Zhao, Dr. Pratim Datta, and Dr. Artem Zvavitch for agreeing to serve on my committee.

I wish to thank Computer Science Department for allowing me to conduct my research and providing any assistance requested.

Finally I would like to thank the statistical consulting at Kent State University Libraries for their help throughout the experiments of this research.

Waleed Saleh Albattah

December 2014, Kent, Ohio
CHAPTER 1

Introduction

1.1 Overview

With the increased importance of software measurements in assessing software properties, research studies have produced and are continuing to produce new software measures. One specific type of measure is cohesion. Cohesion refers to the degree to which the elements of a specific component belong together [59].

During software maintenance, developers spend at least 50% of their time analyzing and understanding software [56]. Software maintenance can be improved by assembling only related elements, e.g., classes, into components. A component should only include elements that are closely related. These components are called packages in object-oriented programming languages, e.g., Java. Package cohesion metrics measure the coherence of a package amongst its elements that should be closely related. Cohesion is an internal attribute of software that affects its maintainability. Following Martin’s design principles [19], high cohesive has as a goal to achieve software maintainability and promote its reusability [113][86].

Developing a software system that is maintainable is still a challenging task. With the great deal of maintenance costs and effort after the first installation of the software, designing a maintainable software system is the key goal. Many researches and empirical studies have been performed to improve the software design. R.C. Martin [19] proposed a well-known and well-accepted design principle suite that improves software design. His three package cohesion principles are the base of this research in which the new cohesion metric is developed. Martin
claims that following the design principles during the software development will improve both software quality and software maintainability.

Based on the design principles [19], the design of packages should maximize cohesion and minimize coupling. The proposed metric in this study will try to emphasize the package cohesion factors that can improve software maintainability and testability. This proposed metric has been theoretically validated. The research will include the theoretical validation, examine applications of the new metric, study the impact of package cohesion on software maintenance and software testing, and apply experimental validation for both the proposed metric and research assumptions.

1.2 Research Framework

This research contributes in the following:

• Investigates (examines) the impact of package cohesion principles on software maintainability and testability.
• Developing an accurate package cohesion metric based on the package cohesion principles.
• Investigates the relationship between package cohesion and two characteristics of software quality, software maintainability and software testability.

This research focuses on two main parts. First, to propose a new package cohesion metric based on the well-known and well-accepted package cohesion principles to overcome the drawbacks in the existing metric, Martin’s package cohesion metric (H) [19], and to assess the package cohesion accurately. The second part includes the following facets:

A. Evaluate the impact of the package cohesion principles on software maintainability and software testability.
B. Explore the ability of the proposed package cohesion metric in predicting both software maintainability and software testability in the early phase of software development cycle; i.e., design phase.

C. Investigate the relationship between package cohesion and software maintainability and software testability, figure 1.1.

D. Propose maintainability prediction models and testability prediction models to help developers evaluate software quality during the design phase of software development.

![Diagram](image)

**Figure 1.1 Exploring the relationships with package cohesion**

Chapter 3 of this dissertation has already been published:


Also, it is planned to submit Chapter 4 and Chapter 5 in the near future.
1.3 Organization

The remainder of this research is organized as follow. Chapter 2 presents a background of the research framework. Chapter 3 proposes the package cohesion metric and its theoretical validation. The investigation of the relationships between package cohesion on one side and the software maintainability and software testability on the other side are presented in chapter 4 and chapter 5 respectively. Chapter 6 concludes the research and suggests future works.
CHAPTER 2

Background

2.1 Software Maintenance

Software maintenance is known as the key and the last phase in the software development lifecycle. The role of the software maintenance is to keep the software updated. The early phases of software development cycle should be completed in the way that makes the software easily understood and maintainable [109]. During the maintenance phase, the software is modified to adapt to a new environment, improve performance, or to correct faults [108]. Different software maintenance definitions have been introduced in the literature. However, all definitions focus on the activities to modify the software after its first installation. One definition by IEEE Standard 1219-1992 [104] is: “Modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment.” The importance of software maintenance, which is the longest phase in terms of time in the software development lifecycle, is because of the great portion it forms of the total software development costs [107]. Companies spend a substantial amount of money for the maintenance of software systems. For example, Nokia Inc. has spent about $90 million for Y2K-bug preventive corrections [109][112]. Although software maintenance is a costly and challenging phase, it is poorly managed [114]. One reason for that is because of the lack of good predictors for software maintenance effort. This makes developing a reliable and accurate software predictors and measures even more serious concern. This research tries to develop an accurate cohesion metric that can help in software maintainability prediction of software system.
Software maintainability is defined as “the ease with which a software system or component can be modified”[133] [110]. According to ISO 25010-quality standard, figure 2.1, maintainability is one of the software product quality main characteristics. Maintainability characteristic by itself has some sub-characteristics, which are: Modularity, Reusability, Analyzability, Modifiability, and Testability. Maintainability and its sub-characteristics can be measured after the maintenance activities have been performed.

![ISO Product quality model](image)

**Figure 2.1 ISO Product quality model [17]**

In order to ease software maintenance, software developers must be aware of the internal attributes and characteristics of the software product that may affect its maintainability. Internal attributes such as coupling, cohesion, and complexity are main factors that expected to be related to software maintainability [19][30][33][35]. R.C. Martin [19] has developed well-known design principles for software development, which can help developers, especially those don’t have enough knowledge in designing programs, to overcome problems or mistakes happening during the software design phase. Problems during the early phases can cause lack of software understandability and software quality, which lead the software system to decay early. One internal attribute that needs to be taken care of during the software development is cohesion,
which is the scope of this research. Martin [19] has stated three package cohesion principles that relate cohesion to the software maintainability and software reusability. Software packages, which are the high level software components, that have high cohesion are expected to be high quality components; then to be easily and highly maintainable. There is a strong need to manage cohesion of the software packages in the early phase of software design.

Traditionally, software maintenance is known to be the umbrella that includes all changes made to the software product after it has been released. Preventive, corrective, perfective, and adaptive are the software changes classification [107]. Some significant changes are made to the software product after the first installation as the software evolves. This kind of changes is major change and can be applied to increase software capabilities. Although there might be some evolution or major changes during the software maintenance phase, we will consider any change made to the software after the first installation as a maintenance activity in general. We understand that knowing the difference between evolution and non-evolution changes is worthwhile and we will be giving an explanation of what the software evolution is meant.

The emergence of the first software evolution concept, by Lehman [4], raised a significant question about the difference between software maintenance and software evolution. Lehman documented that the evolution stage of software is inevitable and that it includes increases in size, complexity, and functions. While software maintenance has a standard definition, software evolution lacks one, which makes researchers interchange them sometimes. Bennet and Rajlich have contributed significant research [105][106] toward better understanding of these two concepts. They introduced a staged model for the software life cycle in which the evolution stage of the software is distinguished by specific types of maintenance activities during the software life cycle. The next section describes the staged model.
2.1.1 Staged Model

Software maintenance is known to include any type of changes after the software has been released for the first time. Researchers have classified changes into different types. One classification by [107] is:

- Perfective: Changes made on software functionality, or new requirements
- Adaptive: Changes to adapt to the software environment
- Corrective: Changes to correct errors
- Preventive: Changes to avoid problems in the future.

As stated by Lehman, software is changed considerably over time since its first release. The concept of viewing software maintenance as a uniform phase needs more investigation. One of the accepted models for software life cycle is a staged model introduced by Bennett and Rajlich [105][106]. The staged model consists of five distinct stages: initial development, evolution, servicing, phaseout, and closedown. Figure 2.2 from [106], shows the staged model, and a summary of each stage is given below from [105] and [106].
Figure 2.2 The versioned staged model

*Initial development*

In the initial development stage, the software is built based on a good understanding of clear initial requirements. All engineering alternatives have been explored and the software has been built in such a way that it can be easily and reliably changed in subsequent phases. The predictable changes can be anticipated in this stage, but the challenge is the unanticipated ones. During this stage, the team’s understandings of the domain, the problem, and the system’s architecture are very critical for future evolution.
Evolution

When the initial development is successful, the evolution stage starts. During this stage, iterative changes, major deletions and additions to software functionality are applied to extend and enhance software capabilities. Changes made during evolution are major changes. Iterations during the evolution stage provide increased learning fueled by users’ demands related to software functionality. Evolution is caused by these demands, changes in business practice, and the operating environment. The backbone of the evolution stage in figure 1.1 describes the case when a specific version of software has been released to the public, evolution continues and eventually a new version is released. The previous version can only get servicing. In this case, the strategic changes during evolution produce a new version while the servicing patches are produced for the previous version. Software evolution is discussed in detail in section 2.2.

Servicing

The evolution stops when software architecture or software understanding is lacking. While change is a basic operation in both evolution and servicing stages, the two stages are separated by the difficulty of the change. Software enters this stage when the costs for further changes exceed the profits. The software is considered mature and only minor changes could be made. During this stage, changes are difficult and expensive which make developers minimize them. At the end, software starts its aging or saturation stage.

Phaseout and Closedown

Phaseout is also called the decline stage. During this stage, no more servicing is provided and users may still use the software but without any further changes. The software becomes
outdated, but is still managed until the end of its life. During the closedown stage, the software use stops and new alternatives are recommended.

2.2 Software Evolution

Software needs to evolve in order to embrace several kinds of changes. These changes include but are not limited to new features added, old features deleted, and performance enhancements. Software evolution is concerned with facilitating software to evolve smoothly and easily so it can accommodate any new or changing requirements.

As stated by Fred Brooks, in his book “The Mythical Man-Month”, that over 90% of the costs of a typical software arise in the maintenance phase [1]. He also argues that any successful software needs to be maintained. Maintenance describes all changes made after the first software installation [2]. Since software evolution is a significant part of software maintenance, there is an extra care given to software evolution and its processes.

External world usually pushes software to evolve. Organizations, stakeholders, and feedback from software executions all push software to evolve and grow.

Software evolution has been discussed first by M. Lehman in 70’s [4]. He believes that software just like any natural species such as cities, societies, and ideas evolve over time each within its context [3]. Software evolves by changing its size, responding to market requests, accommodating and offering new features or functions, and providing satisfaction to its stakeholders. As large-scale systems keep working and satisfying its users’ expectations, they are evolving and responding to their environment’s needs. They must have appropriate and robust design choices that let them survive to accommodate changes over time by adding new code lines to legacy ones. It is an indication of a successful system if it accommodates changes and evolves along with providing satisfaction to its users.
Based on his experience and observations during his work at IBM, Lehman ended up with a set of eight laws for software evolution [2][5][6]. These laws reflect reasons that force software to develop and those that weaken its progress. They were originally observations and implications summarized in behavioral statements that have been called laws from the software engineering point of view [7]. These laws were not formulated at the same time, but rather they were formulated over a period of more than twenty years. He also has classified programs into three types; S, P, and E [2]. An S-type program is one that can be precisely specified and is unlikely to be changed. A P-type program is different from an S-type since it cannot be specified but instead, a process is made to find the best solution. The last type, E-type, is the most changing one and software evolution laws describe general principles for its evolution [8]. E-type programs are embedded in the real world and must be adapted to match changes in its environment [8].

In the next few sections, further discussions about software evolution are provided.

### 2.2.1 SPE Classification

The first presentation of SPE (Specified, Problem, Evolution) program classification was in [2]; then in [9] and [10]. This classification is presented to express the fact that programs are different in their evolutions. There must be a distinction between static specification programs and real world programs [8]. This distinction results in three types of programs; S (Specified), P (Problem), and E (Evolution) program types. Programs that are based on specifications are classified as S-type programs, while those based on real world needs are classified as E-type programs. Others that are classified as P-type often fall under S or E-type [8]. A short description for each type is given below.

**S-type programs** solve certain problems that are clearly specified. This kind of problem
is located in a narrow domain so they are not ambiguous. Good examples for such programs could be like performing simple math operations, such a program that sums two integers or that finds the bigger number. The problem here is clearly specified and identified, and the domain is very closed. So there is no uncertainty in this type of program.

In S-type programs, the problem is completely and clearly defined in its domain and it must be expected that a program will solve and meet specific needs by its users. Certainty is achievable by defining and specifying domain problems, and also in the provided solution or the requested need.

However, in some cases, program solution does not meet specific expectations because of a lack in problem definition and specification [3]. In this case, new programs need to be developed and constructed from scratch. Even if it is developed based on the rejected one, it is considered as a new program.

From the previous explanation, it is clear that program specification provides an explanation of what is required for a program in order to meet user satisfaction in a certain way. This will distinguish this type from others, like E-type that has some sort of uncertainty as will be discussed later.

Another interpretation suggests that S stands for static [11], which represents the opposite of an evolving one in terms of determining program specifications and properties.

**P-type programs** are known as real world problem programs. This type includes programs that provide solutions for problems that can be formally specified; however, considering the real world domain is very essential for its solution [10]. The problem that this program type tries to solve is not always clearly defined and specified. So despite the basic problem that can be identified, it may still have some uncertainty and be unclear to some extent.
Uncertainty causes different views in creating p-type programs. Both the problem specification and the solution provided are intended to match the real world state. Even if the problem is specified clearly, that does not imply necessarily that the given solution will meet users’ expectations. The essential feedback loops in this type of program try to match domain needs. The satisfaction of the provided solution is based on how much the solution matches the program domain [2]. It is very noticeable how uncertainty in P-type programs distinguishes it from S-type programs. While user satisfaction is based on program specification and problem statement in S-type programs, it is based on solution validity in problem domain of P-type programs. Solving real world problems leads to different observations, views, models, and solutions. This diversity does not guarantee providing an expected perfect solution that can fit into real world context. Consequently, continuous changes will occur to satisfy reasonable solution validity to some extent. On the other hand, an environment in which a program is embedded is changing by its nature, which makes it difficult to stop changing the program. This continuous change adds pressure on the program to evolve. Otherwise, it will become less effective and less satisfactory as time goes on.

One good example is a program to play chess [2]. The program needs to be specified by chess and procedural rules for the game. The former can be clearly defined but the latter needs a complete state analysis to decide the next move. Analysis may cover many possible moves of their developed games. With decision rules, the next move can be determined. Since a decision tree at each state can be very large due to many possible moves, such a program will approximate a solution that can be improved after it has been used. The program will be more practical after being used, evaluated, and feedback given to enhance its performance in the real world game.
P-type programs have been defined in the original classification [2][8][10] to make the scheme more comprehensive. However, it has been suggested that this type can be generally classified under one of the other two types. Especially when programs of this type are used to solve real world problems, they have E-type properties and its uncertainty. Despite that this type is not always considered as a separate class, there are some problems that cannot fit under S or E-type, such as with the chess game. Thus, for the scheme to be comprehensive, the P-type needs to be preserved [10].

**E-type programs** are the most changing program type. This type includes programs that automate human or social activities [2]. Later, the definition has been extended to include programs that automate and solve real world problems [3]. This type of program has an intrinsic property of evolution because it has to evolve. An E-type program is embedded in its real world domain and must be active and responsive to users’ needs and environmental changes. Therefore, continuous changes and enhancements are inevitable to satisfy users’ expectations. Some examples of E-type programs include operating systems, payroll systems, and billing systems.

Although problem statement and, program model and specification can be defined, the real world domain in which a program is embedded, has unlimited and changing properties that cause the program to evolve [10][12]. Program satisfaction depends on the domain it models and whether solutions match users’ needs. Requests for unpredictable changes and enhancements start to arise pushing the program to evolve. The nature of E-type programs is to have intrinsic feedback from an embedding domain.

E-type programs have an attribute of uncertainty because of continuous needs for solutions to be validated through real world domain applications [10]. The real world and its applications
change continuously, which makes program does not compatible to its domain as it was in previous release. The first law of software evolution, continuing change, supports this concept [4][2][6][5]. It states that “an E-type program must be continually adapted else it becomes progressively less satisfactory” [5]. E-type program evolution is normally driven by feedback changes. When a program evolves this proves that it is responding to satisfy the users’ satisfaction and domain needs. However, if it does not respond to feedback loops, it stops evolving and starts losing users’ satisfaction. E and P-type programs are closely related and they are different from S-type because they represent real world applications [2].

2.3 Software Evolution Laws

Laws of software evolution have been widely accepted. They have been formulated by M. Lehman, between 1974-1996, in context of software engineering. Lehman has formulated them based on his observations on IBM's OS/360 and OS/370 evolution [2][6]. Software evolution laws are concerned about satisfying the balance between two driving forces. First, forces those cause development to software. Second, forces those cause the decline of the software progress as a result of the first forces.

These are not laws, but rather they are basically observations and implications formulated in behavioral statements that are called laws, in the context of software engineering [7]. Some minor adjustments can be applied to the original formulation to make them more practical [53]. These eight laws relate to E-type systems that present solutions or applications in the real world and represent the first historical phase of software evolution study. The first three laws were introduced in the seventies [13] as observations of studying IBM systems. The fourth and fifth laws were presented in 1980 in [14]. The sixth was formulated in the early Nineties in [15]. The last two laws were introduced in 1996 in [5].
Studying the evolution laws is not the scope of this research. However, mentioning them here can help to understand software maintenance better. Next, a brief description for each law is provided from [5][2][14].

2.3.1 Law one: Continuing Change

“An E-type program that is used must be continually adapted else it becomes progressively less satisfactory”[5].

The real world environment, where a system is embedded, is changing. The software system needs to stay relevant to its environment by means of adaptation. Otherwise, it will start losing users’ satisfaction and then decay over time. Evolution is an intrinsic property of E-type software and it can be achieved by feedback loops.

2.3.2 Law two: Increasing Complexity

“As a program is evolved its complexity increases unless work is done to maintain or reduce it”[5].

This law states that as change after change has been performed, unstructured interactions between system elements increase. Unless preventative actions to reduce negative consequences are taken, evolution increases system complexity. Later, the system’s ability to satisfy expectations becomes difficult. If the system complexity growth is constrained and controlled, the system evolution becomes easier.

2.3.3 Law three: Self-Regulation

“The program evolution process is self regulating with close to normal distribution of measures of product and process attributes”[5].

This law is derived from the nature of the existence rule. Where, all who exists compete
and motivate each other for their existence. It can be observed in software evolution that there are different factors driving software to evolve. Feedback loops, organizational checks and balances, software attributes, and people along with many other factors are combined together to form self-regulation behaviors of the system [2]. Decisions and actions of people in program environments are not simply reflected in program evolution, but rather characteristics of dynamics, which have been determined during system maintenance processes, determine the trends of maintenance and enhancement processes [14].

2.3.4 **Law four: Conservation of Organizational Stability (invariant work rate)**

“The average effective global activity rate on an evolving system is invariant over the product life time”[5].

Stability is demanded in human organizations. Managerial decisions such as substantial changes in developers and budgets are avoided [14]. A level of control and budget allocation is provided for software projects. However, on one hand, this is forced by external constraints such as developers’ availability and on the other, as stated previously in law three, software complexity and attributes may constrain any expenses allocated to gain satisfaction [5]. This phenomenon of dispute forces creates a stable system evolution over time.

2.3.5 **Law Five: Conservation of Familiarity**

“During the active life of an evolving program, the content of successive releases is statistically invariant”[5].

This law is considered to be in association with the system complexity presented in law two. As system complexity increases, it becomes hard to conserve its familiarity affecting its evolution, which means software decay. Feedback loops and changes may affect release content,
then system understandability. It must be understood how a system is designed and how it works to conserve its familiarity. Otherwise, change implementation and ability for the system to evolve will be increasingly difficult because of increased complexity, which leads to system decay.

2.3.6 Law six: Continuing Growth

“Functional content of a program must be continually increased to maintain user satisfaction over its lifetime”[5].

