A COMPREHENSIVE EMPIRICAL VALIDATION OF PACKAGE-LEVEL METRICS FOR OO SYSTEMS

BY

ALI HASHEM ALI

A Thesis Presented to the
DEANSHIP OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

In

COMPUTER SCIENCE

JUNE 2010

KING FAHD UNIVERSITY OF PETROLEUM & MINIEALS

Dhahran 31261, Saudi Arabia

DEANSHIP OF GRADUATE STUDIES

This thesis, written by Ali Hashem Ali under the direction of his thesis advisor and approved by his thesis committee, has been presented to and accepted by the Dean of Graduate Studies, in partial fulfillment of the requirements for the degree of MASTER OF SCIENCE IN COMPUTER SCIENCE.

Thesis Committee

[illegible]

Dr. Mahmoud Elish (Chairman)

[illegible]

Dr. Muhammad Al-Mulhem (member)

Sajjad Mahmood

Dr. Sajjad Mahmood (member)

[illegible]

Dr. Kanaan Faisal

Department Chairman

[illegible]

Dr. Salam A. Zummo

Dean of Graduate Studies

31/5/10

Date

عمادة الدراسات العليا
DEANSHIP OF GRADUATE STUDIES

DEDICATION

To my father and mother

To my brothers and sisters

To my friends and colleagues

ACKNOWLEDGMENTS

All praise is due to Allah the Almighty for his countless blessings and enlightenments throughout my studies.

Acknowledgement is due to King Fahd University of Petroleum & Minerals and ICS Department for supporting this research. My sincere appreciation goes to Dr. Mahmoud Elish, for all assistance, advice, encouragement and invaluable support given as my advisor and mentor throughout the period of this research. His constructive criticism and feedback proved invaluable to the development of this thesis. Great thanks are also due to my thesis committee members, Dr. Muhammad Al-Mulhem and Dr. Sajjad Mahmood, for their cooperation, comments, support, and contributions.

I wish to express my appreciation to my entire family for their love, support, prayers and encouragement throughout my life. In addition, I am thankful to my friends for their encouragement and friendship.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

THESIS ABSTRACT

NAME:	ALI HASEHM ALI
TITLE:	A COMPREHENSIVE EMPIRICAL VALIDATION OF PACKAGE-LEVEL METRICS FOR OO SYSTEMS
MAJOR FIELD:	COMPUTER SCIENCE
DATE OF DEGREE:	June 2010.

During the last few years, several package-level metrics have been developed and used to characterize the attributes of packages in object-oriented software design. These metrics provide ways to evaluate the quality of software. But how do we know which metrics are useful in capturing important quality attributes. Empirical studies are needed to provide relevant answers.

The proposed work will study sets of package-level metrics and empirically validate them against some implementation and quality assessment attributes. In addition, these metrics have been implemented in this work, to automate the extraction of the package level metrics. The collected metrics have been investigated to explore their capability to identify and predict pre- and post-release fault, change density and implementation effort for software packages. The results show that some of the package-level metrics are good indicators for change prediction, fault prediction and implementation effort estimation, even after controlling for package size.

ملخص الرسالة

الإســـــم : علي هاشم علي.

عنوان الدراسة : دراسة تطبيقية شاملة للتحقق من مقدرة مقاييس الحزم في البرامج غرضية التوجه.

التخصص : علوم الحاسب الالي.

تاريخ التخرج : يونيو 2010 م.

خلال السنوات القليلة الماضية، ظهرت العديد من االمقاييس على مستوى الحزم التي طورت واستخدمت لتشخيص سمات الحزم في تصميم البرامج غرضية التوجه. هذه المقاييس توفر سبل لتقييم جودة البرمجيات واستخدامها في المراحل الاولى من مراحل تطوير البرمجيات. ولكن كيف نعرف المقاييس المفيدة في التقاط سمات الجودة المهمة. لا بد عمل دراسات تطبيقية لتقديم أجوبة لمثل هذه الأسئلة. العمل المقترح هو عبارة عن دراسة لمجموعة من مقاييس الحزم و يشمل ايضا التحقق تطبيقيا من بعض سمات التنفيذ وتقييم الجودة. سيتم في هذه الدراسة إعادة تعريف بعض مقاييس المصفوفه لتكون على مستوى الحزم. كذلك تم تطويرأداة لأتممة عملية إستخراج هذه المقاييس. المقاييس المجمعة تم التحري عنها لاكتشاف قدرتها على التعرف والتنبؤ بأخطاء ماقبل اصدار البرنامج وبعده، ودرجة التغير في الحزم والجهد المبذول لتنفيذ الحزم. وقد أظهرت نتائج الإختبارات أن بعض مقاييس الحزم تعد مؤشرا جيدا للتنبؤ بالأخطاء، وتغير الحزم والجهد المبذول في تنفيذها حتى بعد التحكم في حجم الحزم.

Chapter 1

Introduction

The use of Object-Oriented Design (OOD) paradigm for designing and developing software has become quite widespread [1]. Designers are using OOD as it is a faster development process, module based architecture, includes high reusable features, increases design quality and so on [2]. Objects are the basic units of OOD. Identity, states and behaviors are the main characteristics of any object. A collection of objects which have common behaviors are called class. A class identifies the abstract characteristics of object, including the object's characteristics (its attributes or properties) and the behaviors (methods or operations**)**. Jacobson defined class as follow [3]: *"A class represents a template for several objects and describes how these objects are structured internally. Objects of the same class have the same definition both for their operation and for their information structure"*

Since object-oriented software applications grow in size and complexity, they require some kind of high-level organization. Classes, while a very convenient unit for organizing small applications, are too finely grained to be used as the sole organizational unit for large applications. This is where packages come in; packages are mechanism for organizing classes into namespaces. A package is a basic development unit that can be separately created, maintained, released, tested, and assigned to a team

[4]. Niemeyer defined Java packages as follow [1][5]:*"A java package is a group of classes that are related by purpose or by application. Classes in the same package have special access privilege with respect to one another and may be designed to work together closely."*

The purpose of a package is to increase the design quality of large applications by grouping classes that belong to the same category or providing similar functionality into packages [6].

Object-oriented metrics such as class and package-level metrics are used to measure properties of OOD. These metrics are used to characterize the attributes of software in OOD at different levels (e.g. class and package-levels). They provide ways to evaluate the quality of software and their use in earlier phases of software development [7]. They also help organizations in assessing large software development quickly and at a low cost [8, 9, 10, 11].

Studying and analyzing packages in object-oriented software in order to evaluate the quality of software is becoming increasingly important as OOD continues to increase in popularity, the size and number of packages of these software increases [12, 13, 14]. Consequently, several package-level metrics have been developed and used to characterize the attributes of packages in object-oriented software design. Many of these attributes have relation, in one way or the other, with the quality of the software being produced. Such attributes include: size, complexity, coupling and cohesion. Package-level design metrics can be used to measure these attributes. However, some of these

[1] Niemeyer definition has been used to define a package in this research

metrics may not really measure the intended quality attributes of software. Thus, empirical validation is necessary to demonstrate the usefulness of these metrics in practical applications [7]. This research will empirically investigate and study the effect of package-level design metrics on the quality of software. Moreover, sets of system and class-level metrics will be redefined and implemented so that they can be collected at the package-level. The package-level metrics will be investigated against some implementation and quality assessment attributes, mainly: package fault prediction, change prediction and implementation effort estimation.