It seems that this law is identical to the first law. However, both of them focus on the same phenomenon, but from different angles. The continuing change law is associated with software uncertainty [12] where software needs to be adapted to match the changing real world environment that embraces it.

Nevertheless, the continuing growth law focuses on the increase in software functions and size. For an E-type system to become truly pertinent to its domain, its functional and behavioral attributes have to continually increase over time. Software at its first construction, or before it is upgraded, has some requirements and specifications that are limited to some constraints such as budget allocated, due dates, and application understanding. Functions that have been excluded because of constraints or limitations sooner or later will arise as a demand for change in feedback loops, which cause software to inevitably grow. The fact is for bounded software with bounded resources and time embedded in an unbounded real world domain, there is no way to avoid software growth.

2.3.7 Law seven: Declining Quality

“E-type programs will be perceived as of declining quality unless rigorously maintained and
adapted to a changing operational environment”[5].

Relative to the previous laws, software is growing and so is a degree of uncertainty and complexity to some extent. Poor changes may cause different defects. As mentioned in the conservation of familiarity law, a degree of software design decay may increase over time causing quality declines. Direct rigorous intervention to prevent such defects as part of maintenance activities must be considered. Otherwise, software becomes less understood, less adaptable, more resistance to change, and less satisfactory, leading to decay.

2.3.8 Law eight: Feedback System

“E-type Programming Processes constitute Multi-level Feedback systems and must be treated as such to be successfully modified or improved”[5].

During the previous laws’ explanation, feedback has been referenced as a participant in software evolution and is highly related to them. Feedback systems can be formed from different loops, levels, and agencies. For a system to tolerate continuous change and respond to evolution as well as reduce system defects and decay causes, its performance must be monitored. The maintenance team needs to understand feedback characteristics and develop feedback means that reflect system and maintenance performance. The eighth law is central since it is intrinsic in the evolution process.

2.4 FEAST

As part of Lehman’s FEAST, which stands for Feedback, Evolution And Software Technology, project [5] in 1996, he sought to investigate the role and influence of feedback in the software process on software evolution. This project and its hypothesis form the eighth laws of software evolution and open a new discussion about the relationships between the laws [53].
The project supports the FEAST hypothesis. It also provides evidence for the laws of software evolution with some adjustments to them [54]. Size metrics and other software metrics have been used for the purpose of data collection. The project showed a valuable development in how measurement concepts can be applied to the study of software evolution. The ability to detect, measure, and control both feedback and its impact is a key progressive step in software management and execution. The initial results [6] were supportive to laws 1, 2, 4, 6, and 8 but data scarcity caused laws 3, 5, and 7 to remain without evidence. Later results [54] of studying four large software systems proved and increased confidence in the laws except law 4, which was neither supported nor negated.

Development of new metrics and further studies were significantly needed to investigate software lifetime. Software metrics are able to reveal software characteristics and how they change. With the increased study of software development and new technology introduction, such as Object Oriented Programming Languages, new metrics rather than size metrics, such as Line of Code, have been continually introduced and developed.

2.5 Software Package Maintenance

In Object-Oriented programming, a concept of modular programming is used. Where software systems are designed based on a technique that separates software functions into modular components. Classes, which are code elements abstractions, are grouped based on function similarity to ease software understanding and maintenance. Each component, also called package, subsystem, or category is responsible for implementing a specific feature(s) and can interact with other components through an interface. Splitting up set of classes into separate components, which are called packages in modern Object Oriented systems [55], improves system maintainability and development. Since this study focuses on packages which are units of
organization in object oriented systems and are also very common in Java programming language [19], the term “package” will be used.

As software is composed of a number of software packages that cooperate to perform its functionality, they are influenced by factors that affect the software containing them. The rest of this research will focus on packages and their properties in the context of software quality.

During software lifetime, a package changes in different ways. Classes within packages and their methods change over time. Internal relationships, between package classes, and external relationships, between packages, also change. New packages may be added, existing packages deleted or restructured, i.e., figure 2.3 shows an example of theses changes. Version n on the left hand side of the figure has three packages. In the next version n+1, the numbers of packages and internal and external relations have been increased. Also, the number of classes within packages has been increased. These changes may affect package properties, such as package cohesion.

A software package could change from one version to another due to the same reasons that cause the whole system to change but not necessarily at the same rate [22]. Its change is not separated from the change of the system in which it is included. The change of a package causes its size to change [24] and new code to be added or the existing one to be deleted. New classes and (or) methods with new functions may be added, and consequently new internal and (or) external relationships may be added. This, along with other major changes [23], and deletions may affect package properties.
As discussed in law eight, a feedback system provides software performance reflection that help developers to watch software properties. By developing feedback means, developers can make informed decisions that do not negatively affect software properties. Otherwise, code smells [20] and design smells [19], which are two sets of symptoms of software quality decline, start to emerge making software maintenance harder and then cause software to decay. To predict and assess software properties during the software’s development, measurement is used. Measurement, which is a cornerstone in software engineering [26], provides different measures to assess the software product attributes. Software metrics have many helpful uses [27][28]. The most important one is supporting software development decision-making [21]. It can promote understanding and assist in the development of software and its maintenance [29], as well as discovering drawbacks and defects [30] and supporting predictions [31].

On ISO 25010-quality standard, software quality includes eight characteristics: functional suitability, reliability, operability, performance efficiency, security, compatibility,
maintainability, and transferability, with their sub characteristics [17][18], shown in figure 1.1. Most software expenses are spent on its maintenance [1]. Software maintainability refers to the ease of maintaining software products in order to prevent or correct defects and their causes and response to new requirements and environmental changes [110].

2.6 Software Measurement

Software measurement is used to provide reflection of controlled software, figure 2.4. It increasingly plays a critical role in controlling and understanding the software. By definition, software measurement is a process in which measurable attributes of software are described by adjectives (qualitative) or numbers (quantitative). Software measures can be direct, not depend on other measures, or indirect, depend on one measure or more [32] [51], and quantitative, represented by numbers, or qualitative, not represented by numbers.

![Diagram of Software Quality and Measurements](image)

**Figure 2.4 Software quality and software measurements**

Software measurement supports decision-making [34] in development of software
products. It provides information to manage the software development process by making decisions, allocating resources, planning, and scheduling software activities [33] and enabling assessments and trade-offs during the software’s life cycle [21]. It is also used to detect errors [33] and predict complexity [44].

Software measurement has made a positive impact on software development process. With values provided by software measurement, managers and developers can control, make decisions, and, determine information about software attributes.

For example, relating the software complexity with software aging can be useful to mitigate the problem of software aging [25]. The study in [25] shows that software aging effects are related to software static features. Software metrics provide caution lights during software development.

Many software metrics have been proposed [35][36][37][38][39] to measure software in procedural paradigm [30]. However, some of the procedural software metrics are not sufficient in measuring Object Oriented software properties [45]. In addition, many other software metrics have been proposed [30][40][41][42][43][44][45] to measure object oriented software products, of which, some have been validated [33][41][46][47][48].

Software metrics can be classified into two classes: static and dynamic [50]. There is a difference between the number of invoked methods and the number of invocations. The former is static while the latter is dynamic. Static metrics measure software properties based on static analysis without execution, such as measuring complexity [49] from the code. In contrast, dynamic metrics measure software properties based on dynamic execution scenarios [50], such as dynamic complexity and object coupling.

Software quality has internal and external attributes [32]. The internal attributes include
size, cohesion, and coupling along with others. They can be measured by internal measures and cannot be seen or noticed externally. External attributes are related to software functionality such as reliability and maintainability. Internal attributes often help in measuring external ones. External attributes are affected by internal attributes and the latter can be used as indicators to the former, figure 2.5. For example, the increase of coupling or the decrease of cohesion can affect maintainability negatively. In some studies [30][52], internal attributes are called independent variables, and external attributes are called dependent variables.

Figure 2.5 Internal and external attributes

Software measurement is an endless process since software systems are subject to inevitable maintenance and changes. This study focuses on one of the internal attributes, package cohesion, and how it is related to software maintainability and testability.
2.7 Package Cohesion

Developers always tend to reuse solutions that have solved problems and which they are familiar with. Although the modular concept in Object Oriented programming eases software development and maintenance, designing reusable software components is still a challenge. While designing software to solve the problem at hand is demanded, it should be more flexible and accommodating for future changes and requirements.

The majority of software maintenance costs are spent to understand it [56]. To facilitate software understanding and maintenance, interrelated classes should be grouped into packages that are less coupled with each other, because increased coupling may cause change propagation and make software maintenance harder. A cohesion metric measures the coherence of a package among its classes. Cohesion is an internal attribute of software that is related to its maintainability and reusability. In accordance with the design principles [19], high cohesion is a goal to achieve software maintainability and to promote its reusability. In contrast, low cohesion may cause difficulty in reusing and maintaining a package because of disjoints inside it and relations with other packages. Consequently, design principles such as [19] state that a good system design must have high cohesion and low coupling. To construct a high maintainable software system, a package must be formulated based on highly interrelated classes so that its cohesion is maximized. Later, the reuse of the existing high cohesive package enables software maintenance and reduces costs. Because of its relation to software maintainability, the research on package cohesion is vital. The increasing importance of cohesion metrics encourages further research and development of them.
2.7.1 Cohesion Metrics

A number of cohesion metrics have been introduced in the literature on different levels, such as method, class, and package, and also on procedural and object-oriented paradigms. The focus is on the object-oriented paradigm, since it is the scope of this research. In this section, different cohesion metrics are introduced, while those on the package level, which is a focus of this research, are introduced in the next section. A number of metrics have been proposed in the literature on the class and method levels [41] [59] [60] [62][64-69] [70][71] [73-76]. Since our goal is to introduce the main idea of each metric, more details may be given for some metrics.

Chidamber and Kemerer [41] have developed a suite of six metrics based on the theoretical base of Bung ontology [57][58]. The six metrics have been theoretically grounded and empirically validated [41]. One of the six metrics is LCOM, originally defined in [70] as Lack of Cohesion in Methods. They define the cohesion of class [41] based on the degree of similarity between methods in using the same instance variables. The metric is given by:

\[ LCOM = |P| - |Q|, \text{ if } |P| > |Q| \\
= 0 \text{ otherwise} \]

Where \( P \) is the number of method pairs whose similarity is zero, and \( Q \) is the number of method pairs whose similarity is not zero. The LCOM value must fall within the \([0, \infty)\) range. Despite more than one class having a zero value of LCOM, that does not mean that they have the same degree of cohesiveness.

Henderson-Sellers et al. have improved this metric under a new name, LCOM* [62][64]. They have normalized the metric to the number of methods \( m \) and the number of instance variables \( a \) in a class. The metric value range is \([0,2]\) and it is given by:
where $\mu(A_j)$ is the number of methods accessing the same variable.

One drawback of these two metrics is that they do not state whether inherited methods are included in calculating the metric [59].

Li and Henry [69] have made improvements to the first release of Chidamber and Kemerer’s LCOM metric [70], by giving the following definition:

“LCOM = number of disjoint sets of local methods; no two sets intersect; any two methods in the same set share at least one local instance variable; ranging from 0 to $N$; where $N$ is a positive integer.”

Hitz and Montazeri [68] have interpreted the definition of cohesion introduced by Chidamber and Kemerer [41] using graph-theoretic terms as follows:

“Let $X$ denote a class, $I_X$ the set of its instance variables of $X$, and $M_X$ the set of its methods. Consider a simple, undirected graph $G_X(V, E)$ with:

$V = M_X$ and $E = \{(m, n) \in V \times V \mid \exists i \in I_X: (m \text{ accesses } i) \land (n \text{ accesses } i)\}.$”

The connection between any two nodes in the graph means that two methods have at least one instance variable in common. So the LCOM of a class is the number of connected components of $G_X$ ($1 \leq \text{LCOM}(X) \leq |M_X|$).

The second version of Hitz and Montazeri is the C metric (Connectivity) that expanded the original one to discriminate the case when LCOM has only one connected component by counting the number of edges of the connected component [59][73].

Briand et al [59] have adapted their already introduced cohesion measures [60][61] for object-based systems to object-oriented systems. The metric is defined on the level of classes,
and each class is a collection of data declarations and methods. The metric is adapted to measure cohesive interaction $CI$ of a class based on interactions between data declarations (DD-interaction) and methods (DM-interaction). The measure range is $[0,1]$, in which 0 is the minimum, and is given by:

$$RCL(c)=\frac{|CI(c)|}{|Max(c)|}$$

where $CI(c)$ is all DD and DM interactions, and $Max(c)$ is all the possible DD and DM interactions in the class.

Similar to Briand et al., Eder et al. have adapted the existing framework for cohesion in the procedural and object-based paradigms to be applicable in an object-oriented paradigm. They introduced three types of cohesion in object-oriented: method, class, and inheritance cohesion, and different cohesion levels defined for each type [65].

The cohesion metric proposed by Lee and Liang (ICH) [66] measures the cohesion based on information flow through method invocations within the class. As more parameters are passed between the invoked and invoking methods, the relations between them become stronger. The sum of method cohesion values is the class cohesion, and similarly, the cohesion of a set of classes is the sum of their cohesion values.

Gui and Scott [67] have used a similar approach as Lee and Liang [66] in proposing a system level cohesion metric that is simply the mean of all system classes’ cohesion. Their metric gives an evaluation of a component’s reusability.

Bieman and Kang [71] have introduced a cohesion metric for a class based on Chidamber and Kemerer [41]. Their class cohesion metrics consider the number of method pairs that use common instance variables. A method uses an instance variable directly if it reads or writes to it, and it uses it indirectly if it calls another method to read or write to that instance variable. They
have defined Tight Class Cohesion (TCC) and Loose Class Cohesion (LCC) measures based on
direct and indirect use of common instance variables. TCC is the relative number of directly
connected methods only while LCC is the relative number of either directly or indirectly
connected methods.

Ott et al. [75] have adapted an approach of module cohesion measurement that is based
on data slices [72] to measure the cohesion of a class in object-oriented.

Chae and Kwon [74] have developed a cohesion metric based on class members’
connectivity. The Cohesion Based on Member Connectivity (CBMC) metric measures the
cohesiveness of a class based on the patterns of interactions. They have identified two types of
methods: glue and normal methods. Glue methods are those that are joint in the sense that the
represented graph of a class becomes disjoint if they are removed. A connectivity factor is also
defined to provide the ratio of the number of glue methods to the number of normal methods.

Cohesion Among Methods of Classes is a metric proposed by Bansiya and Davis [76] to
assess the relatedness of methods and attributes in a class in which strong overlap between them
is an indication of strong cohesion. This metric can be applied during the design phase to
measure methods’ cohesion in a class, which is an improvement over other traditional metrics
that depend on implementation.

2.7.2 Package Cohesion Metrics

Cohesion measures have been introduced on higher levels of abstraction such as packages
[77][82][19][66][86][87][96], subsystems (similar to packages) [94][95], subjects (similar to
packages) [85], modules [78][83], clusters [79][80][93], subprograms [81], and systems [67].
Some of them were not on an object-oriented paradigm such as [78][79][81]. Others are
proposed to measure aspect cohesion for aspect-oriented systems such as [88][89][90][91][92].
The focus of this study is on the package-level cohesion for multiple reasons. First, the package level has not received the same interest as the class level cohesion. Second, as stated by Martin [19], classes are a very convenient unit for organizing small applications; however, they are too finely grained to be used as the sole organizational unit for large applications. A larger unit, i.e., package, can help to organize large applications. Third, this research improves Martin’s cohesion metric that is on package level. In the following, package cohesion metrics are summarized.

Approach by Misic

Misic [77] has defined a new approach to the coherence concept based on a generic model. The cohesion measure of the new approach is language and paradigm independent, and can be used at different abstraction levels and different phases of the software development cycle.

Misic criticized the previous measures in which they only focused on the internal organization and structure of the module (package) while his goal was to measure cohesion against the objective, function, or purpose of the module. He claimed that as the goal can be found outside the module rather than inside it, cohesion is an external property of the module, and that the internal organization and structure is not enough to measure cohesion.

The approach measures the similarity of package objects (elements). The similarity between elements can be measured by looking to the external clients’ usage patterns.

Method

The approach states that cohesion of an object set (collection of constituent parts of the system at any abstraction, e.g., class) measures how its elements contribute toward an externally
defined objective. Misic defined write and read range concepts. The write range of an object $O$, $W(O)$, refers to the set of objects (servers) used by this object (client). The read range of an object $O$, $R(O)$, refers to the set of objects (clients) that uses this object (server).

Given a set of objects $S$, let $R(S)$ be their client set (Read range), $S_w$ the subset that used to write its clients, and let $S_w(x)$ be the part of that subset that used to write the client $x$. Then, the coherence is given by the following formula:

$$\Psi(S) = \frac{\sum_{x \in R(S)} (#S_w(x) - 1)}{\sum_{x \in R(S)} (#S - 1)}$$

where $#S$ : number of elements in $S$.

$R(S)$: client set or read range

$S_w$ : subset of the set $S$ used to write the client x

The coherence measure proposed by Misic can be calculated internally or externally. For internal coherence, the summation in the numerator and denominator will be restricted only for clients inside the questioned set. Similarly, the summation will be restricted only for clients outside the questioned set to measure the external cohesion.

**Approach by Ponisio and Nierstrasz**

Ponisio and Nierstrasz [86] proposed a similar approach to measure package cohesion. The proposed contextual metric measures the cohesion based on the common use of the package classes by clients. Their metric does not consider the explicit internal dependencies within a package. They tried to measure cohesion based on the functionality of the package, which represents the package cohesion degree. The reused classes measure whether its functionality is related. The number of reused classes and package cohesion are in a direct relationship. The
approach idea is to propose the Common-Use (CU) metric that measures the package cohesion by taking into account the way a package’s classes are accessed by other packages.

**Method**

Common-Use (CU) measures the cohesion of package \( P \) by considering the use of its elements by the package clients. If all the clients use the same set of \( P \)'s elements, these elements share the same responsibilities of \( P \), and then \( P \) is cohesive. Instead, if the clients use different set of \( P \)'s elements, these elements have different responsibilities, and then \( P \) is not cohesive.

There is a need for weight to differentiate between client packages. Not all clients have the same degree in assessing \( P \)'s cohesion. The weight reduces the influence of \( P \)'s cohesion from the promiscuous clients.

Definition: “We define the weight of a (client) package \( P_{client} \) as the inverse of the number of connections that \( P_{client} \) has with other packages.”

\[
w(P_{client}) = \frac{1}{\text{fan in}(P_{client}) + \text{fan out}(P_{client})}
\]

The definition of CU is given as follow:

Definition: “We define Common-Use (CU) as the sum of weighted pairs of classes from the interface of a package having a common client package \((f)\), divided by the number of pairs that can be formed with all classes in the interface.”

\[
CU = \sum_{a,b \in I} \frac{f(a,b) \times \text{weight}(a,b)}{\#Pairs}
\]

Where

\[
\begin{align*}
I &= \text{interface}(P) \\
\#Pairs &= \frac{|I| \times (|I|-1)}{2} \\
C &= \text{clients}(a) \cap \text{clients}(b) \\
f(a,b) &= \begin{cases} 
1, & \text{if } C \neq \emptyset \\
0, & \text{otherwise}
\end{cases} \\
\text{weight}(a,b) &= \sum_{c \in C} \frac{w(c)}{|C|}
\end{align*}
\]
The value of CU is between 0, which represents that the interface classes of the package have disjoint responsibilities, and 1, which means that the interface classes of the package are used together.