1.1 Problem Statement

Several package-level metrics have been proposed and claimed to provide ways to evaluate the quality of software in earlier phases of software development. However, these metrics were not empirically validated. Therefore, empirical validation is necessary to demonstrate the usefulness of such metrics.

1.2 Motivation

With the rapid increase of the size and number of packages of software systems, several package-level metrics have been developed and used to characterize the packages in object-oriented software design. Once the design has been implemented, it is difficult and expensive to change. Therefore, the design should be good from the start. Package design metrics can help to evaluate and improve the quality of a design [12]. Studying design metrics at the package-level has many advantages:

- Since packages can be considered as subsystems and represent the coarse grained structure of an application [14], they can be used as early indicator for the design quality of the whole system.
- Package-level metrics can be collected easily and they are applicable to use in many stages through software development lifecycle, to enhance the quality of milestone of this stage. This is because these metrics don't require source code before they can be collected. For example, software designers can collect design package metrics at the design phase from UML diagrams and then use these mercies to evaluate systems designed whereby early preparation and planning for development are considered of special importance in order to save money in the long run.
- Package-level metrics can be also collected from source code during the implementation or testing phase. Developers and quality assurance engineers can use these metrics to determine change-prone packages or to identify faulty packages while they are implementing and testing the system.
- Software project managers can utilize package-level metrics to early estimate the implementation effort for the system packages which will help them in allocating the proper amount of resources to each package, and hence to the system, in early phases.

Quality models that investigate the relationship between package-level metrics, implementation and quality assessment attributes such as change prediction, fault prediction and effort estimation are needed to use these metrics effectively. Therefore, by empirically validating these metrics, software managers can find opportunities of improvements in the software as early as possible and to latter make plans for modifications that need to be implemented in the future. The importance of the attributes that will be measured and empirically validated is described as follow.

1.2.1 Fault Prediction

One of the most important goals of industrial software development is to achieve high reliability and quality of the released product. This quality is crucially influenced by the number of residual faults present in the product. A fault can be pre-release or post-release fault. Pre-release fault is a defect (e.g. programmer error) that causes a failure before a release, typically during software testing, while post-release fault is defect that causes a failure after a release and initial customer use [15]. It is therefore possible to increase the reliability of a product by reducing the number of residual faults present in the product. This can be achieved, among other things, by using metrics that measure factors that influence the number of faults in the packages. In addition, if a set of package-level metrics, extracted from packages in a release, are known to be a good predictor of fault (pre- or post-release) in that release, developer and quality team can focus on those packages to increase the reliability of the release of the software.

1.2.2 Change Prediction

In OOD, a software system consists of one or more packages. Each package consists of a set of classes. When a software changes, some classes will change (i.e. added, deleted or modified). Changes in these classes ripple through the system [16], affecting other packages and relationships to be reengineered. Since object-oriented software development is affected by change-prone classes (i.e. volatile classes) in packages, which requires effort and resources consumptions in development and maintenance [2], change-prone classes, and hence packages, need more investigation and research in purpose to recognize these packages by identifying their characteristics. Once this is achieved, developers can develop and apply certain actions on these packages of developed software as next increment that make the software in high quality level. Peer-reviews, testing, inspections, and refactoring are some examples of the expected actions on those packages. This, in turn, will help software developers to utilize their resources more efficiently and deliver higher quality products within time and budget.

1.2.3 Implementation Effort Estimation

Because of the growing diffusion of the size, number of packages in object-oriented software and the need of maintaining the process of software development under control, industries are looking for metrics capable of producing suitable implementation effort estimation [17]. These metrics have to produce results with a known confidence since the early phases of software life-cycle in order to establish a process of prediction and correction of costs. To this end, package metrics are needed to be empirically validated in order to estimate effort in object-oriented system development.

1.3 Literature Review

There are many studies that empirically investigated OOD metrics against change prediction [12, 18, 19, 20, 21, 22, 23, 24], fault prediction [15, 2, 25, 26, 27] and development effort estimation [28, 29]. However, the metrics used in these studies were at the class-level and not at the package-level. This work, however, is a comprehensive study that empirically investigates sets of OOD package-level metrics against the mentioned attributes.

Package-level metrics such as coupling and stability metrics have been studied by many researchers. Martin introduced a set of package-level metrics that are related to dependencies, abstraction and stability [12]. He proposed these metrics based on the Stable Dependencies Principle for volatile packages and Stable Abstraction Principle [12]. However, these metrics have not been empirically validated.

Zimmermann, Premraj and Zeller [15] used bug database to calculate pre- and post-release faults for every package and file in the Eclipse releases 2.0, 2.1, and 3.0. Then, they aggregated some of method and class-level complexity metrics at package-level and correlated them with pre- and post-release faults of a package. The results showed that the combination of complexity metrics can predict defects, suggesting that the more complex code it, the more faults it has. However, the predictions were far from being perfect [15].

Yau and Collofello [30] used measures based on algorithm for estimating the stability of a program and the packages of which the program is composed. They used computation based on data abstraction and global variable to find the logical stability. D'Ambros and Lanza [31] proposed evolution radar, which is interactive visualization technique, to understand the package coupling based on their evolution. On the other hand, Ducasse and colleagues [14] proposed a generic visualization technique that can be used to visualize, analyze and understand package relationships. They have applied this technique to several large applications to indicate badly designed packages.

Wilhelm and dehl [32] used Martin and size metrics to build a tool that helps to control package dependencies to avoid degeneration of package designs and hence increase the quality of the software. In addition, Reibing used these metrics to build a model called ODEM (Object-Oriented Design Model) [33]. This model can serve as a foundation for the formal definition of object-oriented design metrics and it provides a formal model of object-oriented designs expressed in UML.

This research will study sets of OOD metrics at package-level and empirically validate them against some of implementation and quality assessment attributes (i.e. change prediction, fault prediction and implementation effort estimation). These metrics include widely know class and system-lever metrics, which will be redefined here at package-level. In addition, prediction models will be produced for change prediction, fault prediction and implementation effort estimation.

1.4 Research Contributions

1- Empirically investigating sets of OOD metrics at the package-level to determine the usefulness of these metrics in identifying and predicting faults in object-oriented packages.

2- Empirically investigating sets of ODD metrics at the package-level to determine the usefulness of these metrics in identifying and predicting change in object-oriented packages.

3- Empirically investigating sets of OOD metrics at the package-level to determine the usefulness of these metrics in estimating implementation effort in object-oriented packages.

4- Redefining some of design metrics at the package-level and implementing them, to automate the extraction of these metrics.

1.5 Organization of the Thesis

The rest of the thesis is organized as follows: Chapter 2 provides a background of package-level metrics. Chapters 3, 4 and 5 present three empirical studies that investigate package-level metrics against fault prediction, change prediction and implementation effort estimation. Chapter 6 discusses thesis conclusion, limitation and provides directions for future work.