Approach by Martin

Martin [19] presented a set of object-oriented package design principles. Three of these principles, package cohesion principles, try to help a software architect to organize classes over packages. These principles are: The Release Reuse Equivalency Principle (REP), The Common Closure Principle (CCP), and The Common Reuse Principle (CRP), discussed later in section 3.2. The three principles aim to provide a high quality of package cohesion.

Method

Martin [19] proposed a number of simple package level metrics. One of them is a relational cohesion of a package. The package cohesion metric is presented as an average number of internal relations per class. Regardless of the package external dependencies that are considered in other cohesion metrics [77][86], the metric measures the connectivity between package elements. This metric is quite simple to apply, and is given by:

\[ H = \frac{R + 1}{N} \]

Where H: package cohesion

R: number of internal relations

N: number of the package classes

The extra 1 in the numerator prevents cohesion H to equal zero when N=1. This metric gives all internal relations the same weight and disregards the external ones. It is also difficult to evaluate and compare package cohesions because the metric’s maximum value H differs based
on the number of classes $N$ in the package. However, it has been applied to software projects and widely accepted.

**Approach by Lee and Liang**

The cohesion metric proposed by Lee and Liang [66], discussed in section 2.7.1 of this research, measures the cohesion based on information flow (ICH) through method invocations. This measure can be applied on the package level by collecting the cohesion of the package classes.

**Approach by Xu et al.**

Xu et al. [82] proposed an approach to measure the package cohesion in Ada95 object-oriented programming language. The proposed metric is based on dependence analysis between package entities. It is assumed that the package may have objects and sub-programs.

**Method**

The package dependence graph (PGDG) describes all types of dependencies: inter-object dependence graph (OOG), inter-subprogram dependence graph (PPG), and subprogram-object dependence graph (POG). The method measures package cohesion according to PGDG. It assumes that package PG has $n$ objects and $m$ subprograms, where $n, m > 0$.

To present the measure in a unified model, a power for each object $O$ is denoted by $PW(O)$ and is given:

$$
PW(O) = \begin{cases} 
\text{Cohesion}(O) & \text{if } O \text{ is a package object} \\
\text{Cohesion}(PG(O)) & \text{if } O \text{ is a type object} \\
1 & \text{others}
\end{cases}
$$

The measure of each type of dependencies is given next.

**Inter-Object Cohesion:**
It measures the tightness between objects in the package. For an object A, the set A_DEP represents all objects that A depends on.

\[ O_{-DEP}(A) = \{B \mid A \to B, A \neq B\} \]

The power of object A is given by:

\[ PW_{-DEP}(A) = \sum_{B \in O_{-DEP}(A)} PW(B) \]

The cohesion of inter-object is:

\[
Cohesion(O_{-O}, PG) = \begin{cases} 
0 & n = 0 \\
PW(A) & n = 1 \\
\frac{1}{n} \sum_{i=1}^{n} PW_{-DEP}(A_i) & n > 1 
\end{cases}
\]

Subprogram-Object Cohesion:

The subprogram-object cohesion is defined as:

\[
Cohesion(P_{-O}, PG) = \begin{cases} 
0 & m = 0 \lor n = 0 \\
\frac{1}{n} \sum_{A \in P_{-O}(P)} PW(A_i) & m = 1 \\
\sqrt{m \sum_{i=1}^{n} Co(Prev) * p(P_i)} & Others 
\end{cases}
\]

Inter-Subprogram Cohesion:

To measure the inter-subprogram cohesion, a set P_DEP(P) is introduced for each P:

\[ P_{-DEP}(P) = \{M \mid P \to M\} \]

The inter-subprogram cohesion \(Cohesion(P_{-P}, PG)\) is:
\[ \text{Cohesion}(P, PG) = \begin{cases} 
0 & m = 0 \\
1 - \frac{\prod_{i=1}^{m} |P \text{DEP}(P_i)|}{m-1} & m > 1 
\end{cases} \]

Package Cohesion:

Two ways are defined to measure package cohesion after calculating the three measurements individually:

Each measurement occupies a field in cohesion measure:

\[ \text{Cohesion}(PG) = \langle \text{Cohesion}(O_O, PG), \text{Cohesion}(P_O, PG), \text{Cohesion}(P_P, PG) \rangle \]

OR

Integrate all of them as follow:

\[ \text{Cohesion}(PG) = \begin{cases} 
0 & m = 0 \\
k \cdot \text{Cohesion}(P, PG) & n = 0, m \neq 0 \\
\sum_{i=1}^{3} k_i \cdot \text{Cohesion}_i(PG) & \text{Others} 
\end{cases} \]

where \( k \in (0,1] \); \( k_1, k_2, k_3 > 0 \), and \( k_1 + k_2 + k_3 = 1 \).

\[ \text{Cohesion}(PG) = \text{Cohesion}(O_O, PG) \]
\[ \text{Cohesion}(PG) = \text{Cohesion}(P_O, PG) \]
\[ \text{Cohesion}(PG) = \text{Cohesion}(P_P, PG) \]

Xu et al. [82] claimed that according to the definitions, it is easy to prove that the measure satisfies the four properties given by Biand et al. [59][97] to develop a good cohesion measure.

However, an Ada package represents a logical grouping of declarations. The role of an Ada package is similar to the role of class in other languages such as Java [87]. Thus, this
package cohesion metric cannot be applied on the unified example in the next section. An Ada package actually falls under the class cohesion metric level.

Approach by Zhou et al.

Dissimilarly, Zhou et al. [87] proposed an approach to measure package semantic cohesion called the Similar Context Cohesiveness (SCC). In this approach, the common context is used to assess the degree of relation between two components. SCC measures the inter- and intra-package dependencies that can reveal semantic cohesion between components.

Method

The proposed package cohesion measure SCC is based on the component context. The context of component \( c \) is composed of two sets: the components that depend on \( c \) and those that \( c \) depends on. The SCC metric is given by:

\[
SCC(p) = \begin{cases} 
\sum_{i \neq j \in E(p)} \frac{Wgt(c_i, c_j)}{m(m-1)} & \text{if } m > 1 \\
1 & \text{if } m = 1 
\end{cases}
\]

where \( m \) : number of components \( c \) in \( p \)

\[
Wgt(c_i, c_j) = CCS(c_i, c_j) + \text{Dep}(c_i, c_j)
\]

\[
\text{Dep}(c_i, c_j) = \begin{cases} 
1 & \text{if } c_i \rightarrow c_j \text{ or } c_j \rightarrow c_i \\
0 & \text{else}
\end{cases}
\]

\( CCS(c_i, c_j) \): denotes the similarity between the contexts of two components \( c_1 \) and \( c_2 \), and given by:

\[
CCS(c_1, c_2) = kRSS(c_1, c_2) + (1-k)DSS(c_1, c_2)
\]

\( k \): represents the position’s importance

\( RSS(c_1, c_2) \): similarity between \( S_R(c_1) \) and \( S_R(c_2) \)
DSS(c1, c2): similarity between $S_D(c1)$ and $S_D(c2)$

$$S_R(c) = \{c_i | c_i \xrightarrow{d_s} c\}$$
$$S_D(c) = \{c_i | c_i \xrightarrow{d_s} c\}$$

Approach by Abdeen et al.

The approach proposed by Abdeen et al. [96] is based on Simulated Annealing technique. The approach aims to reduce package coupling and cycles by moving classes between packages. Two metrics have been defined for this purpose, coupling and cohesion metrics.

Method

The approach automatically reduces package coupling and cycles by moving classes between packages considering the existing class organization and package structure. This approach can help maintainers to define: the maximal number of classes that can change their packages, the maximal number of classes that a package can contain, and the classes that should not change their packages or/and the packages that should not be changed. A set of measures is defined to determine and quantify the quality of a package. The number of package dependencies ($|P_D|$) normalizes these measures.

The package cohesion metric is defined to be the direct dependencies between its classes. Hence, the cohesion of a package $P$ is proportional to the number of its internal dependencies ($|P_{Int.D}|$) according to the Common Closure Principle (CCP) [19]. The cohesion quality is given as follows:

$$Cohesion_Q(p) = \frac{|P_{Int.D}|}{|P_D|}$$

Where $|P_D|$ is the number of all internal and external dependencies of classes in the package.
Other studies have proposed some cohesion metrics on different abstractions, such as subsystems [94][95] and subjects [85], but that are similar to the package level. In the following, a brief description is given for each.

**Approach by Bauer and Trifu**

Bauer and Trifu [94] have proposed an approach, architecture-aware adaptive clustering, to produce meaningful decompositions in a system. They have evaluated their approach by defining two metrics: the average cohesion of a subsystem and the average coupling between subsystems.

**Method**

The approach was based on providing a better understanding of the system. They tried to recover from the original decomposition, and then impose an appropriate structure. The new structure aims to maximize subsystems cohesion. To evaluate the recovered subsystem decomposition, they performed a comparative study that is based on two criteria, accuracy and optimality. For accuracy, they compared the resulting decompositions with both the original package structure and the ideal Common Reuse Principle structure of [19]. For optimality, they used some optimality metrics to show whether the resulting decompositions have high cohesion and low coupling. To evaluate their approach, two metrics were defined, average cohesion of the subsystems and average coupling between the subsystems of a given decomposition. The average cohesion metric is given by:

\[
\text{avgCohesion}(D) = \frac{\sum_{S_i \in D} \frac{n\text{oInternalEdges}(S_i)}{|S_i| \times (|S_i| - 1)}}{|D|^2}
\]

where \( D \): a composition
Our operator simply moves the orphan to the subsystem date for an orphan, i.e. a subsystem candidate containing crossover operator discussed above. The operator connects two subsystem candidates with strong association point in the dependency graph. Similarly, the subsystem candidates by unifying their classes. The process of locating them to those subsystem candidates which have the strongest connections to the classes. The process of locating them to those subsystem candidates which have the strongest connections to the classes. The process of locating them to those subsystem candidates which have the strongest connections to the classes.

Two new children consisting of both old and new subsystems are collected (step 3) and distributed over the remaining subsystem candidates (step 4). In this step we consolidate duplicated classes are deleted (step 2), their non-duplicated classes are collected (step 3) and distributed over the remaining subsystem candidates (step 4). In this step we consolidate.

Furthermore the operators are designed to be non-destructive and to preserve a complete subsystem candidate as far as possible. Thus the newly created individuals are likely to have an infeasible solution. In this case we do not waste computing time on infeasible solutions.

For good starting populations, two competing properties have to be used to create an initial population that might help the GA needs diversity and complete decompositions, so we do not waste computing time on infeasible solutions.

The building block theory tells us, that the GA constructs building blocks from small subsystems and mutually integrates them as new subsystem candidates. After choosing the parents, the operator selects a set of crossover points for the new subsystem candidates. Then, the crossover operator creates a new individual by combining the selected parents.

The strategy for finding highly fit individuals may vary and complete decompositions, so we do not waste computing time on infeasible solutions. On the other hand, the GA needs diversity and to preserve a complete subsystem candidate as far as possible. Thus the newly created individuals are likely to have an infeasible solution. In this case we do not waste computing time on infeasible solutions.

The fitness function consists of cohesion, coupling, complexity metrics as well as cyclic dependencies and bottlenecks heuristics. The value of each individual function is between 0 and 1, where the optimal value is 1. The cohesion of a system $s$ is the summation of cohesion values for the individual subsystems in $s$. The cohesion for a subsystem $s_i$ is measured by counting the number of different classes in $s_i$ known by some class $c_j \in s_i$, $(\#k(c_j))$, and dividing this by the square number of classes in $s_i$, $(\#c(s_i))$. The resulting value can be normalized by dividing it by the number of subsystems $(\#s)$.

$$cohesion(s) = \frac{\sum_{i=1}^{\#s} \frac{\sum_{j=1}^{\#c(s_i)} \frac{\#k(c_j)}{\#c(s_i)^2}}{\#c(s_i)}}{\#s}$$

**Approach by Seng et al.**

The approach by Seng et al. [95] aims to develop existing object-oriented system decompositions by defining new decompositions with better metric values and fewer violations of design principles. They defined the problem as a search problem. The quality of the resulting subsystem decompositions is measured by the fitness function that combines software metrics and design heuristics.

**Method**

The fitness function consists of cohesion, coupling, complexity metrics as well as cyclic dependencies and bottlenecks heuristics. The value of each individual function is between 0 and 1, where the optimal value is 1. The cohesion of a system $s$ is the summation of cohesion values for the individual subsystems in $s$. The cohesion for a subsystem $s_i$ is measured by counting the number of different classes in $s_i$ known by some class $c_j \in s_i$, $(\#k(c_j))$, and dividing this by the square number of classes in $s_i$, $(\#c(s_i))$. The resulting value can be normalized by dividing it by the number of subsystems $(\#s)$.

**Approach by Tagoug**

Tagoug [85] has proposed coupling and cohesion metrics on subjects, which are quite
like packages. Each subject is a collection of classes. The approach aims to measure cohesion and coupling at the system level. The quality metric, which combines cohesion and coupling values, measures the decomposition quality as early as the analysis and design phases of the development life cycle.

**Method**

The two metrics measure the quality of object-oriented decomposition. The cohesion metric focuses on the interactions of components inside a subject, while the coupling metric focuses the interactions of components among subjects. The cohesion of subject \( E \) is given by:

\[
C(E) = \frac{\sum_{i=1}^{n-1} \sum_{j=i}^{n} W_{ij}}{W_{\text{max}} \cdot \frac{n(n-1)}{2}}
\]

\( E \): a set of classes of \( S \).

\( W_{ij} \): the sum of the weights of links in \( L_{ij} \).

\( L_{ij} \): the set of all links between classes \( P_i \) and \( P_j \).

\( W_{\text{max}} = \max \{W_{ij}\} \) in system \( S \)

\( n = |E|, n > 1 \)

The cohesion value is between 0, no links between classes, and 1, maximum links with maximal weight. The weights of links between classes of a subject are ordered in table 2.1 based on the degree of association according to the object-oriented expert designers.

<table>
<thead>
<tr>
<th>Links Type</th>
<th>Weights (( W_{ij} ))</th>
</tr>
</thead>
<tbody>
<tr>
<td>Whole Part Structure</td>
<td>0.9</td>
</tr>
<tr>
<td>Inheritance</td>
<td>0.8</td>
</tr>
<tr>
<td>Instance Connection</td>
<td>0.7</td>
</tr>
<tr>
<td>Message Connection</td>
<td>0.6</td>
</tr>
<tr>
<td>Conceptual Link</td>
<td>0.5</td>
</tr>
</tbody>
</table>
2.7.3 The General Example

While we try to understand each of the existing approaches, we may not fully understand them. The following example tries to fill the gaps and clarify how all approaches can be applied to the same example.

The following example has six packages. Package $P_1$, which is the questioned package in this example, has four classes. The example aims to measure the $P_1$ cohesion based on the approaches explained previously. Again, all calculations are made based on our own understanding of each approach.

![Figure 2.6 The general example](image)

Table 2.2 presents the cohesion values of package $P_1$ for the different approaches:
### Table 2.2 Cohesion values of the general example

<table>
<thead>
<tr>
<th>Approach</th>
<th>Cohesion</th>
<th>Metric</th>
<th>Value</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>Misic [77]</td>
<td></td>
<td>$\Psi(S)$</td>
<td>0.33</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Ponisio and Nierstrasz [86]</td>
<td></td>
<td>CU</td>
<td>0.125</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Martin [19]</td>
<td></td>
<td>$H$</td>
<td>1.25</td>
<td>&gt;0</td>
<td>$\infty$</td>
</tr>
<tr>
<td>Zhou et al. [87]</td>
<td></td>
<td>$SCC(p)$</td>
<td>0.36</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Abdeen et al. [96]</td>
<td></td>
<td>$CohesionQ(p)$</td>
<td>0.29</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Bauer and Trifu [94]</td>
<td></td>
<td>$avgCohesion(D)$</td>
<td>0.67</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Seng et al. [95]</td>
<td></td>
<td>$cohesion(s)$</td>
<td>0.25</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Tagoug [85]</td>
<td></td>
<td>$C(E)$</td>
<td>0.67 *</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

* Assuming that all the connections are instance connections with 0.7 weights.

#### 2.7.4 Comparison Framework

The general example in the previous section motivates us to investigate more about the differences between the studied approaches. The comparison framework has a number of key attributes that may reveal the reasons of different values in measuring package cohesion. The comparison between package cohesion approaches shows some differences in the way that cohesion is measured. A reason for that is how cohesion itself has been defined and interpreted. Each approach has its own interpretation of cohesion. Thus, they have different objectives. For example, some approaches, such as Martin [19] and Abdeen et al. [96], defined cohesion to be the connectivity of internal structure of the package. However, other approaches, such as Misic [77] and Ponisio and Nierstrasz [86], defined cohesion to be the functional property that can be measured externally. In this section, an attribute-based comparison between the popular package cohesion approaches is presented to guide future research of package cohesion metrics as well as practitioners who look for a suitable metric for their objectives. A set of attributes used in this comparison is given next.
**Objective**: shows the goal that an approach tries to address. Since cohesion has been defined and interpreted in different ways, the approach objective determines whether cohesion is structural or functional.

**Internal vs. external**: determines whether the approach objective can be satisfied internally or externally. Internal means that cohesion can be measured based on internal attributes of the package, while external means that cohesion can be measured based on external attributes of the package.

**Number of relations**: regardless of having internal or external objective, does the approach consider the number of relations as a factor to measure cohesion?

**Direct vs. indirect relations**: what kind of relations between components is considered, is it direct or indirect relations?

**Normalized vs. non-normalized**: specifies if the metric value is normalized (e.g., between 0 and 1, Obviously, 0 means that cohesion is at low level, while 1 means that cohesion is good).

**Scale**: specifies the type of the scale used in the measurement. It could be nominal, ordinal, interval, ratio, or absolute [32].

**Validation**: indicates whether the metric proposed is (theoretically or experimentally) validated.

**Weighting**: if the relations between components are considered, are they have the same weight, or are their weights different?

**Level**: specifies the level of abstraction that the cohesion metric tries to measure. For the approaches we have in this comparison, the level is the package level.

**Phase**: indicates the phase of software development cycle when the metric can be applied, such as design or implementation. If the metric can only be applied on code, the phase will be implementation. If the metric can be applied as soon as design, the phase will be design.
The following table addresses the attributes of different approaches.

<table>
<thead>
<tr>
<th>Phase</th>
<th>Level</th>
<th>Weighting</th>
<th>Validation</th>
<th>Scale</th>
<th>Normalized?</th>
<th>Direct vs. Indirect relations</th>
<th>Number of relations</th>
<th>Internal vs. External</th>
<th>Objective</th>
<th>Cohesion metric</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Object set</td>
<td>Package</td>
<td>Package</td>
<td>Package</td>
<td>Package</td>
<td>Package</td>
<td>Package</td>
<td>Package</td>
<td>Package</td>
<td>Package</td>
</tr>
<tr>
<td></td>
<td>No (1)</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>No (3)</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>No (2)</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

- **Normalized?**
  - (1) Satisfies only the first three properties of [84][99], and it does not for the fourth.
  - (2) It has been applied on software projects since 1994.
  - (3) Quite similar to packages.
CHAPTER 3

The Proposed Package Cohesion Metric

3.1 Introduction

Developers tend to reuse solutions that have solved problems and that they are familiar with. Although the modular concept in object-oriented programming eases software development and maintenance, designing reusable software components is still a challenge. While designing software to solve the problem at hand is demanded, it should be flexible and accommodating for future changes and requirements.