Chapter 2

Background

This chapter gives technical background about OOD and software metrics in OOD. In addition, it defines metrics that will be investigated in this study (i.e. Martin, C&K and MOOD metrics).

2.1 Characteristics of OOD

OOD is the process of planning a system of interacting objects for the use of solving a software problem [34]. It is one approach to software design. It is concerned with building an object-oriented module of a software system to apply the identified requirements. Designer will use OOD because it is a faster development process, module based architecture, includes high reusable features, increases design quality and so on [11]. Booch defined OOD as follow [35]:*“Object-oriented design is a method of design encompassing the process of object oriented decomposing and a notation for depicting both logical and physical as well as static and dynamic models of the system under design”*

There are a number of essential characteristics in object-oriented design. These characteristics are generally support object oriented design in the context of measuring. These are as follow:

- **Cohesion**

  Cohesion answers the question, should an object be only one object or more than one object? A cohesive object can be defined as the object that cannot easily be divided into multiple objects [36]. It is an object that has one clear purpose and one clear entity that it matches to. A class or package is cohesive when its components are highly correlated. It should be difficult to split a cohesive class or package. Cohesion can be utilized to recognize the weakly designed classes and packages. Booch defined cohesion as follow: "*Cohesion measures the degree of connectivity among the elements of a single class or object*" [36].

- **Coupling**

  Coupling shows the non-inheritance relationship or interdependency between modules (i.e. classes or packages). A class is coupled to another if methods of one class use non-inheritance methods or attributes of the other, or vice versa [37]. Similarly, a package is coupled to another if methods of one package use methods or attributes of the other, or vice versa. Coupling is a measure of interconnecting among modules in a software structure.

- **Inheritance**

  Inheritance is the ability for a class to extend or override functionality of another class. Inheritance occurs in all levels of a class hierarchy. Rumbaugh [38] defined inheritance as *"Inheritance is the sharing of attributes and operations among classes based on a hierarchical relationship".*

- **Polymorphism:**

Polymorphism which means many forms can be defined as the ability to replace an object with its sub-objects [39]. Polymorphism gives a generic software interface so that a collection of different types of objects may be manipulated uniformly.

2.2 Software Metrics in OOD

The definition of software metrics is introduced by Norman Fenton [40]: "*Software metrics is a collective term used to describe the very wide range of activities concerned with measurement in software engineering. These activities range from producing numbers that characterize properties of software code through to models that help predict software resource requirement and software quality. The subject also includes the quantitative aspects of quality control and assurance - and this covers activities like recording and monitoring faults during development and testing*".

Therefore, software metric is an attributes of software or its specifications. A measure is the value of this attribute. Example of software metrics are lines of source code (LOC) and faults per line of source code. They have long been studied as a way to evaluate the quality of large software systems [41]. Software metrics are used when a certain property of a software system is of interest, a set of one or more metrics are then measured to characterize that property.

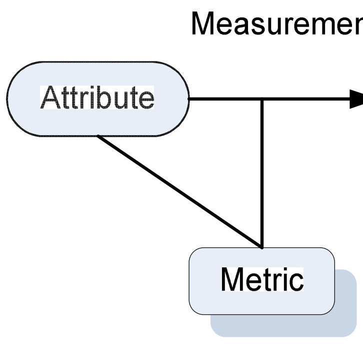


Figure 1: Description of software metrics

Figure 1 explains the term metric. When an attribute is measured, the result is a value. In this process, a metric is defined as the attribute and the applied method of carrying out the measurement. The metrics describes what the attribute is assume to be measured and how the measuring is carrying out, e.g. manually or automatically [13].

The metrics for OOD focus on measurements that are applied to the class, package and design characteristics. These measurements permit designers to assess the software early in process, making changes that will reduce complexity and improve the continuing capability of the design. Mens [42] states in that:*"Improving software quality, performance and productivity is a key objective for any organization that develops software. Quantitative measurements and software metrics in particular can help with this, since they provide a formal means to estimate software quality and complexity"*.

2.3 C&K Metrics Suite

The most commonly cited software metrics to be calculated for OOD are the so called C&K (Chidamber and Kemmerer) metric suite which was proposed by Chidamber and Kemerer [43]. This metric suite presents informative insight into whether developers are using OOD principles in their design [43]. Chidamber and Kemerer claim that using some of their metrics collectively helps managers and designers to make better design decision. C&K metrics have generated a significant amount of interest and are currently the most well known suite of measurements for OO software. C&K suite consists of six metrics that characterize complexity, inheritance, coupling and cohesion of software classes. Since these are class-level metrics, each metric will be aggregated at package-level. Total and average were used as aggregators so that metric values at class-level will be combined into single values at package level [15]. The following discussion presents these metrics [34, 43]:

- **Weighted Method per Class (WMC)**

  The WMC is the sum of the complexities of the methods implemented within a class. Complexity of a class can for example be calculated by the cyclomatic complexities (CC) of its methods. If the CC of each method is considered to be 1, then WMC will be the total number of methods in this class. High[2] value of WMC means the class is more complex than that of low values. Thus, class with

[2] The terms “high” and “low” in this study will be used to indicate the magnitude of the metrics value. It is up to the developer to find a proper threshold value of the current system, since there is no empirically found one. The developer have to take in consider the specific properties of the system.

less WMC is more desirable. WMC metric is redefined at the package-level (total and average) as follow:

- **Total WMC in a Package (TWMP)**

  TWMP is the total number of methods in a package, assuming that all complexities are considered to be unity. Therefore,

  $$\text{TWMP} = \sum_{\text{i=1}}^{\text{TC}} WMC_i$$

  Where the summation occurs over i =1 to TC. TC is defined as Total number of Classes in a package. For example, if *Package A* has 40 classes and each one has 10 methods then, TWMP = $\sum_{\text{i=1}}^{40} WMC_i$ = 40 * 10 = 400.

- **Average Weighted Methods per Package (AWMP)**

  $$\text{AWMP} = \frac{\text{TWMP}}{\text{TC}}$$

  So, if TWMP for a package that has 20 classes is 100, then AWMP= 100 / 20 = 5 which is the average number of methods for each class in that package.

o **Coupling Between Object (CBO)**

CBO for a class is a count of the number of other classes to which it is coupled. A class is coupled to another if methods of one class use methods or attributes of the other, or vice versa. In other words, CBO is total number of classes that a class referenced plus the number of classes that referenced the class (but not counted twice). An increase of CBO indicates the reusability of a class will decrease. Consequently, the CBO values for each class should be kept as low as possible. CBO will be redefined at the package-level as follow:

- **Total CBO in a Package (TCBO)**

$$\text{TCBO} = \sum_{\text{i=1}}^{\text{TC}} CBO_i$$

Where the summation occurs over i=1 to TC. TC is defined as Total number of Classes in a package. As an example, consider Package *A* in Figure 2. It consists of 5 classes. The total references (CBO) for *Class*1 is 3 since there are three classes coupled to *Class*1 (i.e. *Class*2 *Class*3 and *Class*4). CBO for *Class*2 is 2 since it is coupled to two classes (i.e. *Class*1 and *Class*4). Similarly, CBO for *Class*3 and *Class*5 is 2. CBO for *Class4* is 3. Therefore, TCBO for Package *A* = $\sum_{\text{i=1}}^{5} CBO_i$ = 3 + 2 + 2 + 2 + 3 = 12.