The majority of software maintenance costs are spent to understand the software itself [56]. In accordance with the design principles [19], high cohesion is a goal to achieve software maintainability and promote its reusability. In contrast, low cohesion may cause difficulty in reusing and maintaining a package because of disjoints inside it and relations with other packages. Consequently, design principles, such as [19], state that a good system design must have high package cohesion. To construct a highly maintainable software system, a package must be formulated based on highly interrelated classes so its cohesion is maximized. Later, the existing high cohesive packages improve the software reuse and maintenance, and reduce costs. Because of its relation to software reuse and maintenance, the research on package cohesion is vital. The increasing importance of cohesion metrics encourages further research and development of them.
3.2 Towards Cohesion Classification

Since software maintenance is a continual process throughout the software life, the software parts need to be organized to meet growth in code lines and functions. One organization level is to allocate system’s classes into packages. A package groups the related classes together in order to ease software maintenance and reuse. The more the classes in a package are related, the greater the package cohesion. The cohesion of the package can assist the software maintenance and reuse. The important question is what are the criteria that need to be addressed in order to allocate the classes into packages?

A number of cohesion approaches have been proposed on class and method levels [41][59-60][62][64-76]. Other cohesion measures have been introduced on higher levels of abstraction such as packages [77][82][19][66][86][87][96], subsystems (similar to packages) [94][95], subjects (similar to packages) [85], modules [78][83], clusters [79][80][93], subprograms [81], and systems [67]. Some of them do not involve the object-oriented paradigm such as [78][79][81]. Others are proposed to measure aspect cohesion for aspect-oriented systems such as [88][89][90][91][92]. The focus of this study is on the package-level cohesion for multiple reasons. First, the package level has not received the same interest of the class level cohesion. Second, as stated by Martin [19], classes are very convenient unit for organizing small applications; however, they are too finely grained to be used as the sole organizational unit for large applications. A larger unit, i.e., package, can help to organize large applications. Third, this research improves Martin’s cohesion metric that is on package level.

R. C. Martin [19] has outlined six principles, which help in package design and which have become well known and well accepted. The first three principles, package cohesion principles, help to allocate system classes to packages. This allocation can help to manage the
software during its development. In the following, the three package cohesion principles and their explanations are from [19].

3.2.1 The Reuse-Release Equivalence Principle (REP)

“The granule of reuse is the granule of release”

According to [19], this granule of reuse and release is the well-known package. The size of reused code should be the same as the size of released one. If the one decides to reuse someone else’s code, s/he needs a guarantee that the support will continue and the release of new versions will be of the same size. To ensure the reusability of the code, the author must organize the classes into reusable packages. As the REP principle states, the grain size of reuse (package) shouldn’t be smaller than the size of release.

In short, if the package has classes that should be reused, it should not have others that are not reused. The goal is to have all the classes in the package reusable by the same user. Not to have some classes in the package that the user needs and others that are not.

3.2.2 The Common Reuse Principle (CRP)

“The classes in a package are reused together. If you reuse one of the classes in a package, you reuse them all.”

The principle helps decide classes should be grouped together. The classes that are reused together should be in the same package. CRP states that the classes of a package should be inseparable, which means if a package depends on this package, it should depends on all of its classes. In short, classes that are not tightly related to each other should not be in the same package.
From another view, the CRP tells what classes should not be in the package. For example, if a class in a package uses only one class in another package, the using package will depend on the used package. In this case, if the used package is released, the using package needs to be validated and rereleased even if the new release of the used package is because of changes in classes that the using package does not use. In this case, the revalidation and rerelease will be more work than necessary.

3.2.3 The Common Closure Principle (CCP)

“The classes in a package should be closed together against the same kinds of changes. A change that affects a package affects all the classes in that package and no other packages”

From the maintenance point of view, though the change is not avoidable, it should be controlled. If the changes are made on one package, there is no need to rerelease or revalidate other non-dependent packages. While the previous two principles, REP and CRP, focus on the reusability, the CCP focuses on the system maintainability. If a change is made on the code, it would be better to be on one package rather than being on many packages. If all changes are made on one package, then only changed package needs to be released. The packages that do not depend on the changed package do not need to be rereleased or revalidated. Classes that share the same function are related to each other, so they are more likely to be changed together because they have the same reasons for change. Such classes must be located in the same package. The classes that are tightly related will change together. Therefore, they should be in the same package. The effort regarding revalidating and releasing of software will be minimized.

Martin proposed the rational cohesion metric \( H = \frac{R+1}{N} \). However, this metric doesn’t conform to the three principles presented by Martin [19]. \( H \) measures the ratio of the relationships between classes of the package. This simple concept doesn’t measure the common
reuse or the common closure of the package, but rather it may measure the ratio of the relationships between classes. The H metric depends on the number of relations rather than how these relations are designed. The H metric doesn’t distinguish with regard to which class depends on the other as long as there is a relation between them. In this case, a well-designed package and a badly designed package could have the same cohesion value. Figure 3.1 shows two examples of good and bad package design.

In figure 3.1(a), the using package depends on both classes of P1. So, the design of the package is cohesive in terms of use, and it follows the CRP principle. This principle denotes that all the classes in the package should be used together by the same user and the classes should not be separated. In figure 3.1(b), the package design does not follow the two principles. The using package uses only one class of P1. So, the classes are not cohesive in terms of use. These two designs must be differentiated by CRP. Martin’s H metric does not distinguish between these two examples. The H value for both examples is 1 even though the design in (b) does not follow the CRP principle of cohesion.

![Figure 3.1 Package design examples](image)

Another example is given in figure 3.2. In figure 3.2(a), both classes in P1 depend on the same set of packages, P2 and P3. They are cohesive in terms of common closure because they
are closed to the same reasons for change. Any change in P2 or P3 will affect both classes of P1. The design of P1 conforms to the CCP principle, which denotes that all the classes should be closed to the same kind of changes. In figure 3.2(b), the package design does not conform to the CCP principle. The P1 classes depend on different sets of packages. C1 depends on P3 and P2 while C2 depends only on P2. So, they are not closed to the same kind of changes. Also, if P1 is affect by a change in P3, only C1 will be affected and not all P1 classes. Clearly, there is a difference in the two package designs in terms of the common closure concept. However, H doesn’t differentiate between them. The H value for both designs is 1.

![Figure 3.2 Package design examples](image)

The H metric has some drawbacks that we try to overcome. First, H metric does not have an upper bound. Its value can be greater than 0 with no upper limit, which makes it difficult to manage and control the packages of a software system. Second, the metric considers, i.e., measures, the number of relationships rather than the common reuse or the common closure of the package. For example, since the H values for well designed package and the poorly designed package in the above examples are the same, then the H metric may not help a software developer determine whether the package design conforms to the package cohesion principles.
It focuses on the ratio of the internal dependencies to the number of classes in the package. The ratio can show the extent to which classes in the package are connected; it does not conform to the package cohesion principles. Third, since cohesion is an attribute that can be measured from outside the package, the metric doesn’t consider any external considerations. However, the common reuse and the common closure of the package are attributes that are measured based on external considerations. For a package, the incoming dependencies represent the use of the package by the using packages, which is stated in the CRP principle. Also, the outgoing dependencies represent the common closure of the package, which is stated in CCP principle. In this context, we try to enhance the cohesion concept by focusing more on the understanding of the package cohesion principles. This enhancement can lead to more accurate cohesion metrics.

The package cohesion principles describe a rich spectrum for cohesion. These principles provide a concept in which classes can be distributed based on different motivations for reusability and maintenance. The distribution that is based on the reusability motivation may differ from the ones of maintenance. These principles are always in tension. Martin argues that each principle appeals to a different group of people. People who care about reuse adhere to the Common Reuse Principle (CRP), and people who care about maintenance adhere to the Common Closure Principle (CCP).

While CRP tries to make packages small for reusability, CCP tries to enlarge packages to minimize the number of changed packages for maintainability. These two opposite forces as well as the needs of the software application itself need to be balanced. The balance changes over time along with reusability and maintenance needs. That means the packaging that is suitable in the present may not be suitable in the future [19].
Cohesion aims to minimize change propagation and maximize reuse. These two purposes are in inverse relationship. To minimize the change propagation, a package needs to include all the most likely affected classes, so the number of changed packages will be as few as possible. To maximize the reuse, a package needs to include only classes that are used together. Cohesion is composed of two parts or types, and these two parts can be defined from Martin's cohesion principles.

The first type of cohesion focuses on the common reuse concept. A package includes all the classes that are used together by the same user. When a user uses the package, it should use all the classes in the package. The focus is on how the package serves other packages, and how the classes cooperate to serve the using packages. This type of cohesion is based on the CRP principle.

The second type focuses on the common closure concept. All the classes in a package should have the same reasons for change, and if a change affects the package, it should affect all the classes in the package. This concept focuses on how the classes in a package use the same packages so they have the same reasons for change. This type of cohesion is based on the CCP principle.

Some existing metrics try to measure both parts of cohesion simultaneously. This may produce inaccurate values for cohesion. Some existing cohesion metrics could be classified with respect to the type of cohesion, which was emphasized during its development. Some metrics, which can not be so classified, were developed based on the idea of the number of relationships between classes of the package. Instead of measuring the common reuse or the common closure of the package, a connectivity metric focuses on the inner relationship between elements.
3.3 Cohesion Types

Common reuse focuses on the reusability of the package while common closure focuses on the package closure against the same kind of changes. How can we assess how well a package conforms to these design principles? A solution is to measure reuse with the REP and CRP principles by focusing on the incoming external dependencies because these dependencies determine the common reuse of the package classes. Similarly, to assess how the package conforms to the CCP principle, we focus on the outgoing external dependencies that influence change propagation and thus represent the common closure of the package.

Since the package design should maximize each type of cohesion, there is a need to propose a new metric for each type to assess whether its principle is satisfied. Two proposed metrics in this work, Common Reuse (CR) to assess the common reuse cohesion part and Common Closure (CC) to assess the common closure cohesion part focus only on the factors that help to measure the desired part.

3.3.1 Common Reuse (CR)

The CR metric tries to measure cohesion based only on the common reuse factors of the package. CR is defined based on the CRP principle [19], and it considers incoming dependencies from classes in other packages. This factor has been excluded in some cohesion metrics although it is important in measuring reuse coverage of the package. For example, Martin’s cohesion metric [19] is relational and disregards the external dependencies of a package, and therefore, we believe it does not accurately measure the common reuse of a package; see figure 3.1. The CR metric measures the common reuse of the package based on the external incoming dependencies of the package as well as the internal dependencies. Since external classes (or clients) may use the package’s classes (or servers), the incoming dependencies represent this usage.
cooperation between one package’s classes to serve the using packages means that they share the same responsibility in providing this service. So, they are cohesive in this task. The CR of the used package can be measured based on the dependencies of the using packages. The commonly used classes are cohesive in terms of use even though they may not have direct or indirect dependencies between them. CR can be explained by the extent to which all services provided by the package are related. For all incoming dependencies, the commonly used classes of the package represent the reuse coverage. We use a hubness measure, which will be defined next, to help us determine the use coverage of the package.

Hubness

Not all the elements of a package have the same degree of reachability. The degree of reachability is the number of classes in the package that can be reached directly or indirectly from a specific class in that package. We define the hubness of a class to be the local reachability of that class. In other words, hubness of class \( x \) is the set of classes in the same package that can be reached from \( x \) directly or indirectly. For example, in Fig 3.3, the hub set of \( C_3 \) is \( \{C_3, C_1, C_2, C_4\} \), and the hub set of \( C_1 \) is \( \{C_1\} \).

Hubness represents a class’s reachability to the other classes in the same package. We represent the graph of package \( P \) by \( G_P = \langle C, R \rangle \), where \( C \) represents the set of classes in package \( P \), and \( R \) is the set of direct relationships between classes of \( P \), i.e., \( R \subseteq C \times C \). Let \( c \in C \). Then, \( \text{Hubness}(c) = \{d \in C: \text{there is a path } c \rightarrow d\} \)

The proposed package Common Reuse metric, CR, is evaluated as follows:

\[ \text{Hubness} \] is the local reachability of a class to other classes within the package.
Let \( c \in C \), and suppose there is an incoming relation to \( c \) from a class in a different package. Then \( c \) is called an in-interface class. The cardinality of the intersection of the hub sets of all the in-interface classes in \( C \) divided by the number of classes in \( C \) is the Common Reuse of \( P \).

\[
CR = \frac{|\bigcap \text{in-interface class hub sets}|}{|C|}
\]

The numerator is the cardinality of the intersection of all hub sets of in-interface classes, and \(|C|\) is the number of classes in the package \( P \). The value of \( CR \) is between 0, no commonly used classes, and 1, all classes are always used together.

### 3.3.2 Common Closure (CC)

CC is developed based on the CCP principle [19] and it considers a package’s dependencies on other packages as well as the internal dependencies between classes of the package. The changes that will affect the package include changes in the packages that this package depends on, i.e., server packages. Therefore, CC includes outgoing package dependencies as an important factor in change closure measurement. The classes of the package should depend on the same set of packages, and thus, they will have the same reasons for change. As much as the package classes share the same server packages, so they will have the same change closure.

For this purpose, the global reachability\(^2\) for each class of the package is calculated. For global reachability for each class in the package, we find the set of all packages this class depends on directly or indirectly. In the global reachability for a class, we don’t include the set of

\[^2\text{The global reachability set is the set of packages that can be reached from a specific class in the package.}\]
classes in the package itself. We find the reachability set for each class in the package to find how many and how often classes in the same package face the same kind of changes. We count the set of depended-upon packages instead of the set of classes because as the REP implies if a class in the package is changed the whole package needs to be released.

The proposed package Common Closure metric, CC, is stated as follows:

Let system S be composed of a set of packages, \{P1, P2, ..., Pn\}, and for package P, \(G_p = \langle C, R \rangle\). For each class \(c \in C\), define the reachable set of packages in S that \(c\) depends on directly or indirectly. The reachable set for \(c\) involves two types of dependencies. We say there is a direct dependency from package \(Q_1\) to package \(Q_2\) if there is a direct dependency from a class in \(Q_1\) to a class in \(Q_2\). We say there is a dependency path from \(Q_1\) to \(Q_m\) if there are packages \(Q_2, ..., Q_m-1\) such that there is a direct dependency from \(Q_j\) to \(Q_{j+1}\) for \(1 \leq j \leq m-1\). Please note that the classes in, for example, \(Q_k\), that make the direct dependencies from \(Q_{k-1}\) and to \(Q_{k+1}\) do not need to be the same classes, and, in fact, they may not be directly or even indirectly related.

For \(c \in C\), \(Q\) is in the reachable set of \(c\) if there is a \(d \in C\) with \(d \in \text{Hubness}(c)\), and there is a direct dependency from \(d\) to a class in a package \(Q^*\) (where \(Q^*\) is not \(P\)) and there is a dependency path from \(Q^*\) to \(Q\). The cardinality of the intersection of these reachable sets divided by the cardinality of the union of these sets represents the Common Closure of \(P\).

\[
CC = \left( \frac{|\cap \text{Reachable Package sets}|}{|\cup \text{Reachable Package sets}|} \right)
\]

Where the numerator is the cardinality of the intersection of the reachable sets for all classes in \(C\), and the denominator is the cardinality of the union of these sets. The value of \(CC\) is between 0, if there is no intersection between the reachable sets, and 1, if all the package’s classes depend on the same set of packages.
3.4 Working Example

This section provides the working example that shows how the two proposed metrics are calculated. The same example is used for both metrics that give different values, which demonstrates the need of cohesion classification.

![Diagram of the working example]

**Figure 3.3 The working example**

To measure the CR, hubness for in-interface classes is calculated first. In-interface classes are those that have direct incoming external dependencies (dependencies from client packages).

<table>
<thead>
<tr>
<th>Class</th>
<th>Hub set</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>{C1}</td>
</tr>
<tr>
<td>C2</td>
<td>{C1, C2}</td>
</tr>
<tr>
<td>C3</td>
<td>{C3, C4, C2, C1}</td>
</tr>
</tbody>
</table>
The table above presents the hub set for the participating classes (in-interface) in the package P1, which are C1, C2, and C3 in this example. The CR value is:

\[ |\cap \text{in-interface hub sets}| = |\{C1\}| = 1 \]

\[ CR = 1/4 \]

For the working example in figure 3.3, the CC value of P1 is calculated as follows:

The global reachability for each class:

<table>
<thead>
<tr>
<th>Class</th>
<th>Global Reachability</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>{P2, P3, P4}</td>
</tr>
<tr>
<td>C2</td>
<td>{P2, P3, P4}</td>
</tr>
<tr>
<td>C3</td>
<td>{P2, P3, P4, P5}</td>
</tr>
<tr>
<td>C4</td>
<td>{P2, P3, P4}</td>
</tr>
</tbody>
</table>

\[ \cap \text{Reachable Package sets} = \{P2, P3, P4\} \]

\[ \cup \text{Reachable Package sets} = \{P2, P3, P4, P5\} \]

\[ CC = \frac{3}{4} \]

The two different values of CR and CC illustrate the distinction between the cohesion types proposed earlier in this paper.

However, Martin’s cohesion metric, which measures the package cohesion based on the number of internal relations of the package, returns the same value for each package; and therefore, it cannot differentiate the new cohesion types.

### 3.5 What Is Good in This Metric Suite?

The proposed metric suite has the following strengths.

1. The metric suite relies on and has been developed based on the package cohesion principles [19] that are well known and well accepted in the field.
2. The suite considers the in-coming and out-going dependencies of the package as well as the intra-dependencies between classes in the package.

3. The metrics don’t depend only on the number of dependencies. They depend on the influence the dependency may have. A relationship between two classes is not the only reason to keep them in the same package.

4. Each metric of the suite, \( CR \) and \( CC \), focuses on an important type (part) of package cohesion. The two metric values could be combined to form an overall value of the package cohesion while still recognizing the two types.

5. The two proposed cohesion types can be used to classify the existing cohesion metrics. This classification will help to understand the different interpretations of cohesion.

6. This suite can be used to classify system packages, into \( CR \) or \( CC \) packages, based on the \( CR \)'s and \( CC \)'s values. This classification may help to discover which of these types has dominated or has been considered more in a specific version of the software system.

7. Both \( CR \) and \( CC \) treat cohesion based on direct and indirect relationships. The indirect relationships between classes are taken into account in both metrics. Many existing cohesion metrics do not consider the indirect dependencies.

8. The proposed metrics can measure cohesion during the design phase of software development.

The proposed cohesion metrics have flexibility, which is a new feature that has not been considered before. This feature is discussed in the next section.
3.6 The Flexibility of The Proposed Metrics

The proposed metric suite has a degree of flexibility that allows the subject-matter expert to create their own customized metric for both CR and CC. It is worth noting that the default metrics also provide very good results for the general cases.

In real life, the value of 1 for the CR is hard to attain, at least for the most of the packages, because it is hard to make all the using packages use the same set of classes and do this for all the packages in the system. Moreover, the CRP principle has been built on the concept of common reuse instead of 100% complete reuse. For these reasons, an expert can adjust the degree of commonness that is desirable for a specific situation. The CR metric can be customized to the desired degree of commonality. One suggestion is to find the classes that appear in more than 50% of the hub sets.