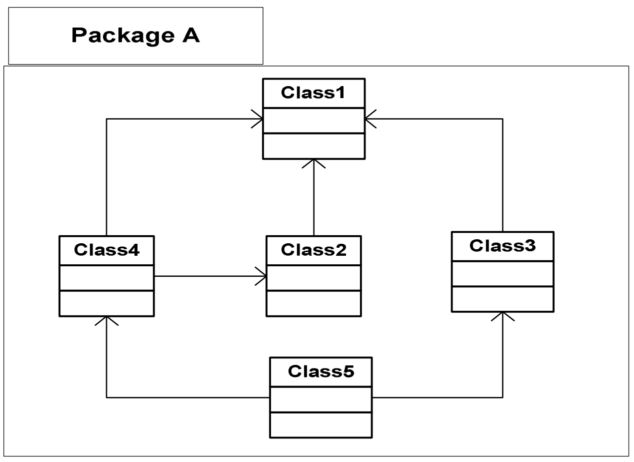


Figure 2: References in *Package* A

- **Average CBO (ACBO)**

  ACBO = $\frac{\text{TCBO}}{\text{TC}}$ in the previous example, ACBO for *Package A* = 12 / 5 = 2.4.

- **Lack of Cohesion of Methods (LCOM)**

This metric measures the correlation between the methods and the local instance variables of a class; high cohesion points to good class subdivision. Sellers [44] built mathematical and normalized definition for LCOM based on C&K suite. Sellers calculate LCOM as follow [44]:

LCOM = (M – sum (MF) / F) (M-1) Where

- M is the number of Methods in class
- F is the number of instance Fields in the class.
- MF is the number of Methods of the class accessing a particular instance Field.
- Sum (MF) is the sum of MF over all instance fields of the class.

To illustrate this consider Class *Car* and *Truck* in Figure 3 and 4. Class *Car* has 2 methods and 3 instance fields (i.e. M = 2 and F=3). Both of *turnRight* and *turnLift* methods access all instance fields (x, y and z) in the *Car* Class. Hence, MF = 3 + 3 = 6. Accordingly, LCOM for class *Car* = (2 - $\frac{6}{3}$) (2 - 1) = 0, which indicates that Class *Car* is completely cohesive (since all its methods use all its instance fields).

On the other hand, Class *Truck* in Figure 4 has 2 methods and 3 instance fields. Method *turnRight* access all instance fields (x, y and z). However, Method *turnLift* access only 1 instance field (only z). Therefore, MF = 3 + 1 = 4. Consequently, LCOM for Class *Truck* = $(2 - \frac{4}{3})\ (2 - 1) = 0.66$.

```
public class Car {
      int x, y, z;
      public void turnRifght() {
      x = 10;
      y = 20;
      z = x + y;
      return z;
                               }

      public void turnLift() {
      x = -10;
      y = -20;
      z = x + y;
      return z;              }
                  }
```

Figure 3: Class Car

```
public class Truck{

      int x, y, z;

      public void turnRifght() {

      x = 10;

       y = 20;

      z = x + y;

      return z,

                        }

      public void turnLift() {

      z = -4;

      return z;

                                 }

                        }
```

Figure 4: Class Truck

LCOM takes its values in the range [0-2] where values bigger than 1 should be considered alarming. LCOM increases complexity. Hence, classes with high LCOM (i.e. low cohesion) could probably be subdivided into two or more subclasses with increased cohesion. LCOM is aggregated at package-level as follow:

- **Total LCOM in a Package (TLCOM)**

$$\text{TLCOM}=\sum_{\text{i}=1}^{\text{TC}} LCOM_i$$

Where the summation occurs over i=1 to TC. TC is defined as Total number of Classes in a package.

For example, suppose that Package *Vehicle* consists of two classes: *Car* and *Truck,* shown in Figure 3 and 4. Then, TLCOM for Package *Vehicles:*

$$\text{TLCOM } (Vehicle) = \sum_{i=1}^{2} LCOM_i = LCOM_{Car} + LCOM_{Truck}$$

$$= 0 + 0.66 = 0.66$$

- **Average LCOM in a Package (ALCOM)**

$$\text{ALCOM} = \frac{\text{TLCOM}}{\text{TC}}$$

- **Response for a Class (RFC)**

RFC is the number of methods that can be invoked in response to a message in a class. It is calculated by adding the number of methods in the class (not including inherited methods) plus the number of distinct method calls made by the methods in the class. For example, Class *C1 in* Figure 5 has 2 Methods (*method1C1* and *method2C1*). Method *method1C1* calls Method *method1C2* in Class *C2* and Method method2C1 calls Method method1C3 in Class C3. Therefore, RFC for *C1* = 2 + 1 + 1 = 4.

```
public class C1 {

public void method1C1(C2 aC2) {

return aC2.method1C2();               }

public void method2C1(C3 aC3) {

return aC3.method1C3();      }

                   }
```

Figure 5: Class C1

When RFC increases, the overall design complexity of the class will increase and becomes hard to understand. RFC will be modified here to be at the package-level as follow:

- **Total RFC in a Package (TRFC):**

  $$\text{TRFC}= \sum_{\text{i=1}}^{\text{TC}} RFC_i$$

  Where the summation occurs over i=1 to TC. TC is defined as Total number of Classes in a package.

- **Average RFC in a Package (ARFC)**

  $$\text{ARFC} = \frac{\text{TRFC}}{\text{TC}}$$

o **Depth of Inheritance Tree (DIT)**

The DIT measures the level for a class within its class hierarchy. DIT metric is the length of the maximum path from the node to the root of the tree. As a result, this metric determined how far down a class is declared in the inheritance hierarchy. DIT will be modified here to be at the package-level as follow:

- **Total DIT in a Package (TDIT)**

$$\text{TDIT}= \sum_{\text{i=1}}^{\text{TC}} DIT_i$$

Where the summation occurs over i=1 to TC. TC is defined as Total number of Classes in a package. For example, Package *A* in figure 6, contains Class *Class1* which has 4 subclasses and Class *Class6* which has 3 subclasses (direct and indirect subclasses). Since *Class1* and *Class6* are root classes, the DIT for both of them are 0. *Class2, Class3* and *Class4* directly inherit *Class1*. Therefore, DIT for all of them is 1. Similarly, DIT for *Class7 and Class9 are 1. Since* the maximum inheritance path from *class5* to the root class is 2 (from *Class5* to *Class1*), DIT for *Class8* is 2. Similarly, DIT for *Class6* is 2. Thus, TDIT for Package *A* = $\sum_{\mathbf{i=1}}^{\mathbf{9}} DIT_i$ = 0 + 1 + 1 + 1 + 2 + 0 + 1 + 1 + 2 = 9.