For example, if there are five hub sets, we find the classes that appear in three or more sets instead of finding those appear in all of the sets. For the working example, the new calculations will be as following:

The in-interface classes are: C1, C2, and C3

\[ |\cap_{\text{in-interface hub sets}}| = |\{C1, C2\}| = 2 \]

\[ CR = \frac{2}{4} = \frac{1}{2} \]

Similarly for the CC metric, it is hard to make all the classes in the package depend on the same set of packages and do this for all the packages in the system. Moreover, the CCP principle has been built on the idea of common closure instead of 100% complete closure. For these reasons, an expert can customize the CC metric by fixing the degree of intersection of the reachable sets based on the required cohesion for the situation in hand. One suggestion is to find the packages that appear in more than 50% of the reachable sets. For example, if there are five...
reachable sets, we find the packages that appear in three or more sets instead of finding those appear in all of the sets. If we apply this on the working example, the new calculations for CC will be:

<table>
<thead>
<tr>
<th>Class</th>
<th>Global Reachability</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>{P2, P3, P4}</td>
</tr>
<tr>
<td>C2</td>
<td>{P2, P3, P4, P5}</td>
</tr>
<tr>
<td>C3</td>
<td>{P2, P3, P4, P5}</td>
</tr>
<tr>
<td>C4</td>
<td>{P2, P3, P4, P5}</td>
</tr>
</tbody>
</table>

\[
\text{\cap \text{Reachable Package sets}} = \{P2, P3, P4, P5\}
\]

\[
\text{\cup \text{Reachable Package sets}} = \{P2, P3, P4, P5\}
\]

\[
CC = \frac{4}{4} = 1
\]

### 3.7 The Unified Metric

After each type of cohesion has been measured by itself, the two values of CR and CC may be combined to one unified value of package cohesion. Each metric of the suite, CR and CC, focuses on an important type (part) of package cohesion. The two metric values are combined to form an overall value of the package cohesion while still recognizing the two types.

The CR and CC values are presented in the Cartesian coordinate system of two dimensions, where CR is in the X dimension and CC is in the Y dimension, figure 3.4. Each value of CR or CC will range between 0 and 1, where 0 is the minimal and 1 is the maximal. The maximal values will represent the optimal (position) case (1,1) of package cohesion. In the opposite, the minimal values will represent the worst (position) case (0,0) of package cohesion. The combined cohesion \(CH\) is defined as follow:
\[ CH = \frac{\sqrt{2} - D}{\sqrt{2}} \]

Where \( D \) is the distance between the optimal (position) case and the position of the package, which is represented by \( CR \) and \( CC \) values, and it is given by:

\[ D = \sqrt{(1 - CR)^2 + (1 - CC)^2} \]

The maximum value of \( D \) is the maximal distance between two \((x,y)\) coordinates in the Cartesian coordinate system, Figure 3.4. In other words, it is the distance between the optimal \((1,1)\) and the worst \((0,0)\) coordinates, which is \( \sqrt{2} \).

![Figure 3.4 The Cartesian coordinate system of CR and CC](image)

The value of \( CH \) is between 0, if the position is in the worst case \((0,0)\) where both \( CR \) and \( CC \) are minimal, and 1, if the position is in the optimal case \((1,1)\) where both \( CR \) and \( CC \) are maximal. For example if \( CR=0.7 \) and \( CC=0.4 \), these values represent the position of the point \((0.7,0.4)\) in the Cartesian coordinate system. The \( CH \) value is then calculated based on the distance between this point and the optimal position \((1,1)\). \( CH \) in this case is 0.53.

### 3.8 Theoretical Validation

For any proposed metric to be accepted and used, it should be validated. In this section, theoretical validation of the proposed package cohesion metric suite is provided using the four
cohesion properties given by Briand et al [99]. There are different complexity validation frameworks in the literature, such as [100]. However, the Briand framework has been chosen because it has cohesion properties.

3.8.1 Non negativity and Normalization:

According to the definition of the CR cohesion metric, the value of CR belongs to the non-negative unit interval, \( CR \in [0,1] \).

Similarly, according to the definition of the CC cohesion metric, the value of CC belongs to the non-negative unit interval, \( CC \in [0,1] \). Hence, the cohesion values resulting from the proposed metrics are always non-negative.

The CR is normalized by the number of classes in C. Also, the CC is normalized by the union of all the reachable sets of classes in the package.

Accordingly, the combined cohesion \( CH \) relies on the fact of the distance between two points in the Cartesian coordinate system, which is always non-negative. Also, \( CH \) is normalized by the maximum distance can be obtained based on the CR and CC values.

3.8.2 Null value and maximum value:

With respect to [99], we are only measuring cohesion of a module, i.e., our packages are the modules in [99].

The CR cohesion of P= \(<C, R>\) is zero if there is more than one in-interface class in P, and there are no relationships between the in-interface classes. Our CC metric does not fit the model of [99] because [99] assumes that everything which can influence the cohesion of a package is within the package. Our paper, in fact, shows that relationships between classes in a
single package may be established by what happens outside the package. This possibility does not seem to be recognized in [99].

3.8.3 Monotonicity:

Monotonicity is clearly preserved for both CR and CC because the addition of more relationships within a package increases both the local reachability, i.e., the size of hub sets, and the global reachability of each class in P with respect to the union of all globally reachable sets in P. Accordingly, the increase in the value of one or both CR and CC will increase the CH value.

3.8.4 Merging of Unconnected Packages:

Cohesion property 4 from [99] is clearly satisfied by CR because when two unconnected packages, P1 and P2, are merged to form the new package P, the new package will have more classes than either P1 or P2 (assuming neither P1 nor P2 is empty), but the intersection of the in-interface hub sets will, in fact, be empty if both P1 and P2 have in-interface classes. (The claim about the empty intersection holds because the packages P1 and P2 are not connected.)

The CC metric for P will not be more than the CC metric evaluated at P1 or P2 because potentially more global reachability sets will be intersected in the numerator of CC of P but since P1 and P2 are not connected then the size of the union of global reachability set will not decrease. Thus, CC of P must be less than or equal to the maximum of CC of P1 and CC of P2. Accordingly, CH must be less than or equal to the maximum of P1 and P2. Thus, the cohesion property 4 of [99] holds.

3.9 Applications of The new Metric

The purpose of developing new cohesion metrics is to enhance the accuracy of package cohesion measures that in turn help in predicting software maintainability and software
testability as early as the design phase of software development. Predicting software maintenance effort and software testing effort at early stages can reduce the cost of future software maintenance and help developers to be aware of the weak and poor designed software packages that may raise the maintenance and testing costs and efforts. In the next chapters, using the proposed package cohesion metric, we try to predict software maintenance effort and software testing effort.

Chapters 4 and 5 present two empirical studies of the role of the new package cohesion metric in maintenance effort and testing effort predictions. The results show that the new metric provides good predictions and that might be because of the theory behind it, since it has been developed based on the package cohesion principles [19], well-known and accepted principles in the field.
CHAPTER 4

Maintainability Prediction Using Package Cohesion

4.1 Introduction

Software maintainability refers to the ease of maintaining software products in order to prevent or correct defects and their causes, and to respond to new requirements and environmental changes [110]. The quality of the software design has a considerable impact on software maintainability, which makes design level metrics potentially beneficial predictors for software maintenance [116]. Predicting software maintainability during the software design phase can reduce much of the maintenance costs and efforts. One application of the proposed cohesion metric in this research is to predict needed software maintenance efforts as early as the software design phase because early prediction can lead to substantial cost reductions as well as to software maintainability improvements. While a number of research studies performed to predict software maintainability were based on measures taken after the coding phase, the cohesion metric we developed has an advantage of measuring cohesion in an earlier phase, the design phase. Another advantage of this metric is that it has been developed based on well-known and well-accepted package cohesion principles [19]. Further, if there is a relationship between our metric and software maintainability, then we will potentially establish a relationship between these principles and software maintainability.

At the present, there is no one single predictive measure for software maintainability [115]. However, cohesion as one factor, which is measured by the proposed cohesion metric (CH), in this research study, can be counted as a predictor for software maintainability.

This chapter investigates the relationship between package cohesion, using the proposed metric CH, and software maintenance efforts. For this purpose, the package cohesion metric has
been developed, based on a solid theory of the package design principles [19] to predict software maintainability. A number of experiments and statistical analyses have been designed and performed to investigate this relationship.

### 4.2 Internal and External Attributes

Software product quality is divided into internal and external attributes [32], see section 2.6. External attributes cannot be measured based only on software artifacts [27][133]. That means, for example, that the maintainability of the package depends on many factors such as software environment, package design, experience of the maintenance team, and the age of the system [133]. An older software system is likely to require more maintenance efforts because its size is likely to increase, which may make it more likely to be less organized and less understandable [133]. The external attributes of the package, e.g., maintainability, cannot be known before it is measured, i.e., it has been maintained. However, the external attributes can be predicted by using internal attributes as indicators for the external ones.

In this empirical study, we consider package cohesion as the internal attribute to estimate an external attribute, namely package maintainability, using probabilities and probabilistic models. This approach has been utilized in research studies such as Morasca [152], Al Dallal [133], and Li and Henry [117]. There are two advantages to using a probabilistic approach [152]. First, it allows us to use a theoretically well-defined and well-studied mathematical concept. Second, from the application point of view, probabilities are practically and commonly used.

We use a package cohesion internal attribute as an independent variable to predict the maintainability of the package. This prediction process requires data collection of maintenance from the actual software histories. We use two measures. The first measure is the number of revisions (#Revisions), which is the number of times that a package has been involved in
maintenance activities. The second measure is the number of revised lines of code (RLOC), i.e., added, deleted, or modified, [133] to indicate the package maintainability.

On the other hand, the internal attribute of the package, i.e., cohesion, can be measured based on the package design itself before it has been maintained. The package cohesion metric is useful when it is used to predict package maintainability [152]. In this empirical study, package cohesion will be measured using the metric defined in Chapter 3 of this research.

4.3 Related Work

In this section, we present some related work on the measurement of software maintainability. Most of the existing research works provide maintainability indicators on a class level. Some of these research works theoretically or empirically studied the relationships between internal attributes, e.g., cohesion, and maintainability.

Researchers and practitioners proposed several software metrics to evaluate internal quality properties, such as cohesion. A number of cohesion metrics have been proposed on a class level in object-oriented systems, see section 2.8.

Many researchers and practitioners proposed software metrics in relation to software maintainability and its characteristics. While some of them were theoretically validated, only a few were empirically validated. Several research studies were conducted to investigate the relationship between class-level cohesion and software maintainability. One of the early investigation studies was by Li and Henry [117] to investigate the validity of object-oriented metrics in predicting software maintenance efforts. The study tested if there is a strong relationship between object-oriented software metrics and maintenance efforts. LCOM, a cohesion metric developed by Chidamber and Kemerer [70], was among ten software metrics that were investigated. The results of the statistical analysis performed on two software systems
showed that there is a strong relationship between the studied software metrics and maintenance efforts. Briand et al. [60] proposed cohesion and coupling measures based on object-oriented design principles to evaluate software maintainability. However, this approach was not validated. Briand et al. [61] defined a ratio-scale metric for cohesion to predict the error-proneness in the software design. The results of the experiments proved that software metrics can predict software error-proneness. Dagpinar and Jahnke [114] provided empirical evidence that software metrics can effectively be used to predict software maintainability. However, they found that Bieman and Kang’s Loose Class Cohesion (LCC) [71], metric was not a significant predictor for class maintainability. Basili et al. [33] were concerned about fault detection and the fault prone-ness part of maintenance. They showed by their experiments’ results that the Chidamber and Kemerer’s metrics [70] are, individually, good indicators for faulty modules. This was supported by Gyimothy et al. [122] where a validation of the ability of the LCOM metric as a good indicator of software fault-proneness was indicated. The study was conducted on open source software, Mozilla. Koru et al. [126] showed that there is a correlation between the number of bugs and size. Chaumun et al. [131] proposed a change impact model to assess changeability, which is one important aspect of maintainability. Based on the assumption that high-level design has an impact on maintainability, they assess the system changeability to compute the impact of changes made to system classes. The results of the experiments showed that there is a weak correlation between WMC design metric, Weighted Method per Class that is one of Chidamber and Kemerer’s six design metrics [70], and the mean change impact. The higher the WMC value is, the higher is the mean change impact. Following the same approach, Kabaili et al. [127] investigated the relation between the cohesion metrics, LCC and LCOM, and the changeability. The Pearson correlation coefficient was used to measure the relationship. They concluded that
LCC and LCOM are not good indicators for change and cannot be trusted. Gui and Scott [67][128] developed a new cohesion measure to assess the reusability of Java components. The purpose of the study was to predict how much effort was needed to reuse a component within a larger system. The components were assigned to experts and the number of lines of code needed to adapt a component in the new environment was counted as the reusability measure. The more lines required, the lower the reusability. They demonstrated by an empirical study that the proposed metric is a very good predictor of the number of line of code required to make modifications to the reused components. Atole and Kale [129] measured the software quality by applying Martin’s design metrics [19], where cohesion metric H is one of them, on a software design example. They concluded that the design metric has the ability to evaluate the software design quality. Badri et al. [130] aimed at empirically investigating the relationship between the lack of cohesion and the testability on a class level. The results of the empirical study of the two systems proved the relationship between (the lack of) cohesion of classes and testability. Rizvi and Khan [132] proposed a maintainability estimation model in object-oriented software in the design phase to estimate the maintainability of a class diagram in terms of their understandability and modifiability. The results of the empirical study found that class diagram maintainability strongly correlated to understandability and modifiability. Al Dallal [133] empirically investigated the relationship between a number of internal class quality attributes (size, cohesion, and coupling) and class maintainability. Prediction models, based on statistical techniques, were constructed and validated to estimate the class maintainability. The results showed that internal attributes (size, cohesion, and coupling) have an impact on class maintainability. The higher the cohesion is, the higher the class maintainability is.
Looking carefully to the existing studies, some studies were conducted using a cohesion metric on the class level. Others were not validated or only validated theoretically without any empirical validation of the relationship with software maintenance. Some studies [124] used a subjective expert’s surveys to measure the metric’s prediction. Some related experimental studies [118-123] were performed to investigate the ability of software metrics on a class level in predicting some aspects of software maintenance, such as defect density or fault-proneness, but they don’t consider other types of maintenance, such as adaptive maintenance. Some studies [127][131] did not rely on the reported maintenance history of the studied software systems. The drawback in such studies is that the maintenance data collected for the experimental studies does not represent the actual maintenance data. Some studies, such as [117][114], have relatively a small sample size of the experimental study, which makes the results hard to be generalized. Some studies, such as [117], did not investigate the ability of software measures in predicting software maintainability [133]. In other words, the study did not conclude if there is an impact of the internal attribute on the external one. Other studies [132][134] were limited to size and complexity attributes, and did not consider other important ones such as coupling and cohesion.

In contrast, we found that our study is unique in several different ways. It proposes a cohesion metric on a package level based on the well-known package cohesion principles, both theoretically and experimentally validated, uses actual maintenance data history of software, uses objective data instead of subjective ones, and considers all types of maintenance activities. To the best of our knowledge, there is no study that has investigated the relationship between package level cohesion and software maintainability, which makes this research original and vital in this matter.
This study is set to investigate the relationship between the newly developed measure of package cohesion (CH) and software maintainability. In general, the expectation is that a highly cohesive software package requires less effort to maintain. Thus, a set of analyses examines the relationship between software cohesion as measured by the proposed package cohesion metric and package maintainability. The analysis is repeated for each one of the four data sets provided by the four systems, Camel, Tomcat, JHotDraw, and JEdit. The four data sets have similar characteristics and the need to have a larger sample size resulted in a regression analysis being run on the integrated data from the four systems. Regression analysis includes package size (#Classes) as part of the prediction model to control for its effect and improve model prediction.

4.4 Descriptive Statistics

This empirical study is based on four open-source java software systems used to investigate the role of package cohesion measure in predicting software maintainability. This section provides descriptions about the studied software systems and the maintenance data collection. Two package cohesion metrics are included in this study: Martin’s cohesion metric (H) and our package cohesion metric (CH), proposed in Chapter 3, which was developed based on Martin’s package cohesion principles [19].

4.4.1 The software systems

Four open-source Java software systems were involved in the empirical study. All the four systems were selected based on the following criteria to allow results’ generality; they had: (1) to be implemented using the Java programming language, (2) to have maintenance repositories available, namely Apache Subversion (SVN), (3) to have sufficient number of versions for each system that have been maintained, (4) to be organized using packages, (5) to
have different sizes ranging from very large to small systems in terms of number of packages and number of classes, (6) to be from different domains, and (7) to have positive reviews and to be mature. We expect these criteria will allow the generalization of the results obtained from the study. The first system, Camel [153], is a rule-based and mediation engine to configure routing and mediation rules. The second system, Tomcat [154], is an open source webserver developed to implement Javaservlet and Java Server pages (JSP). Apache Tomcat is developed by the Apache Software Foundation. It has been developed and released under Apache License version 2. The third system, JHotDraw [155], is a Java GUI framework for technical and structured graphics. The fourth system, JEdit [156], is an open source Java text editor for programmers. It is licensed by GPL General Public License version 2.0. Table 4.1 provides details of the maintenance history; and Table 4.2 provides details about the studied systems.

Table 4.1 Maintenance history details

<table>
<thead>
<tr>
<th></th>
<th>Base Release</th>
<th>End Release</th>
<th>History Studied</th>
</tr>
</thead>
<tbody>
<tr>
<td>Camel</td>
<td>2.0.0</td>
<td>2.2.0</td>
<td>Aug/24/09 – Feb/6/10</td>
</tr>
<tr>
<td>Tomcat</td>
<td>7.0.6</td>
<td>7.0.22</td>
<td>Jan/14/11 – Oct/1/11</td>
</tr>
<tr>
<td>JHotDraw</td>
<td>7.5</td>
<td>7.6</td>
<td>July/29/10 – Jan/9/11</td>
</tr>
<tr>
<td>JEdit</td>
<td>4.5.0</td>
<td>5.1.0</td>
<td>Jan/31/12 – July/28/13</td>
</tr>
</tbody>
</table>

Table 4.2 Details of the studied systems

<table>
<thead>
<tr>
<th></th>
<th>#LOC</th>
<th>#Methods</th>
<th>#Classes</th>
<th>#Packages</th>
<th>#Revised-Packages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Camel</td>
<td>143732</td>
<td>17369</td>
<td>5111</td>
<td>264</td>
<td>179</td>
</tr>
<tr>
<td>Tomcat</td>
<td>170461</td>
<td>15372</td>
<td>1725</td>
<td>113</td>
<td>62</td>
</tr>
<tr>
<td>JHotDraw</td>
<td>77194</td>
<td>7122</td>
<td>1026</td>
<td>65</td>
<td>65</td>
</tr>
<tr>
<td>JEdit</td>
<td>111861</td>
<td>7386</td>
<td>1238</td>
<td>35</td>
<td>23</td>
</tr>
</tbody>
</table>
4.4.2 Maintenance Data

The source of the maintenance data for this study is the Version Control System (VCS), subversion (SVN), which is publicly available. The public can view the history of maintenance activities that have been made on the software system using an SVN client. Each log entry in the repository log has a revision number, date and time, and short message that explains the maintenance activity. We considered all types of maintenance activities: perfective, adaptive, corrective, and preventive. We don’t differentiate between different maintenance activities. Figure 4.1 shows a repository log example of maintenance history. The upper part shows the revision details, while the bottom part shows the affected paths for the selected revision.

![Repository log example](image)

Figure 4.1 Repository log example

For this empirical study, as suggested by Al Dallal [133], we considered two package maintenance measures: the number of revisions (#Revisions) in which the package has been involved, and the number of revised lines of code (RLOC) during the studied maintenance history. The number of revised lines of code RLOC is calculated as suggested by Li and Henry.
[117], where a line added or deleted is considered one revised line, and a line modified is considered two revised lines, one deletion and one addition. We consider these two measures for two reasons. First, the number of revisions refers to the maintenance rate, while the number of RLOC is found to be correlated with maintenance cost [158][133] and maintenance effort measured in unit of time [159][133]. Packages with lower maintenance rates are better than those with higher rates because the code with more revisions becomes less organized, less understandable, and more fault-prone [109][133]. Second, these two measures are measurable using the freely available software maintenance history [133].