- **Average DIT in a Package (ADIT)**

$$\text{ADIT}= \frac{\text{TDIT}}{\text{TC}}$$

In the previous example, TC in Package *A* is 9. Hence, ADIT for *Package A* = 9 / 9 = 1.

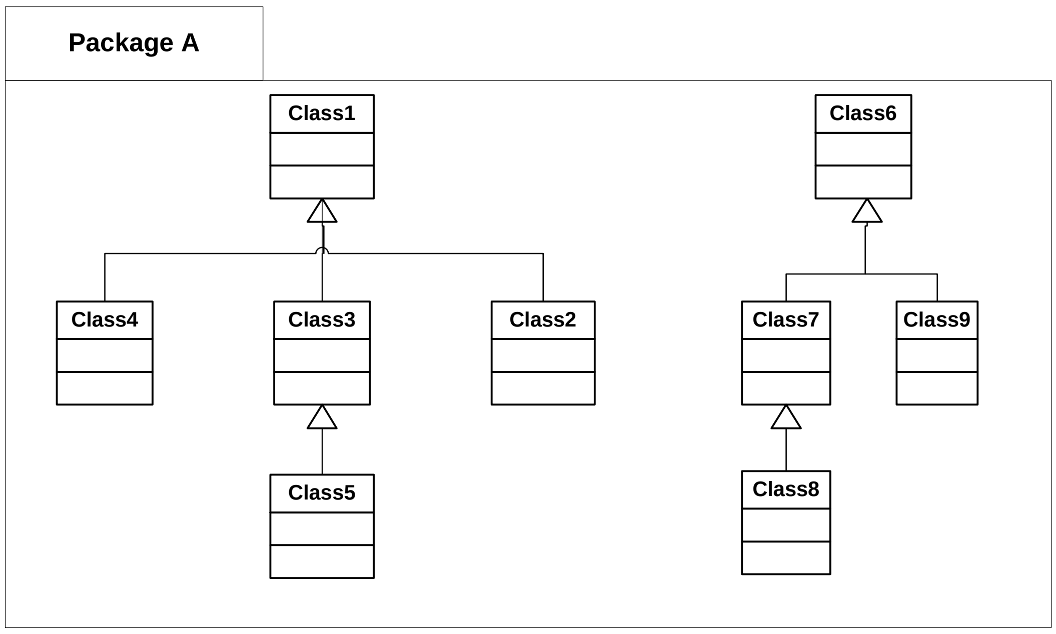


Figure 6: Example of inheritance in Package A

- **Number of Children (NOC)**

  NOC measures how many immediate subclasses are going to inherit the methods of the parent class. For example, NOC for *Class1* in Figure 6 is 3 since there are 3 immediate subclasses (*Class2, Class3 and Class4*) that inherit the methods in *Class1.* Similarly, NOC for *Class6* is 2 since there are only 2 immediate subclasses (*Class7 and Class9*) inherit the methods in *Class6.* In the same way, NOC for *Class3* is 1. However, NOC for *Class4* is 0 as there are no immediate subclasses for *Class4.* The size of NOC can be utilized to show the level of reuse in a system. For example, If NOC raises it means reuse increases. Alternatively, as NOC increases, the amount of testing will also increase as more children in a class show more responsibility. Thus, NOC correspond to the effort required to test the class and reuse. NOC will be aggregated and averaged at the package-level as follow:

- **Total NOC in a Package (TNOC):**

$$\text{TNOC} = \sum_{\text{i=1}}^{\text{TC}} NOC_i$$

Where the summation occurs over i=1 to TC. TC is defined as Total number of Classes in a package. In Figure 6, TNOC for Package *A* =

$$\sum_{\text{i=1}}^{9} NOC_{classi} = 3 + 0 + 1 + 0 + 0 + 2 + 1 + 0 + 0 = 7.$$

- **Average NOC in a Package (ANOC)**

$$\text{ANOCP} = \frac{\text{TNOC}}{\text{TC}}$$

ANOC for Package *A* = 7 / 9 =0.77.

2.4 MOOD Metrics suite

Abreu proposed a set of metrics for OOD called MOOD (Metrics for Object-Oriented Design) [39]. MOOD denotes a basic structural mechanism of the object-oriented paradigm as encapsulation, inheritance, polymorphism, and message passing at system-level. This section will discuss these metrics and show how they can be redefined here to be at the package-level.

- **Method Hiding Factor (MHF)**

  Abreu et al. [39] defined MHF metric as the sum of the invisibilities of all methods in all classes. The invisibility of a method is the percentage of the total class from which the method is hidden. Thus, MHF is proposed as a measure of information hiding for a system. It is calculated as follow [39]:

$$MHF = \frac{\sum_{i=1}^{N} M_h(C_i)}{\sum_{i=1}^{N} M_d(C_i)}$$

Here, $M_d(C_i) = M_v(C_i) + M_h(C_i)$

$M_d(C_i)$ = the number of Methods defined in class C_i.

$M_v(C_i)$ = the number of Methods that visible in class C_i.

$M_h(C_i)$ = the number of Methods hidden in C_i.

Where the summation occurs over i = 1 to N. N is defined as total Number of classes in a system. MHF at package-level (MHFP) is calculated as follow:

$$\text{MHFP} = \frac{\sum_{i=1}^{TC} M_h(C_i)}{\sum_{i=1}^{TC} M_d(C_i)}$$

Here, $M_d(C_i) = M_v(C_i) + M_h(C_i)$

$M_d(C_i)$ = the number of Methods defined in class C_i.

$M_v(C_i)$ = the number of Methods that visible in the class C_i.

$M_h(C_i)$ = the number of Methods hidden in C_i.

Where the summation occurs over i = 1 to TC. TC is defined as Total number of Classes in a package.

To illustrate this, suppose that we have a Package called *Animal* that contains two Classes: *Cat* and *WildCat* (shown in Figure 7). Class *Cat* consists of 6 Methods (2 public and 4 private). Hence, $M_d(Cat) = 6$, $M_v(Cat) = 2$ and $M_h(Cat) = 4$. Class *WildCat* contains 4 Methods (2 public and 2 private), which means that $M_d(WildCat) = 4$ and $M_v(WildCat) = M_h(WildCat) = 2$. Therefore, MHFP for Package *Animal* = ((4+2) / (6+4) = 0.6.

If the value of MHFP is (100%), it means all methods are private which indicates very little functionality. Thus it is not possible to reuse methods with high MHFP. MHFP with (0%) value indicate all methods are public that means most of the methods are unprotected.

- **Attribute Hiding Factor (AHF)**

  AHF measures the invisibilities of attributes in all classes. The invisibility of an attribute is the percentage of the sum of all classes in a system from which the attribute is not visible (hidden). They can be kept from being accessed by other objects by being declared to be not visible (i.e. private in Java). An attribute is identified as visible if it can be accessed by another class or object. AHF is defined as follow:

$$\text{AHF} = \frac{\sum_{i=1}^{N} A_h(C_i)}{\sum_{i=1}^{N} A_d(C_i)}$$

Here, $A_d(C_i) = A_v(C_i) + A_h(C_i)$

$A_d(C_i)$ = the number of Attributes defined in class C_i.