To collect maintenance data, we used the free software tool, TortoiseSVN [102], which is a subversion client developed to access the subversion (SVN) repositories. For each software system, the log of the SVN repository includes the following revision information: revision number, revision description, all the packages and classes affected by the revision, the previous and the current class versions, and the number of lines added, deleted, or modified. We had to create a list of all the packages and the classes within the package to relate each revision’s information to the appropriate package. Then, revisions and revised lines of code were collected on package level. We considered different versions for each system, and collected the maintenance data reported during the entire maintenance period. Table 4.3 summarizes maintenance data for each system.

<table>
<thead>
<tr>
<th>System</th>
<th>#Revisions</th>
<th>Mean #Revisions</th>
<th>#RLOC</th>
<th>Mean #RLOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Camel</td>
<td>1614</td>
<td>6.11</td>
<td>60688</td>
<td>229.87</td>
</tr>
<tr>
<td>Tomcat</td>
<td>636</td>
<td>5.63</td>
<td>22027</td>
<td>194.93</td>
</tr>
<tr>
<td>JHotDraw</td>
<td>354</td>
<td>5.45</td>
<td>21857</td>
<td>336.26</td>
</tr>
<tr>
<td>JEdit</td>
<td>323</td>
<td>9.23</td>
<td>9981</td>
<td>285.17</td>
</tr>
</tbody>
</table>
Two computer science PhD students were dedicated to collecting the maintenance data. The data was collected manually from the maintenance repositories. We have randomly checked the validity of the data collected. This process increased our confidence about the validity of the data collected.

For the purpose of a system’s list of classes and list of packages, we have used the JHawk tool [157]. Then, each revision reported in the maintenance history was specified to the associated class along with the number of revised lines of code RLOC. Finally, maintenance data was collected on the package level.

4.4.3 Package Cohesion Data

Package cohesion data was gathered from two package cohesion metrics. The first metric is our proposed package cohesion metric in Chapter 3, CH. The second metric is Martin’s cohesion metric, H. These two metrics have been used to investigate the correlation between package cohesion and maintainability. For the purpose of data gathering, we have developed our Java tool to measure the CH package cohesion metric. The tool has been extended to calculate Martin’s package cohesion metric, H. For each system, a list of all the packages, the number of classes in each package, and the associated cohesion values were generated.

4.5 Exploring the relationship between package cohesion and maintainability using correlation

In this section, we present the correlation part of the empirical study we performed to explore the relationship between package cohesion and maintainability. Correlation is a widely used statistical test to investigate the relationship between internal quality attributes and external quality attributes; and it has been used in different studies such as [125][130][140][150][160].
The correlation analysis aims to determine whether each individual package cohesion metric (CH and H) is significantly related to the maintenance measures (#Revisions and RLOC) of the package. For this purpose, we have performed Spearman’s rank correlation due to the non-parametric nature of the metrics’ data. We have used the well-known SPSS software for the correlation analysis of the empirical study. We have created and analyzed a correlation matrix for each software system in the study. Each correlation matrix has all the studied variables (cohesion and maintenance), a correlation coefficient (r), and significance level. For each pair of variables, r value can range between -1 and +1, where 1 represents a perfect positive correlation between the pair variables; -1 denotes a perfect negative correlation; and 0 indicates that there is no relationship between the variables. The magnitude of the coefficient determines the degree of the correlation. The ratings of correlation strength follow the adjectives developed by Cohen [161]:

<table>
<thead>
<tr>
<th>Correlation coefficient</th>
<th>Adjective rating</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; 0.1</td>
<td>Trivial</td>
</tr>
<tr>
<td>0.1 to 0.3</td>
<td>Minor</td>
</tr>
<tr>
<td>0.3 to 0.5</td>
<td>Moderate</td>
</tr>
<tr>
<td>0.5 to 0.7</td>
<td>Large</td>
</tr>
<tr>
<td>0.7 to 0.9</td>
<td>Very large</td>
</tr>
<tr>
<td>0.9 to 1</td>
<td>Almost perfect</td>
</tr>
</tbody>
</table>

Besides the strength of the correlation, the relationship between any pair of variables should be assessed for its significance as well. The significance is assessed by the p-value, which corresponds to the probability that the found correlation might be due to purely random effects. The smaller the p-level, the more significant is the relationship between variables [162]. The significance of the correlation in this empirical study was tested at a 95% confidence level (i.e.,
While the correlation can establish the relationship, it cannot establish a cause-effect relationship between the pair variables [162].

4.5.1 Hypotheses

Our objective is to assess to what extent is the package cohesion metric related to the maintenance effort of the software packages. The hypotheses of the empirical study are:

\( H_01: \) There is no significant correlation between package cohesion, CH, and the number of Revisions, #Revisions.

\( H_02: \) There is no significant correlation between package cohesion, CH, and the number of revised lines of code, RLOC.

\( H_03: \) There is no significant correlation between Martin’s package cohesion, H, and the number of Revisions, #Revisions.

\( H_04: \) There is no significant correlation between Martin’s package cohesion, H, and the number of revised lines of code, RLOC.

In this experiment, rejecting the null hypothesis indicates that there is a statistically significant relationship between the pair of variables (significance level \( \alpha = 0.05 \)).

4.5.2 Statistical Analysis

The number of software revisions (#Revisions) and the number of revised lines of code (RLOC) on the package during the maintenance history assess software package maintainability. A lower number of package revisions and a smaller number of revised lines of code during the package maintenance history indicates less effort needed to maintain the software and thus, indicate high maintainability.
Data screening and evaluation of linearity for assessing correlations and regression analysis led to the natural log transformation of the variables number of revisions (#Revisions), package size (#Classes), and number of revised lines of code on the package during the maintenance history (RLOC). We included Martin’s package cohesion metric (H) in the list of variables for the purpose of comparison.

Table 4.4 provides descriptive statistics (mean and standard deviation) for the variables used in analyzing software maintainability across the four systems, Camel, Tomcat, JHotDraw, and JEdit.

Table 4.4 Means and Standard Deviations of the variables used in the maintainability analysis

<table>
<thead>
<tr>
<th>Variable</th>
<th>Camel N=264</th>
<th>Tomcat N=113</th>
<th>JHotDraw N=65</th>
<th>JEdit N=35</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mean</td>
<td>S.D.</td>
<td>Mean</td>
<td>S.D.</td>
</tr>
<tr>
<td>H</td>
<td>.636</td>
<td>.361</td>
<td>.817</td>
<td>.524</td>
</tr>
<tr>
<td>CH</td>
<td>.530</td>
<td>.388</td>
<td>.358</td>
<td>.374</td>
</tr>
<tr>
<td>#Classes</td>
<td>13.700</td>
<td>29.637</td>
<td>16.17</td>
<td>23.063</td>
</tr>
<tr>
<td>LogClasses</td>
<td>.772</td>
<td>.518</td>
<td>.968</td>
<td>.478</td>
</tr>
<tr>
<td>Log#Revi</td>
<td>.499</td>
<td>.483</td>
<td>.448</td>
<td>.498</td>
</tr>
<tr>
<td>RLOC</td>
<td>229.879</td>
<td>732.318</td>
<td>194.69</td>
<td>511.186</td>
</tr>
<tr>
<td>LogRLOC</td>
<td>1.213</td>
<td>1.073</td>
<td>1.135</td>
<td>1.125</td>
</tr>
</tbody>
</table>

4.5.3 Results and Discussion

A Spearman Rho correlation is the appropriate 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 proposed measure of package cohesion CH, the Martin’s package cohesion metric H, package size (#Classes), and the two measures of package maintainability, the number of package
revisions (#Revisions) and the number of revised lines of code (RLOC), within each of the four data sets. Table 4.5 provides the list of these correlations for the four sets of data.

<table>
<thead>
<tr>
<th>Data Set</th>
<th>Data Set</th>
<th>CH</th>
<th>H</th>
<th>CH</th>
<th>#Classes</th>
<th>#Revisions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Camel</td>
<td>CH</td>
<td>.281**</td>
<td>- .350**</td>
<td>- .655**</td>
<td>.720**</td>
<td>.962**</td>
</tr>
<tr>
<td>N=264</td>
<td>#Classes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>#Revisions</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>RLOC</td>
<td></td>
<td></td>
<td>- .129*</td>
<td>- .533**</td>
<td>.702**</td>
</tr>
<tr>
<td>Tomcat</td>
<td>CH</td>
<td>.169</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N=113</td>
<td>#Classes</td>
<td></td>
<td></td>
<td>- .069</td>
<td>- .736**</td>
<td></td>
</tr>
<tr>
<td></td>
<td>#Revisions</td>
<td></td>
<td></td>
<td>- .010</td>
<td>- .545**</td>
<td>.686**</td>
</tr>
<tr>
<td></td>
<td>RLOC</td>
<td></td>
<td></td>
<td>- .007</td>
<td>- .521**</td>
<td>.663**</td>
</tr>
<tr>
<td>JHotDraw</td>
<td>CH</td>
<td>.157</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N=65</td>
<td>#Classes</td>
<td></td>
<td></td>
<td>- .041</td>
<td>- .706**</td>
<td></td>
</tr>
<tr>
<td></td>
<td>#Revisions</td>
<td></td>
<td></td>
<td>- .07</td>
<td>- .594**</td>
<td>.674**</td>
</tr>
<tr>
<td></td>
<td>RLOC</td>
<td></td>
<td></td>
<td>.098</td>
<td>- .631**</td>
<td>.769**</td>
</tr>
<tr>
<td>JEdit</td>
<td>CH</td>
<td>.468**</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N=35</td>
<td>#Classes</td>
<td></td>
<td></td>
<td>- .205</td>
<td>- .709**</td>
<td></td>
</tr>
<tr>
<td></td>
<td>#Revisions</td>
<td></td>
<td></td>
<td>- .024</td>
<td>- .650**</td>
<td>.754**</td>
</tr>
<tr>
<td></td>
<td>RLOC</td>
<td></td>
<td></td>
<td>- .008</td>
<td>- .623**</td>
<td>.711**</td>
</tr>
</tbody>
</table>

** Correlation is significant at the .001 level
* Correlation is significant at the .05 level

Table 4.5 reveals that the new proposed measure of package cohesion, CH, consistently has a negative large correlation with the two measures of package maintainability, number of package revisions (#Revisions) and the number of revised lines of code (RLOC), across all the four data sets. The correlation values between package cohesion CH and number of revisions (#Revisions) across the four data sets range from -0.545 (for the Tomcat system data set) to -0.650 (for the JEdit system data set). Similarly, the correlation values between package cohesion CH and the number of revised lines of code (RLOC) across the four data sets ranges from -0.521 (For the Tomcat system data set) to -0.631 (for the JHotDraw system data set). The statistically significant correlations confirm that the expectation of a highly cohesive software package
requires less effort to maintain. That is high values of the proposed measure of package cohesion are associated with a lower number of its revisions and a lower number of revised lines of code.

In this study, the correlations between Martin’s package cohesion metric H and the two package maintainability measures, number of package revisions (#Revisions), and the number of revised lines of code (RLOC) are not as strong as the ones with the newly proposed measure of package cohesion CH. These correlations are consistently weak and statistically insignificant across all the four data sets, except for the correlation with the revised lines of code (RLOC) for the Camel system’s data. The value of the correlation is -.129, which relatively small yet statistically significant at an .05 level. The significance of the weak correlation might be justified by the large sample size of the Camel system data set. The correlation values between Martin’s package cohesion H and number of revisions (#Revisions) across the four data sets range from -0.010 (for the Tomcat system data set) to -0.101 (for the Camel system data set). Similarly, correlation values between Martin’s package cohesion H and the number of revised lines of code (RLOC) across the four data sets ranges from -0.007 (for the Tomcat system data set) to -0.129 (for the Camel system data set).

Table 4.6 summarizes the results of the examined null hypotheses. In this experiment, rejecting the null hypothesis indicates that there is a statistically significant relationship between the pair of variables (significance level \( \alpha = 0.05 \)).

<table>
<thead>
<tr>
<th>Null Hypothesis</th>
<th>Camel</th>
<th>Tomcat</th>
<th>JHotDraw</th>
<th>JEdit</th>
</tr>
</thead>
<tbody>
<tr>
<td>H_{01}</td>
<td>Rejected</td>
<td>Rejected</td>
<td>Rejected</td>
<td>Rejected</td>
</tr>
<tr>
<td>H_{02}</td>
<td>Rejected</td>
<td>Rejected</td>
<td>Rejected</td>
<td>Rejected</td>
</tr>
<tr>
<td>H_{03}</td>
<td>Accepted</td>
<td>Accepted</td>
<td>Accepted</td>
<td>Accepted</td>
</tr>
<tr>
<td>H_{04}</td>
<td>Rejected</td>
<td>Accepted</td>
<td>Accepted</td>
<td>Accepted</td>
</tr>
</tbody>
</table>
4.6 Exploring the relationship between package cohesion and maintainability using regression analysis

In this section, we present the second part of the empirical study we performed, using a regression analysis technique, to investigate the relationship between package cohesion, using the proposed metric CH, and maintainability in terms of maintenance effort. We expected that a package that has low cohesion will likely require a high maintenance effort, i.e., #Revisions and RLOC. Regression analysis is one of the most widely used analyses to predict a dependent variable based on independent variable(s). It measures the relationship between one or more independent variables and one dependent variable. This kind of regression analysis is widely applied to predict the degree of software external quality attributes, such as maintainability [124][114][132], or its characteristics [125][122][67][128].

The main advantages of regression analysis are its simplicity and that it is supported by many popular software packages [163], SPSS in our case. In our study, both univariate analysis and multivariate analysis were conducted using linear regression analysis. The univariate analysis was performed to examine the impact of package cohesion on package maintainability. The multivariate analysis was intended to examine the collective usefulness of CH and #Classes on predicting the package maintainability.

This analysis provides a statistical model for predicting the dependent variable (maintainability) by an equation of independent variable(s). The prediction model is in the form of equations where the dependent variable is expressed by predictors (independent variables) [164]. The general form of a simple linear regression model, which uses only one predictor (independent variable), is given by:

\[
y = \beta_0 + \beta_1 x
\]
Where $Y$ is the response, or the dependent variable, and $x$ is the independent variable. $\beta_0$ is the intercept, $Y$’s value when $x$ equals zero. $\beta_1$ is the estimated regression coefficient value of the predictor $x$.

The general form of the multiple regression model, which uses more than one predictor (independent variable), is given by:

$$ Y = \beta_0 + \beta_1 x_1 + \cdots + \beta_p x_p $$

Where $Y$ is the response, or the dependent variable, $x_1, \ldots, x_p$ are the predictors (independent variables), and $\beta_1, \ldots, \beta_p$ are the estimated regression coefficients.

The strength of the impact of the independent variable is determined by the absolute value of the coefficient. For each independent variable, $x$, the p-value is computed and assessed by the ($\alpha = 0.05$) significance level [111]. $R^2$ is the square of ($r$), the correlation coefficient discussed in section 4.5. $R^2$ is the proportion of the total variance in the dependent variable that is explained by the model. It is a measure of how well the regression model predicts the dependent variable, and it takes values from 0 to 1, i.e., $0 \leq R^2 \leq 1$. As high as is the $R^2$ value, as high is the impact of the independent variables on the dependent one, and as reliable is the model [115][111].

**Goodness-of-fit**

To verify the goodness-of-fit of the constructed simple and multiple regression analysis models, we use the following tests:

- $R^2$ (R square) is the most popular measure of the goodness-of-fit of the regression model. $R^2$ is the proportion of variation in the dependent variable that is accounted for by the variation in the independent variable. Its value varies between 1 (perfect relationship) and 0 (no relationship). Increases in the $R^2$, when predictors
are added, indicate improvements in the regression model [103]. The values of $R^2$ that are considered to be “good” depend on the context. There is no absolute number but if it is hard to predict the dependent variable, the small values of $R^2$ will be good.

- $R^2_{adj}$ (adjusted R square) is developed to minimize the impact of sample size and to adjust the number of parameters (independent variables). It incorporates the regression model’s degrees of freedom. It will increase as the variables are added if the increase in model fit is worthwhile [103].

- F-test evaluates the null hypothesis that all regression coefficients are equal to zero versus the alternative that at least one is not. A significant F-test indicates that the observed $R^2$ is reliable. Thus, the F-test determines whether the regression model is statistically reliable, and can be useful when the research objective is either prediction or explanation [103].

- $t$-test tests the null hypothesis that a particular predictor coefficient is equal to zero versus it is not equal to zero. While the F-test tests the fitness of the whole model, it does not tell us the significance of each predictor in the model. When the $t$-test is significant, it implies that the coefficient is significantly different from zero, and its associated predictor is contributing to the significance of the model.

### 4.6.1 Dependent Variables

Two dependent variables in this study are defined for linear regression analysis to measure the maintainability of packages. Al Dallal [133] suggested similar measures for class maintainability. We consider maintainability from the maintenance effort point of view using two maintenance measures:
The number of revisions (#Revisions) in which the package has been involved in maintenance activities during the studied history of software maintenance. The #Revisions variable represents the maintenance rate, which is how many times has the package been involved in maintenance activities.

The number of revised lines of code (RLOC) during the studied maintenance history. RLOC represents the maintenance cost and maintenance effort.

4.6.2 Statistical Analysis

After examining the correlation, we realized that the four studied systems have a consistent behavior (between cohesion and maintenance) and characteristics. This reason, along with the need for a larger data set to run the regression analyses, encouraged us to integrate the four data sets to form one data set for the regression analyses. Table 4.7 presents the means and standard deviations for the variables used in the regression analyses based on the combined data.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Combined Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>N=477</td>
</tr>
<tr>
<td></td>
<td>Mean</td>
</tr>
<tr>
<td></td>
<td>S.D.</td>
</tr>
<tr>
<td>CH</td>
<td>.445</td>
</tr>
<tr>
<td>#Classes</td>
<td>16.23</td>
</tr>
<tr>
<td>LogClasses</td>
<td>.943</td>
</tr>
<tr>
<td>#Revisions</td>
<td>6.124</td>
</tr>
<tr>
<td>Log#Revi</td>
<td>.530</td>
</tr>
<tr>
<td>RLOC</td>
<td>240.096</td>
</tr>
<tr>
<td>LogROLC</td>
<td>1.349</td>
</tr>
</tbody>
</table>

Data screening and evaluation of the linearity condition for the regression analysis led to the natural log transformation of the variables number of revisions (#Revisions), package size (#Classes), and number of revised lines of code on the package (RLOC).
4.6.3 Results and Discussion

Two sets of regression analyses run to predict package maintainability. The first set of regression analyses is for predicting the natural log of the number of revisions (log#Rev) as the first measure for package maintainability. The second set of regression analyses is for predicting the natural log of the number of revised lines of code (logRLOC) as the second measure for package maintainability. For each set, we run both: the simple (univariate) regression analysis where package maintainability is predicted by the newly proposed measure of cohesion (CH); and the multiple (multivariate) regression analysis to predict package maintainability by the newly proposed measure of cohesion (CH), controlling for the natural log transformed package size (LogClasses). The addition of the size (LogClasses) variable can improve the regression model results, and this is verified by the proportional increases in $R^2$ and $R^2_{adj}$ values.