$A_v(C_i)$ = the number of Attributes that visible in the class C_i.

$A_h(C_i)$ = the number of Attributes hidden in C_i.

Where the summation occurs over i=1 to N. N is defined as total Number of classes in a system. AHF at package-level (AHFP) is calculated as follow:

$$\text{AHFP} = \frac{\sum_{\text{i=1}}^{\text{TC}} A_h(C_i)}{\sum_{i=1}^{TC} A_d(C_i)}$$

Here, $A_d(C_i) = A_v(C_i) + A_h(C_i)$

$A_d(C_i)$ = the number of Attributes defined in class C_i.

$A_v(C_i)$ = the number of Attributes that visible in the class C_i.

$A_h(C_i)$ = the number of Attributes hidden in C_i.

Where the summation occurs over i=1 to TC. TC is defined as Total number of Classes in a package.

Class *Cat* in the previous example (Figure 7) has 9 attributes (3 public and 6 private). On the other hand, Class *WildCat* has 6 attributes (3 public and 3 private). Therefore, AHFP for the *Animal* Package = ((3+6) / (9+6)) = 0.6.

If the value of AHFP is (100%), it means all attributes are private. AHFP with (0%) value indicates all attributes are public.

```
Class Cat   {

Private int a, b, c;

Private string d, e, f;

public  double j,h,i;

public void turnRight() {…..}

public void turnLeft() {…..}

private void method1() {…..}

private void method2() {…..}

private void method3() {…..}

private void method4(){…..}
            }
Class WildCat extends Cat {

//Inherits turnRight and turnLeft. Inherits j, h and i attributes.

private int j, k,l;

public string m,n,o;

public void move() {……}

public void increaseSpeed() {……}

private void method5() {…}

private void method6() {…}
                           }
```

Figure 7: Classes *Cat* and WildCat

- **Method Inheritance Factor (MIF):**

MIF is proposed as a measure of level of reuse in all classes of the system. It is defined as the ratio of the sum of the inherited methods in all classes of the system as follow:

$$\text{MIF}=\frac{\sum_{\text{i}=1}^{\text{N}} M_i(C_i)}{\sum_{i=1}^{\text{N}} M_a(C_i)}$$

Here, $M_a(C_i) = M_d(C_i) + Mi(C_i)$

$Mi(C_i)$ = inherited Methods in C_i.

$M_d(C_i)$ = the number of Methods defined in C_i.

Where the summation occurs over i=1 to N. N is defined as total Number of classes in a system. MIF is calculated at package-level (MIFP) as follow:

$$\text{MIFP}=\frac{\sum_{\text{i}=1}^{\text{TC}} M_i(C_i)}{\sum_{i=1}^{TC} M_a(C_i)}$$

Here, $M_a(C_i) = M_d(C_i) + Mi(C_i)$

$Mi(C_i)$ = inherited Methods in C_i.

$M_d(C_i)$ = the number of Methods defined in C_i.

Where the summation occurs over i=1 to TC. TC is defined as Total number of Classes in a package.

In Figure 7, the *WildCat* Class extends the *Cat* Class. Hence, it has access to all public methods in class *Cat*. Accordingly, $M_i(WildCat) = 2$. However, Class *Cat* does not inherit any methods, thus $M_i(Cat) = 0$. Since total number of methods (inherited and defined) in Package *Animal* is 10, MIFP for the *Animal* package = (2+0) / (6+4) = 0.2.

Since the methods available in a leaf class (as in Class *Cat* in the previous example), $M_a(\text{Leaf\_Class})$ are not inheritable, the MIFP denominator, by including leaf classes in the $M_a(C_i)$ sum does not represent the maximum possible inheritance. It is actually represents a value greater than the maximum. Therefore, the value for MIFP, for any package, can never be 1. When this value is close to 1, it indicates superfluous inheritance or too wide member scopes. However, if the value of MIFP is (0%), it means that there is no method exists in the class as well as the class is lacking an inheritance statement.

- **Attribute Inheritance Factor (AIF)**

  Similar to MIF, AIF is defined as the ratio of the sum of inherited attributes in all classes of the system. AIF denominator is the sum of available attributes for all classes In other words, AIF is defined as follows:

$$\text{AIF} = \frac{\sum_{i=1}^{N} A_i(C_i)}{\sum_{i=1}^{N} A_a(C_i)}$$

Here, $A_a(C_i) = A_d(C_i) + A_i(C_i)$

$A_i(C_i)$ = inherited Attributes in C_i.

$A_d(C_i)$ = the number of Attributes defined in C_i.

Where the summation occurs over i=1 to N. N is defined as Total number of Classes a system. AIF at Package-level (AIFP) is calculated as follow:

$$\text{AIFP}=\frac{\sum_{i=1}^{TC} A_i(C_i)}{\sum_{i=1}^{TC} A_a(C_i)}$$

Here, $A_a(C_i) = A_d(C_i) + A_i(C_i)$

$A_d(C_i)$ = inherited Attributes in C_i.

$A_i(C_i)$ = the number of Attributes defined in C_i.

Where the summation occurs over i=1 to TC. TC is defined as Total number of Classes in a package.

Since j, h, I in Class *Cat* (in Figure 7) will be accessed by *WildCat*, $A_i(Wildcat)$ = 3. However, $A_i(Cat)$ = 0 as it does not inherit any attribute. The total number of attributes in Class *WildCat* is 6 and 9 in Class *Cat.* Thus, AHFP for the *Animal* Package = (3 + 0) / (6 + 9) = 0.2. When this value is close to 1, it indicates superfluous inheritance or too wide member scopes. However, If the value of AIFP is low (0%), it means that there is no attribute exists in the class as well as the class lacking an inheritance statement.

- **Polymorphism Factor (PF)**

Polymorphism is an important characteristic in OOD. It measures the degree of overriding in the class inheritance tree. It corresponds to the number of actual method overrides divided by the maximum number of possible method overrides. So, PF is defined as follows:

$$\text{PF}=\frac{\sum_{i=1}^{N} M_o(C_i)}{\sum_{i=1}^{N}\left[M_o(C_i)\times DC(C_i)\right]}$$

Where

$M_o(C_i)$ =the number of overriding Methods in class C_i.

$M_n(C_i)$ = the number of new Methods in C_i.

DC (C_i) = the Descendants count in Class (C_i). N is defined as total Number of classes in a system.

Polymorphism Factor at Package-level (PFP) measures the degree of method overriding in the class inheritance tree. It equals the number of actual method overrides divided by the maximum number of possible method overrides.

$$\text{PFP} = \frac{\sum_{j=1}^{TC} M_o(C_i)}{\sum_{j=1}^{TC}\left[M_{n_o}(C_i)\times DC(C_i)\right]}$$ Where

$M_o(C_i)$ = overriding Methods in class C_i.

$M_n(C_i)$ = new Methods in C_i.

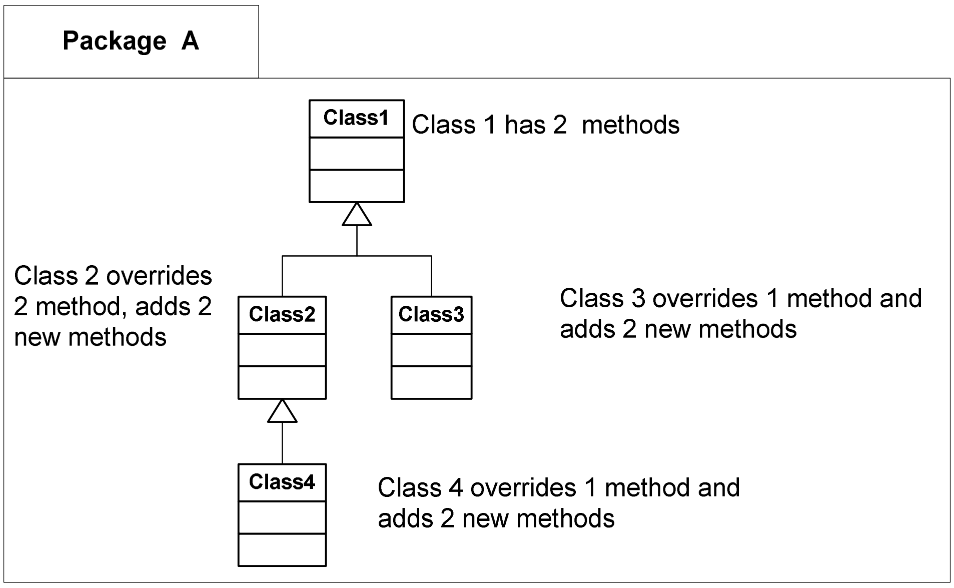
DC (C_i) = number of Descendants of Class C_i.

TC is defined as Total number of Classes in a package.

To illustrate this, consider Package *A* in Figure 8. It has 4 classes that contain 8 new methods and 4 overriding methods. The number of descendants of

Class1 is 3 (i.e. *Class3, Class2 and Class4*). *Class2* has one descendant while *Class3* and *Class4* have nothing. Thus, PFP (A) =

$$\frac{2+1+1}{(2\times 3)+(2\times 1)+(2\times 0)+(2\times 0)} = 0.5$$



Package A
Class1
Class 1 has 2 methods
Class2
Class3
Class 2 overrides 2 method, adds 2 new methods
Class 3 overrides 1 method and adds 2 new methods
Class4
Class 4 overrides 1 method and adds 2 new methods

Figure 8: Classes in *Package* A

The value of PFP can differ between 0% and 100%. If a package has 0% PFP, it indicates the package uses no polymorphism and 100% PFP indicates that all methods are overridden in all derived classes.

- **Coupling Factor (CF)**

Coupling shows the relationship between classes. The CF is defined as the ratio of the actual non-inheritance number of coupling to the maximum possible non-inheritance number of couplings in the system to. In other words, CF is:

$$\text{CF}=\frac{\sum_{i=1}^{N}\left\lfloor\sum_{j=1}^{N} is\_client(C_i, C_j)\right\rfloor}{N^2 - N}$$

Where

$is\_client(C_i,C_j)$ = 1 if $(C_c \rightarrow C_s) \wedge (C_c \neq C_s)$

0 otherwise.

Where the summation occurs over i=1 to N. N is defined as Total number of Classes in the system. The client-supplier relation, represented by $(C_c \rightarrow C_s)$, means that C_c (client Class) contains at least one non inheritance reference to a feature (method or attribute) of class C_s (supplier Class). Coupling Factor at Package-level (CFP) is calculated as follow:

$$\text{CFP} = \frac{\sum_{i=1}^{TC}\left\lfloor\sum_{j=1}^{TC} is\_client(C_i, C_j)\right\rfloor}{TC^2 - TC}$$

Where

$is\_client(C_i,C_j)$ = 1 if $(C_c \rightarrow C_s) \wedge (C_c \neq C_s)$

0 otherwise.

The summation occurs over i=1 to TC. TC is defined as Total number of Classes in a package.

The client-supplier relation, represented by $(C_c \rightarrow C_s)$, means that C_c (client class) contains at least one reference to a feature (method or attribute) of class C_s (supplier class). For example, Package *A* in Figure 2 consists of 5 classes. The client-supplier relations of these classes are as follow:

Class1 does not have any reference to any class. (i.e. $(C_c \rightarrow C_s)$, = 0)

$(Class2_c \rightarrow Class1_s)$ = 1.

$(Class3_c \rightarrow Class1_s)$ = 1.

$(Class4_c \rightarrow Class2_s)$ = 1 & $(Class4_c \rightarrow Class1_s)$ =1.

$(Class5_c \rightarrow Class3_s)$ = 1 & $(Class5_c \rightarrow Class4_s)$ = 1.

Hence, CFP for Package *A* will be as follow: CFP (A) = $\frac{0+1+1+2+2}{5^2-5}$ = 0.3

2.5 Martin Metrics

Martin has proposed a set of package-level metrics that are related to stability, dependencies and abstraction [12] in OOD. The stability of a package is related to the amount of work required to make a change to that package. Martin presents the Stable Dependencies Principle (SDP) for such volatile packages. The SDP says that *"the dependencies between packages in a design should be in the direction of stability. A Package should only depend upon packages that are more stable than it is"*. Martin used the number of dependencies that enter and leave that package to measure the Instability (I) of a package. Three metrics have been identified [12]:

- **Afferent Couplings (Ca**): The number of classes outside this package that depend upon classes inside this package.
- **Efferent Couplings (Ce)**: The number of classes inside this package that depend upon classes outside this package.

- **Instability (I)**: (Ce ÷ (Ca + Ce): This metric has the range [0, 1]. Where I = 0 indicates a maximally stable package and I = 1 indicates a maximally Instable package. In other words, when "I" is one it means that no other package depends upon this package and when the "I" metric is zero it means that the package is depended upon by other packages, but doesn't itself depend upon any other packages.

Martin thinks that if classes are abstract, then they are highly stable since they depend upon nothing and are depended upon by others that extend the abstract classes. Martin states that if a package (or category[3]) is happening to be stable, it should consist of abstract classes so that it can be extended. He also said that: "*Stable packages that are extensible are flexible and do not constrain the design. If stable packages should be highly abstract, one might infer that Instable packages should be highly concrete. In fact, this stands to reason. An abstract category must have dependents since there must be classes, outside the abstract category, that inherit from it and implement the missing pure interfaces. However, we do not want to encourage dependencies upon Instable categories. Thus, Instable categories should not be abstract, they should be concrete*". Based on this, Martin defines a metric which measures the "Abstractness" of a package as follows:

[3] The term *category* is usually used as a synonym for *package.*

- **Abstractness (A):** Number of Abstract classes in a package (NOA) divided by total number of classes in this package. This metric range is [0, 1]. 0 means concrete and 1 means completely abstract.

He also introduces the stable abstraction principle (SAP). SAP states that a package should be as abstract as it is stable [12]. Using the Abstraction and the Instability, Martin proposed another metric called Normalized Distance (D):

- **Normalized Distance (D)**: |A + I - 1|.