Number of Revisions

Simple Regression

Regression results for the number of revisions (Log#Rev) indicate that the overall model of the new proposed measure of package cohesion (CH) significantly predicts the natural log of the number of package revisions [$R^2=0.284$, $R^2_{adj} = 0.282$, $F(1, 475) = 187.970$, $p = 0.000$].

The prediction model accounts for 28.4% of the variance in the log number of package revisions, which is a good value in this case. Table 4.8 presents a summary of the regression model coefficients.

<table>
<thead>
<tr>
<th>Predictor</th>
<th>B</th>
<th>Beta</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>CH</td>
<td>-.654</td>
<td>-.532</td>
<td>-13.710**</td>
</tr>
</tbody>
</table>

** $p < 0.001$
Multiple Regression

Regression results for the number of revisions (Log#Rev) indicate that the overall model of the new proposed measure of package cohesion (CH) and package size (LogClasses) significantly predict the natural log of the number of package revisions \( R^2 = 0.513, R^2_{adj} = 0.511, F(2, 474) = 249.422, p = 0.000 \).

The prediction model accounts for 51.3% of the variance in the log number of package revisions, which is better than the case of the simple model. Table 4.9 presents a summary of the regression model coefficients.

Table 4.9 Summary of the model predicting the log number of package revisions (N=477)

<table>
<thead>
<tr>
<th>Predictor</th>
<th>B</th>
<th>Beta</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>CH</td>
<td>-.121</td>
<td>-.098</td>
<td>-2.268*</td>
</tr>
<tr>
<td>LogClasses</td>
<td>.659</td>
<td>.646</td>
<td>14.934**</td>
</tr>
</tbody>
</table>

* p < 0.05  ** p < 0.001

Number of Revised Lines of Code

Simple Regression

Regression results for the number of revised lines of code (LogLOC) indicate that the overall model of the new proposed measure of package cohesion (CH) significantly predict the natural log of the number of package revised lines of Code \( R^2 = 0.276, R^2_{adj} = 0.274, F(1, 475) = 180.660, p = 0.000 \).

The prediction model accounts for 27.6% of the variance in the log number of revised lines of code, which is a good value in this case. Table 4.10 presents a summary of the regression model coefficients.
Table 4.10 Summary of the model predicting the log number of revised lines of code (N=477)

<table>
<thead>
<tr>
<th>Predictor</th>
<th>B</th>
<th>Beta</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>CH</td>
<td>-1.485</td>
<td>-.525</td>
<td>-13.441**</td>
</tr>
</tbody>
</table>

** p < 0.001

Multiple Regression

Similarly, regression results for the number of revised lines of code (LogLOC) indicate that the overall model of the new proposed measure of package cohesion (CH) and package size (LogClasses) significantly predict the natural log of the number of package revised lines of Code [$R^2=0.441$, $R^2_{adj} = 0.439$, $F(2, 474) = 186.949$, $p = 0.000$].

The prediction model accounts for 44.1% of the variance in the log number of revised lines of code, which is better than the case of the simple model. Table 4.11 presents a summary of the regression model coefficients.

Table 4.11 Summary of the model predicting the log number of revised lines of code (N=477)

<table>
<thead>
<tr>
<th>Predictor</th>
<th>B</th>
<th>Beta</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>CH</td>
<td>-.441</td>
<td>-.156</td>
<td>-3.364**</td>
</tr>
<tr>
<td>LogClasses</td>
<td>1.289</td>
<td>.549</td>
<td>11.843**</td>
</tr>
</tbody>
</table>

** p < 0.001

4.6.4 Applying maintainability prediction models

Although developing maintainability prediction models in a research environment is interesting and potentially useful, these models truly become useful when software developers in an industrial environment can utilize them. We believe that our models can provide valuable information for software development teams in the design phase of software development. For this purpose, we have performed this study of maintainability using real data from four open source software systems.
Based on the results of this study, the multivariate models showed that combining other measures, such as size measure, along with cohesion measure could optimize the prediction models. We found that the multivariate models are always better than univariate models involving only cohesion when predicting maintenance effort. In practice, we believe that applying multivariate models will be more desirable than applying the univariate models.

Developers can use the produced prediction models to assess the software maintainability using package cohesion. Given more than one design for the software system, developers can decide the most maintainable design based on the predicted maintenance effort results obtained from the prediction models.

If the developers find that a specific package in the system has a low quality design, they may wish to develop alternative designs for that package, and then will be able to use the prediction models to reach the best package design. They can predict the software maintainability for the design of the package before and after the design has been changed. A simple comparison between the prediction model results for the old and the new designs of the package can provide the development team with the better design for that package.

Another scenario that might happen is that even after the prediction models located packages with relatively low maintainability in the software system, developers may not be willing to change the package design. However, this kind of packages should receive more attention and therefore they may need to be well documented and well tested. Well-tested packages are more likely to have fewer errors and faults. Similarly, good documentation can improve the system understandability and then enhance maintenance activities and reduce maintenance efforts.
Providing these benefits to a software development team can improve the software design with the goal of improving the software quality.

4.7 Conclusion

This chapter investigated the relationship between the software internal attribute, package cohesion, and the software external attribute, package maintainability. We empirically investigated this relationship and we found that package cohesion, using our proposed metric (CH), is highly correlated with package maintainability, measured by number of revisions (#Revisions) and number of revised lines of code (RLOC); see table 4.5. As high cohesion, the package is the easiest to be maintained. Such relationship is explained by the Spearman’s ranking correlations involving data sets of four Java open-source software systems. The high correlation found led us to run regression analyses to predict package maintainability from package cohesion. Predicting software maintainability during the software design phase can reduce much of maintenance costs and efforts.

Two linear regression analyses were run to predict package maintainability. Simple regression analysis was performed to predict package maintainability from package cohesion. The multiple regression was run to predict package maintainability using the collective usefulness of package cohesion (CH) and package size (#Classes) predictors.

One strength of this study is the number of the studied systems and the relatively large sample used in the linear regression analyses. The proposed package cohesion metric (CH) is found to be a significant early predictor for software maintenance efforts. The stability of the impact of CH across statistic analyses performed allows us to draw optimistic conclusions about its use as an indicator. Additionally, based on the obtained results, we can claim that following
the package cohesion principles by Martin [19], which the proposed cohesion metric was
developed based on, can improve the software maintainability.

The experiments support the ability of such indicator, based on objective empirical
studies, to predict software maintainability, although it may behave differently based on a
system's domain. So the results in this study should be viewed as indicative rather than
conclusive.

Knowing that maintainability is affected by different factors, it would be very interesting
to consider other metrics, such as coupling, along with those already used, cohesion and size, to
predict the maintenance effort. It would be also interesting to involve more different metrics for
cohesion, coupling, and size; and comparing these metrics to cohesion metrics in terms of
predicting maintainability. The study only involved systems developed in Java, and the results
could be different with systems developed in other object-oriented languages (such as C++).
CHAPTER 5

Testability Prediction Using Package Cohesion

5.1 Introduction

Software testability is known to be one of the software maintainability characteristics, figure 2.1. During the software development process, the detection of faults and errors is always one of the main goals for the software development team. The early detection of errors and faults can reduce the maintenance effort and costs. Software testing is the process that offers this advantage to deliver a high quality software system. Software testability aims to facilitate the process of software testing. ISO [130] defines software testability as “attributes of software that bear on the effort needed to validate the software product.” Another definition by ISO [135] is the degree of effectiveness and efficiency with which test criteria can be established for a system, product, or component, and tests can be performed to determine whether those criteria have been met. IEEE [130] defines it as the degree of the software that facilitates the establishment of test criteria and the performance of tests to determine whether criteria have been met. Software testability has a relation to testing effort reduction and software quality [138]. As stated by Gao et al. [138], the late detection of a lack of testability may be difficult and expensive to repair, and it can badly affect the testing and maintenance effort [111].

It has been argued that software testability, as one of the maintainability characteristics, should be considered as a key factor in software quality. Software quality measurements depend on software testability measures. Therefore, predicting software testability using software measurements is expected to give a chance to improve software quality.
Measuring software attributes that have an impact on testability after coding for the purpose of testability evaluation is later and more costly. However, predicting testability earlier, during the software design phase, may greatly reduce the cost and the time [136]. Software metrics can predict software quality characteristics [137][33][52]. Several metrics were proposed to predict quality attributes related to testability such as maintainability. As discussed earlier in Chapter 4, many studies have investigated the role of cohesion, which is one of the most important software quality internal attributes, in predicting software maintainability, and how cohesion can impact software maintainability in different abstraction levels. A cohesion metric can be a good predictor for software maintainability and software testability. Such predictions include, but are not limited to, fault prone-ness and defect density. In this chapter, we empirically investigate the relationship between package cohesion and package testability of the software system. Our hypothesis is that a package with low cohesion is difficult to test.

5.2 Internal and External Attributes

Software product quality is divided into internal and external attributes [32], see section 2.6. External attributes cannot be only measured based on software artifacts [27][133]. That means that the external attribute of the package, i.e., testability, cannot be known before it is measured, i.e., before the package has been tested. However, the external attribute can be predicted by using internal attributes as indicators for the external ones.

In this empirical study, we consider package testability as the external attribute. To estimate the package testability, an external attribute, we use package cohesion, an internal attribute, as an independent variable to predict the testability of the package. This prediction process requires data collection of the testing effort from the actual software artifacts. We use the
number of testing lines of code (TLOC) to indicate the package testing effort. This study follows the approach used by [130][140][146][148][111].

On the other hand, the internal attribute of the package, i.e., cohesion, can be measured based on the package design itself before the package has been tested and before the package is written. A package cohesion metric is useful when it is used to predict a package external attribute, testability [152]. In this empirical study, package cohesion will be measured using the metric defined in Chapter 3 of this research.

5.3 Related Work

Finding a clear view of all the factors that can affect software testability is difficult because testability is an elusive concept [136]. Many testability approaches have been proposed to investigate the degree of software testability.

Freedman [139] proposed testability metrics based on observability and controllability. The proposed testability measures examine the input and output domains. He meant by observability the ability of specific input to affect the output. Controllability is meant to be the ease of producing specific output from specific input. Fenton et al. [27] considers software testability as an external attribute that can be affected by internal attributes. Voas [141] states that the test case of a component will fail if it has a fault. Using the testability definition proposed by Voas [141], Khoshgoftaar et al. [145] modeled the relationship between static software measures and testability. They developed two distinct models, and classified the program modules as having low or high testability. Jungmayr [142] focuses on dependencies between software components and proposed the notion of “test-critical dependencies.” This new concept is used to estimate testability of software through integration testing. The reduction metric is used to calculate the effect of individual factors to find out the required testability
metric [136]. Bruntink and Deursen [140] proposed some testability metrics to assess the

testability of the classes of Java systems. They defined some testability factors based on source
code metrics. One limitation of this study is the late detection of errors, which makes any repair
expensive. Without empirical validation, Baudry et al. [143] aimed to detect the weaknesses of a
UML class diagram to reduce the final testing effort. They addressed two configurations in a
UML class diagram that can lead to code difficulties in testing. They proposed a testability
measurement for a UML class diagram as well as solutions to improve the testability of the
software design. Jianping and Minyan [144] proposed a request-oriented method of software
testability measurement. The proposed method can select the appropriate elements from a self-
contained software testability measurement framework to measure testability of all kinds of
software. Their goal was to lower the difficulty and the cost of a software testability
measurement as well as to accelerate the application and development of a software testability
measurement.

Badri et al. [111][130] aimed to empirically explore the relationship between a lack of
cohesion and the testability of classes in object-oriented systems. Using two Java software
systems that have JUnit test cases, they evaluated the capability of lack of cohesion metrics to
predict testability. The results support the idea that there is a significant relationship between the
(lack of) cohesion of classes and testability. In another work, Badri et al. [146] performed an
empirical investigation to study the relationship between object-oriented design metrics and the
testability of classes. Using logistic regression methods, they evaluated the individual and the
combined effect of metrics on the unit testing effort of classes. The results indicated that
complexity, size, cohesion, and (to some extent) coupling were found to be significant predictors
of the testing effort of classes. Later, Badri et al. [150-151] studied the effect of control flow of
the unit testing effort of classes. They classified the classes into low and high according to the required testing effort.

Singh et al. [147] measured the testing effort in terms of lines of code added or changed during the life cycle of a defect. They predicted testing effort using object-oriented metrics and neural networks [147]. In another work, Singh et al. [148] performed a case study on Eclipse to predict testability at the package level. The results showed that there is a significant correlation between source code metrics and test metrics that obtained from JUnit test classes of test packages. They found that the low value of cohesion increases testing effort and decreases testability.

Although there is a good interest in software testability on the class level as seen from the above, software testability on the package level has not received the same interest. So this study focuses on the package testability and how it is affected by package cohesion. It uses the cohesion metric on the package level, Chapter 3, proposed based on the well-known package cohesion principles, both theoretically and experimentally validated. Actual testing data of software have been used to investigate the relationship between the internal quality attribute, package cohesion, and the external quality attribute, package testability, by conducting several statistical analysis tests.

We performed a set of analyses to look into the relation between software package cohesion (CH) and package testability (TLOC). Five systems Camel, Tomcat, Hadoop, Synapse, and Ant provided five data sets for the analyses of software testability. Similar to the package maintainability analysis, the analysis for testability mirrored each one of the five data sets provided by the five systems Camel, Tomcat, Hadoop, Synapse, and Ant except the regression analyses. Similarly, regression analyses were run on the integrated data from the five systems.
The regression analyses included package size (#Classes) as part of the prediction model to control for its effect and to improve model prediction.

5.4 Descriptive Statistics

This empirical study was conducted on five open-source Java software systems to discover the role of a package cohesion measure in predicting software testability. This section provides descriptions about the studied software systems and the testing data collection. Two package cohesion metrics are included in this study, Martin’s cohesion metric (H) and our proposed package cohesion metric (CH), discussed in Chapter 3, which was developed based on Martin’s package cohesion principles [19].

5.4.1 The software systems

Five open-source Java software systems were involved in the empirical study. All the five systems were selected based on the following criteria to allow results generality; they had: (1) to be implemented using Java programming language, (2) to have testing cases available, (3) to have a sufficient number of versions for each system that have been tested, (4) to be organized using packages, (5) to have different sizes ranging from very large to small systems in terms of number of packages and number of classes, (6) to be from different domains, and (7) to have positive reviews and to be mature. We expect these criteria will allow the generalization of the results obtained from the study. The first system, Camel [153], is rule-based and mediation engine to configure routing and mediation rules. The second system, Tomcat [154], is an open source webservice developed to implement Javaservlet and Java Server pages (JSP). Apache Tomcat is developed by the Apache Software Foundation. It has been developed and released under Apache License version 2. The third system, Hadoop [165], is open source framework
software for large-scale processing of data sets on clusters of computers. It is licensed under the Apache License 2.0. The fourth system, Synapse [166], is a lightweight and high-performance open source Enterprise Service Bus (ESB). It provides exceptional support for XML and Web Services. It also supports several content interchange formats. Apache Synapse is licensed under the Apache Software License version 2.0. The fifth system, Ant [167], is a Java library and command-line tool used to build Java applications. It can be also used to build non-Java applications such as C or C++ applications. Table 5.1 provides details about the studied software systems.

<table>
<thead>
<tr>
<th>System</th>
<th>Release</th>
<th>#LOC</th>
<th>#Classes</th>
<th>#Packages</th>
<th>#TestPackages</th>
<th>#TestClasss</th>
<th>#TLOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Camel</td>
<td>2.0.0</td>
<td>143732</td>
<td>5111</td>
<td>264</td>
<td>44</td>
<td>459</td>
<td>31494</td>
</tr>
<tr>
<td>Tomcat</td>
<td>7.0.6</td>
<td>170461</td>
<td>1725</td>
<td>113</td>
<td>37</td>
<td>323</td>
<td>10732</td>
</tr>
<tr>
<td>Hadoop</td>
<td>2.2.0</td>
<td>522903</td>
<td>4445</td>
<td>222</td>
<td>205</td>
<td>3351</td>
<td>181007</td>
</tr>
<tr>
<td>Synapse</td>
<td>2.1.0</td>
<td>82032</td>
<td>1115</td>
<td>117</td>
<td>47</td>
<td>288</td>
<td>13177</td>
</tr>
<tr>
<td>Ant</td>
<td>1.92</td>
<td>106300</td>
<td>1120</td>
<td>67</td>
<td>36</td>
<td>442</td>
<td>23170</td>
</tr>
</tbody>
</table>

5.4.2 Testing Data

The source of the testing data for this study is the test classes of the studied systems. Test classes are written for the purpose of software testing. JUnit, which is an open source framework, is designed for running tests in Java programming language [168]. JUnit has gained a lot of popularity [169][170]. It helps in testing a Java class by defining how to write the corresponding classes and provides the tool to run them [125]. JUnit gives testers support to write test classes for the system classes, in a convenient way, and then run them to output a report about the successful and failed methods of the class tested [168][111]. The source class and the test class
can be kept in the same or different packages [125]. The more test classes that are written, the more the testing effort.

To indicate the testing effort required for the software package, we consider a Testing Lines Of Code (TLOC) measure in which we count the number of LOC used for the purpose of testing the classes of the package. The written TLOC represents the effort spent to test a specific package. The more TLOC written, the more effort is spent and the lower the testability of the package. We consider this measure for two reasons. First, it seems to be a reasonable and a good measure for testing effort in terms of cost and time spent to test a package. Second, this measure is measurable using the freely available data.

Two computer science PhD students were dedicated to collecting the testing data. The data was collected manually from the systems’ artifacts. The collected data were tested to check its validity. This process increased our confidence about the validity of the data collected.

For the purpose of listing all classes in each system and listing all packages, we have used the JHawk tool [157]. Then, each test class is assigned to its system class along with its LOC. Then, testing effort data are collected on the package level. The testing effort data were collected individually for the five studied systems. Table 5.2 summarizes testing data for the studied systems.

<table>
<thead>
<tr>
<th>System</th>
<th>#LOC</th>
<th>Mean #LOC</th>
<th>#TLOC</th>
<th>Mean #TLOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Camel</td>
<td>143732</td>
<td>544</td>
<td>31494</td>
<td>715</td>
</tr>
<tr>
<td>Tomcat</td>
<td>170461</td>
<td>1508</td>
<td>10732</td>
<td>290</td>
</tr>
<tr>
<td>Hadoop</td>
<td>522903</td>
<td>2355</td>
<td>181007</td>
<td>882</td>
</tr>
<tr>
<td>Synapse</td>
<td>82032</td>
<td>701</td>
<td>13177</td>
<td>280</td>
</tr>
<tr>
<td>Ant</td>
<td>106300</td>
<td>1586</td>
<td>23170</td>
<td>643</td>
</tr>
</tbody>
</table>
5.4.3 Package Cohesion Data

Package cohesion data is gathered from two package cohesion metrics. The first metric is our proposed package cohesion metric in Chapter 3, CH. The second metric is Martin’s cohesion metric, H. These two metrics have been used to investigate the correlation between package cohesion and testability. For the purpose of data gathering, we have developed our Java tool to measure the CH package cohesion metric. The tool has been extended to calculate Martin’s package cohesion metric, H. For each system, a list of all the packages, number of classes in each package, and the associated cohesion values are generated.

5.5 Exploring the relationship between package cohesion and testability using correlation

In this section, we present the correlation part of the empirical study we performed to explore the relationship between package cohesion and package testability, in terms of testing effort. Correlation is one of the widely used statistical tests to investigate the relationship between internal quality attributes, e.g., cohesion, and external quality attributes, e.g., testability; and it has been used in different studies such as [130][140][148][150][160]. The correlation analysis aims to determine whether each individual package cohesion metric (CH and H) is significantly related to the testing measure, TLOC, of the package. For this purpose, we have performed Spearman’s rank correlation due to the non-parametric nature of the metrics’ data. We have used the well-known SPSS software for the correlation analysis of the empirical study. We have created and analyzed a correlation matrix for each software system in the study. Each correlation matrix has all the studied variables (cohesion and testing), a correlation coefficient (r), and significance level. For each pair of variables, the value of (r) can range between -1 and +1, where 1 represents a perfect positive correlation between the pair variables; -1 denotes a perfect negative correlation; and 0 indicates that there is no relationship between the variables.
The magnitude of the coefficient determines the degree of the correlation. The ratings of correlation strength follow the adjectives developed by Cohen [161]:

<table>
<thead>
<tr>
<th>Correlation coefficient</th>
<th>Adjective rating</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; 0.1</td>
<td>Trivial</td>
</tr>
<tr>
<td>0.1 to 0.3</td>
<td>Minor</td>
</tr>
<tr>
<td>0.3 to 0.5</td>
<td>Moderate</td>
</tr>
<tr>
<td>0.5 to 0.7</td>
<td>Large</td>
</tr>
<tr>
<td>0.7 to 0.9</td>
<td>Very large</td>
</tr>
<tr>
<td>0.9 to 1</td>
<td>Almost perfect</td>
</tr>
</tbody>
</table>

Besides the strength of the correlation, the relationship between any pair of variables should be assessed for its significance as well. The significance is assessed by the \( p \)-value, which corresponds to the probability that the found correlation might be due to purely random effects. The smaller the \( p \)-level, the more significant is the relationship between variables [162]. The significance of the correlation in this empirical study was tested at 95% confidence level (i.e., \( p \)-level \( \leq 0.05 \)). While the correlation can establish the relationship, it cannot establish a cause-effect relationship between the pair of variables [162].

5.5.1 Hypotheses

Our objective in this experiment is to explore empirically to what extent package cohesion is related to the package testability, in terms of testing effort. We evaluated cohesion at the package level and we counted the testing effort at the package level based on the test classes of the software system. The hypotheses of the empirical study are:

\( H_{01} \): There is no significant correlation between package cohesion, CH, and the number of testing lines of code, TLOC.
There is no significant correlation between Martin’s package cohesion, H, and the number of testing lines of code, TLOC.

In this experiment, rejecting the null hypothesis indicates that there is a statistically significant relationship between the pair of variables (significance level $\alpha = 0.05$).

### 5.5.2 Statistical Analysis

The number of testing lines of code (TLOC) of the software package assesses the software package testability. A smaller number of testing lines of code during the software-testing phase indicates that less effort is needed to test the software, i.e., the software is highly testable.

Data screening and evaluation of linearity for assessing correlations and regression analysis led to the natural log transformation of the variables number of testing lines of code on the package during the testing phase (TLOC) and package size (#Classes). We included Martin’s package cohesion metric (H) in the list of variables for the purpose of comparison.

Table 5.3 provides descriptive statistics (mean and standard deviation) for the variables used in analyzing software testability across the five systems, Camel, Tomcat, Hadoop, Synapse, and Ant.

### Table 5.3 Means and Standard Deviations of the variables used in testability analysis

<table>
<thead>
<tr>
<th>Variable</th>
<th>Camel N=264</th>
<th>Tomcat N=113</th>
<th>Hadoop N=222</th>
<th>Synapse N=117</th>
<th>Ant 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>H</td>
<td>.636</td>
<td>.361</td>
<td>.817</td>
<td>.524</td>
<td>.674</td>
</tr>
<tr>
<td>CH</td>
<td>.530</td>
<td>.388</td>
<td>.358</td>
<td>.374</td>
<td>.281</td>
</tr>
<tr>
<td>LogClasses</td>
<td>.772</td>
<td>.518</td>
<td>.968</td>
<td>.478</td>
<td>1.045</td>
</tr>
<tr>
<td>TLOC</td>
<td>119.295</td>
<td>689.072</td>
<td>76.60</td>
<td>206.907</td>
<td>815.35</td>
</tr>
<tr>
<td>LogTLOC</td>
<td>.397</td>
<td>.930</td>
<td>.558</td>
<td>.997</td>
<td>1.618</td>
</tr>
</tbody>
</table>
5.5.3 Results and Discussion

Spearman Rho correlation is the appropriate 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 proposed measure of package cohesion CH, the Martin’s package cohesion metric H, package size (#Classes), and the measure of package testability, the number of testing lines of code (TLOC), within each of the five data sets. Table 5.4 provides the list of these correlations for the five sets of data.

Table 5.4 Spearman's Rho correlations for testability analysis

<table>
<thead>
<tr>
<th>Data Set</th>
<th>H</th>
<th>CH</th>
<th>#Classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Camel</td>
<td>.281**</td>
<td>-.350**</td>
<td>-.655**</td>
</tr>
<tr>
<td></td>
<td>.329**</td>
<td></td>
<td>.334**</td>
</tr>
<tr>
<td>N=264</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Camel</td>
<td>.086</td>
<td></td>
<td></td>
</tr>
<tr>
<td>N=264</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tomcat</td>
<td>.169</td>
<td>-.069</td>
<td>-.736**</td>
</tr>
<tr>
<td>N=113</td>
<td></td>
<td>-.123</td>
<td>-.394**</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>.549**</td>
</tr>
<tr>
<td>Hadoop</td>
<td>.157</td>
<td>.063</td>
<td>-.688**</td>
</tr>
<tr>
<td>N=222</td>
<td></td>
<td>-.033</td>
<td>-.488**</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>.615**</td>
</tr>
<tr>
<td>Synapse</td>
<td>-.038</td>
<td>-.084</td>
<td>-.490**</td>
</tr>
<tr>
<td>N=117</td>
<td></td>
<td>-.310**</td>
<td>-.199*</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>.465**</td>
</tr>
<tr>
<td>Ant</td>
<td>.227</td>
<td>.078</td>
<td>-.527**</td>
</tr>
<tr>
<td>N=67</td>
<td></td>
<td>-.135</td>
<td>-.385**</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>.450**</td>
</tr>
</tbody>
</table>

** Correlation is significant at the .001 level
* Correlation is significant at the .05 level

Table 5.4 reveals that the new proposed measure of package cohesion CH, consistently has a negative moderate correlation with the measure of package testability, the number of testing lines of code (TLOC), across all the five data sets except for the Synapse system where
the correlation is minor. The correlation values between package cohesion CH and the number of testing lines of code (TLOC) across the five data sets ranges from -0.199 (for the Synapse system data set) to -0.488 (for the Hadoop system data set). The statistically significant correlations confirm the expectation that a highly cohesive software package requires less effort to be tested. That is high values of the proposed measure of package cohesion are associated with lower number of testing lines of code.

Correlations between Martin’s package cohesion metric H and the package testability measure, the number of testing lines of code (TLOC), tend to be not as strong as the ones with the newly proposed measure of package cohesion CH. These correlations are consistently weak and statistically insignificant across all the five data sets, except for the correlation with the testing lines of code (TLOC) for Synapse system’s data. The value of the correlation is -.310, which is statistically significant at the .001 level. The correlation values between Martin’s package cohesion H and the number of testing lines of code (TLOC) across the five data sets is never more than 0.135 except for Synapse system data set (-0.310).

Table 5.5 summarizes the results of the examined null hypotheses. In this experiment, rejecting the null hypothesis indicates that there is a statistically significant relationship between the pair of variables (significance level \( \alpha = 0.05 \)).

<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>( H_{01} )</td>
<td>Rejected</td>
<td>Rejected</td>
<td>Rejected</td>
<td>Rejected</td>
<td>Rejected</td>
</tr>
<tr>
<td>( H_{02} )</td>
<td>Accepted</td>
<td>Accepted</td>
<td>Accepted</td>
<td>Rejected</td>
<td>Accepted</td>
</tr>
</tbody>
</table>
5.6 Exploring the relationship between package cohesion and maintainability using regression analysis

In this section, we present the second part of the empirical study we performed, using a regression analysis technique, to investigate the relationship between package cohesion, using the proposed metric, and package testability in terms of testing effort. We expected that a package that has low cohesion will likely require a high testing effort, Testing Lines of Code (TLOC). Regression analysis is a widely used statistic analysis to predict a dependent variable based on independent variable(s). It measures the relationship between one or more independent variables and one dependent variable. This kind of regression analysis is widely applied to predict the software external quality attributes, such as maintainability [124][114][132], fault proneness [125][122], and reusability [67][128].

The main advantages of regression analysis are its simplicity and that it is supported by many popular software packages [163], SPSS in our case. Both univariate analysis and multivariate analysis were conducted using linear regression analysis. The univariate analysis was performed to examine the impact of package cohesion on package testability. The multivariate analysis was intended to examine the collective usefulness of CH and #Classes variables on predicting the package testability.

This analysis provides a statistical model for predicting the dependent variable, package testability, by an equation of independent variable(s). The prediction model is in the form of equations where the dependent variable is expressed by one or more predictors (independent variables) [164]. The general form of a simple linear regression model, which uses only one predictor (independent variable), is given by [171]:

$$Y = \beta_0 + \beta_1 x$$
Where $Y$ is the response, or the dependent variable, and $x$ is the independent variable. $\beta_0$ is the intercept, $Y$’s value when $x$ equals zero. $\beta_1$ is the estimated regression coefficient value of the predictor $x$.

The general form of the multiple regression model, which uses more than one predictor (independent variable), is given by [171]:

$$Y = \beta_0 + \beta_1 x_1 + \cdots + \beta_p x_p$$

where $x_1, \ldots, x_p$ are the predictors (independent variables) and $\beta_1, \ldots, \beta_p$ are the estimated regression coefficients.

The strength of the impact of the independent variable is determined by the absolute value of the coefficient. For each independent variable, $x$, the p-value is computed and assessed by the ($\alpha = 0.05$) significance level [111]. $R^2$ is the square of ($r$), the correlation coefficient discussed in section 5.5. $R^2$ is the proportion of the total variance in the dependent variable that is explained by the model. It is a measure of how well the regression model predicts the dependent variable, and it takes values from 0 to 1, i.e., $0 \leq R^2 \leq 1$. As high as the $R^2$ value, as high is the impact of the independent variables on the dependent ones, and as reliable is the model [115][111].

**Goodness-of-fit**

Please review section 4.6.

**5.6.1 Dependent Variables**

The linear regression analysis in this study includes only one dependent variable to measure the testing effort of the packages. We considered testability from the testing effort point of view. The testing effort of the package was measured by a TLOC (Testing Lines Of Code)
metric, which was proposed by Brutink and Deursen [101]. The TLOC metric was calculated from the JUnit test classes and then calculated at the package level. This measure represented the effort spent to test the package. To achieve high package testability, testing effort should be very low.

5.6.2 Statistical Analysis

After examining the correlation, we realized that the five studied systems have a consistent behavior (between cohesion and testing) and characteristics. This reason, along with the need for a larger data set to run the regression analysis, encouraged us to integrate the five data sets to form one data set for the regression analysis. Table 5.6 presents the means and standard deviations for the variables used in the regression analysis based on the combined data.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Combined Data N=783</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mean</td>
</tr>
<tr>
<td>CH</td>
<td>.438</td>
</tr>
<tr>
<td>#Classes</td>
<td>15.48</td>
</tr>
<tr>
<td>LogClasses</td>
<td>.938</td>
</tr>
<tr>
<td>TLOC</td>
<td>328.87</td>
</tr>
<tr>
<td>LogTLOC</td>
<td>.904</td>
</tr>
</tbody>
</table>

Data screening and evaluation of the linearity condition for the regression analysis led to the natural log transformation of the variables number of testing lines of code of the package (TLOC) and the package size (#Classes).

5.6.3 Results and Discussion

The regression analysis was run to predict package testability. The simple (univariate) regression analysis was run to predict package testability by the newly proposed measure of
cohesion (CH). To improve the prediction, the multiple (multivariate) regression analysis was run to predict package testability by the newly proposed measure of cohesion (CH), controlling for the natural log transformed package size (LogClasses). The addition of the size (LogClasses) variable can improve the regression model results, and this is verified by the proportional increases in $R^2$ and $R^2_{adj}$ values.

**Number of Testing Lines of Code**

*Simple Regression*

Regression results for the number of testing lines of code (LogLOC) indicate that the overall model of the new proposed measure of package cohesion (CH) significantly predicts the natural log of the number of package testing lines of code [$R^2=0.147$, $R^2_{adj}=0.146$, $F(1, 781) = 134.448, p = 0.000$].

The prediction model accounts for 14.7% of the variance in the log number of testing lines of code. Table 5.7 presents a summary of the regression model coefficients.

**Table 5.7 Summary of the model predicting the log number of testing lines of code (N=783)**

<table>
<thead>
<tr>
<th>Predictor</th>
<th>B</th>
<th>Beta</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>CH</td>
<td>-1.237</td>
<td>-0.383</td>
<td>-11.595**</td>
</tr>
</tbody>
</table>

**Multiple Regression**

Regression results for the number of testing lines of code (LogLOC) indicate that the overall model of the new proposed measure of package cohesion (CH) and the package size (LogClasses) significantly predict the natural log of the number of package testing lines of Code [$R^2=0.286$, $R^2_{adj}=0.284$, $F(2, 780) = 156.359, p = 0.000$].
The prediction model accounts for 28.6% of the variance in the log number of testing lines of code, which is better than the case of the simple model. Table 5.8 presents a summary of the regression model coefficients.

**Table 5.8 Summary of the model predicting the log number of testing lines of code (N=783)**

<table>
<thead>
<tr>
<th>Predictor</th>
<th>B</th>
<th>Beta</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>CH</td>
<td>-0.297</td>
<td>-0.092</td>
<td>-2.400*</td>
</tr>
<tr>
<td>LogClasses</td>
<td>1.315</td>
<td>0.473</td>
<td>12.338**</td>
</tr>
</tbody>
</table>

* * p < 0.05  
** ** p < 0.001

5.6.4 Applying testability prediction models

Please review section 4.6.4.

5.7 Conclusion

In this chapter, we investigated empirically the relationship between Package cohesion metrics (H and CH) and the testability of software packages in terms of required testing effort. We performed an empirical analysis using data collected from five Java open source software systems for which JUnit test classes are available. To measure the testability of packages, we used testing lines of code (TLOC) to quantify the corresponding testing effort.

One strength of this study is the number of the studied systems and the relatively large sample used in the linear regression analyses. The proposed package cohesion metric (CH) is found to be a significant early predictor. The stability of the impact of CH across statistic analyses performed allows us to draw optimistic conclusions about its use as an indicator. Additionally, based on the obtained results, we can claim that following the package cohesion principles by Martin [19], which the proposed cohesion metric was developed based on, can improve the software testability.
Spearman’s ranking correlation was used to evaluate the relationship between package cohesion metrics (H and CH) and the testing effort of packages. Another statistical analysis, simple and multiple regression analyses, was conducted to predict the testing effort of the package using the proposed package cohesion metric. The simple regression model evaluated the individual effect of the proposed package cohesion metric (CH) on the testing effort of packages. The multiple-regression model evaluated the combined effect of package cohesion (CH) and package size (#Classes) on the testability of packages. The results of both regression models supported that the proposed package cohesion metric (CH) is always significant in predicting the testing effort. Knowing that cohesion is one among other factors, such as size, that affect the testability of packages, the results of the analyses showed that the proposed package cohesion metric is a reliable predictor, to some extent, of the testability of packages. Moreover, the multiple-regression model showed that by including a size variable, the prediction model could be improved. However, the results of this study should be viewed as exploratory and indicative rather than conclusive.

We hope these findings will help lead to a better understanding of the relationship between package cohesion and package testability. However, we think that this study can be extended to include more testability measures, such as the number of test classes in packages. We also plan, in the future, to investigate the combined ability of multiple factors (i.e., cohesion, coupling, size) in predicting the testability of packages. Additionally, open source systems developed in other languages (such as C++) can be investigated, since this study has focused on Java open source systems.
CHAPTER 6

Conclusion

This dissertation explored the existing cohesion metrics. Since the package cohesion metrics haven’t received the same interest as the class level cohesion metrics, our interest is on the package level.

Although R. C. Martin has proposed well-known and well-accepted package cohesion principles, we see that his proposed package cohesion metric, H, has some drawbacks. We have developed a new package cohesion metric, CH, which accurately reflects the cohesion of packages. The new metric has been validated both theoretically and empirically.

The relationship between package cohesion and package maintainability has been empirically investigated. We found that package cohesion, using our proposed metric (CH), is highly correlated with package maintainability. We have measured package maintainability by the number of revisions (#Revisions) and the number of revised lines of code (RLOC). The high Spearman’s ranking correlations encouraged us to run regression analyses to predict package maintainability from package cohesion. Simple regression analysis was performed to predict package maintainability from package cohesion. The multiple regression was run to predict package maintainability using the collective usefulness of package cohesion and package size predictors. Predicting software maintainability during the software design phase can reduce much of the maintenance costs and efforts.

Another study in this research empirically investigated the relationship between package cohesion metrics (H and CH) and the testability of software packages in terms of the required testing effort. We performed an empirical analysis using data collected from five Java open
source software systems for which JUnit test classes were available. To measure the testability of packages, we used testing lines of code (TLOC) to quantify the corresponding testing effort.

Spearman’s ranking correlation was used to evaluate the relationship between package cohesion metrics (H and CH) and the testing effort of packages. Simple and multiple regression analyses were conducted to predict the testing effort of the package using the proposed package cohesion metric. The simple regression model evaluated the individual effect of the proposed package cohesion metric (CH) on the testing effort of packages. The multiple-regression model evaluated the combined effect of package cohesion (CH) and package size (#Classes) on the testability of packages. The results of both regression models showed that the proposed package cohesion metric (CH) is always significant in predicting the testing effort.

A strength of these two studies is the number of the studied systems and the relatively large sample used in the linear regression analyses. The proposed package cohesion metric (CH) is found to be a significant early predictor for software maintenance and software testability. The consistency of the impact of CH across the statistic analyses performed allows us to draw optimistic conclusions about its use as an indicator. Additionally, based on the obtained results, we can claim that following the package cohesion principles by Martin [19], which the proposed cohesion metric is developed based on, can improve the software maintainability and software testability.

The compatibility between the results of the two experiments (maintainability and testability), in which they are both in direct relationship with package cohesion, contributes to the validity of the experiments. The experiments demonstrate the ability of such indicator, based on objective empirical studies, to predict software maintainability and software testability,
although it may behave differently based on a system’s domain. Therefore, the results in this study should be viewed as indicative rather than conclusive.

Knowing that maintainability and testability are affected by different factors, it would be very interesting to consider other metrics, such as coupling, along with those already used, cohesion and size, to predict the maintenance effort and testing effort. When considering the combination of measures, the constructed prediction models can be more statically stable and provide better maintainability and testability predictions than the univariate models.

We hope that these findings will help lead to a better understanding of the relationship between package cohesion from the one side and package maintainability and package testability on the other side. However, we think that this study can be extended to include more maintainability and testability measures, such as the number of test classes in packages, costly packages, and frequently revised packages. We also plan, in the future, to investigate the combined ability of multiple factors (i.e., cohesion, coupling, size) to predict the maintainability and testability of packages.

It would also be interesting to involve more different metrics for cohesion, coupling, and size; and compare these metrics to cohesion metrics in terms of predicting maintainability and testability. This study only involved systems developed in Java, and the results could be different with systems developed in other object-oriented languages (such as C++).
REFERENCES


http://www.cs.jyu.fi/~koskinen/smcosts.htm


132