This is used as indicator for packages with a high “D” (near to 1) are candidates for reexamining and restructuring as they are either not abstract enough or they depend too much on other classes.

To understand Martin metrics, consider the following example in Figure 9. Figure 9 shows the dependencies between Packages A, B, C, D and E. The dashed arrows between the Packages represent packages dependencies. The relationships between the classes of those packages are shown in Figure 10. It shows how those dependencies are actually implemented. There are inheritance, aggregation, and association relationship.

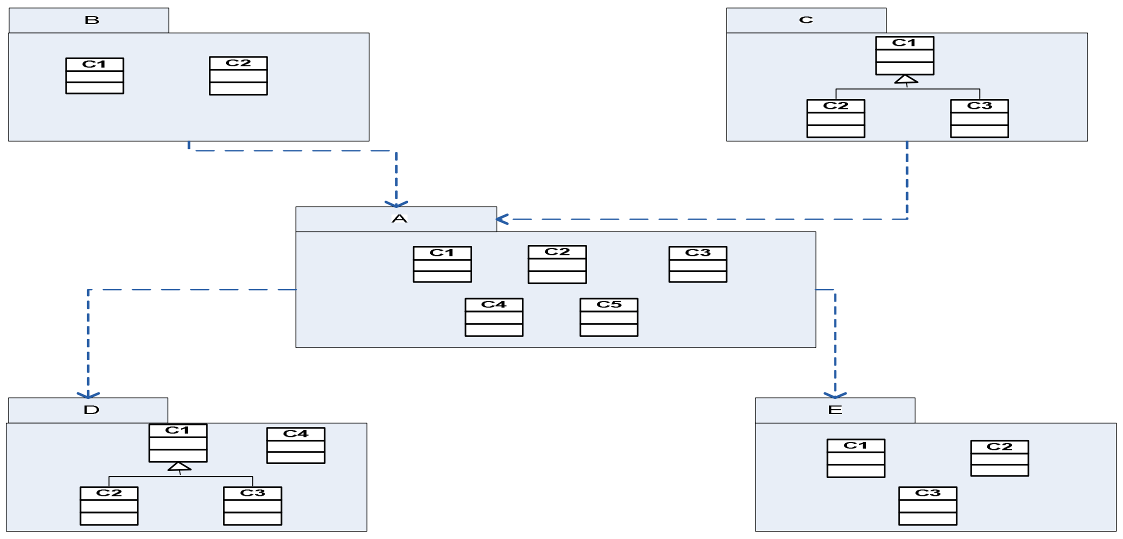


Figure 9: Dependencies between Packages A, B, C, D and E

Now, suppose that we want to calculate Ca, Ce, "I", A and D of Package *A*. These are calculated as follow:

- Ca: Since the number of external classes coupled to classes in Package *A* due to outgoing coupling is 3, then Ca = 3.
- Ce: The number of classes inside Package *A* that depend upon classes outside this package are 5. Hence, Ce = 5.
- "I": Instability is (Ce ÷ (Ca + Ce)) =0.625. "I" is an indicator of the package's resilience to change which is here 62%.
- "A": Abstractness of Package *A* is the ratio of the number of abstract classes (NOA) to the total number of classes in this package. Package *A*

contains one abstract Class (C2) and 4 concrete Classes (C1, C3, C4 and C5). Therefore, Abstractness of Package *A* = 1 / (4 + 1) = 0.20.

- "D": Normalized Distance is |(A + I - 1)|, therefore, D for Package *A* = 0.2 + 0.625 - 1= 0.175. According to Martin this package does not have to be restructured since its value near to 0.

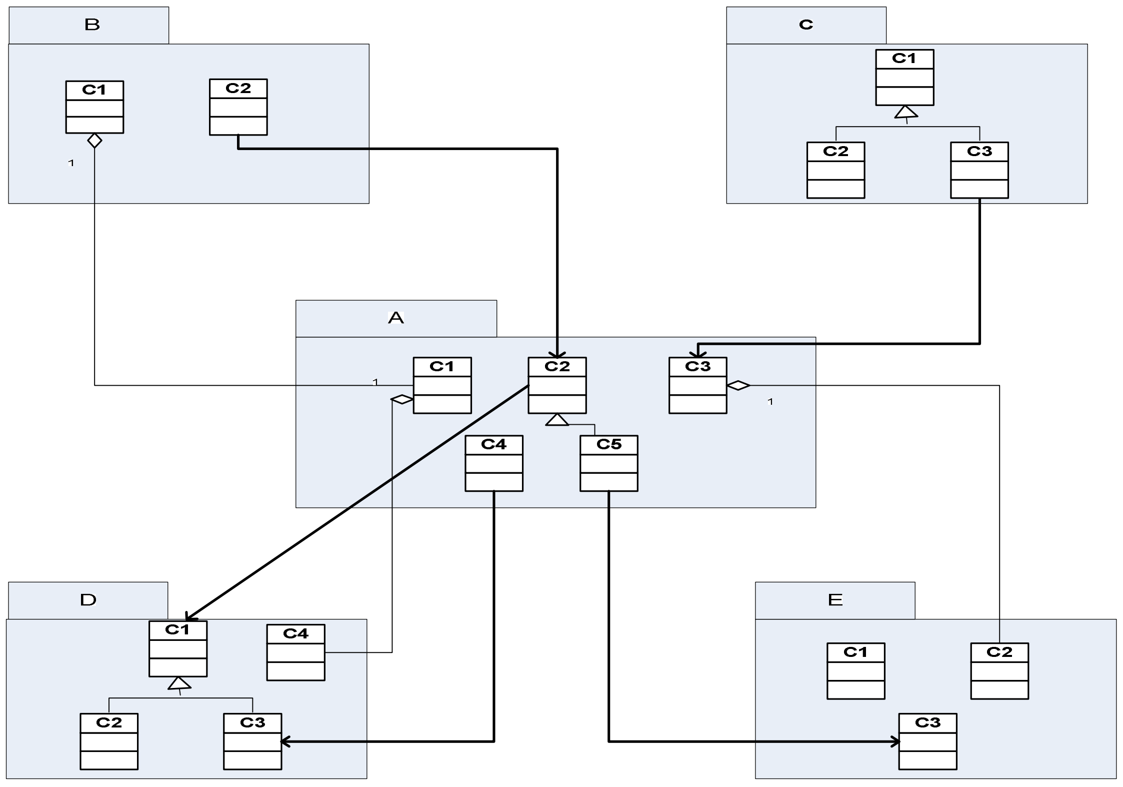


Figure 10: Relationship between Packages A, B, C, D and E

Package	Ce	Ca	I	A	D
A	5	3	0.625	0.2	0.175
B	2	0	1	0	0
C	1	0	1	0.33	0.33
D	0	3	0	0.25	0.75
E	0	2	0	0	1

Table 1: Ca, Ce, "I", A and D values for packages in Figure 10.

Chapter 3

Empirical Study 1: Faults and Package Metrics

3.1 Goal

The goal of this empirical study can be defined, using the GQM template [45], as follows: investigate the significant correlation between faulty packages and the sets of package-level metrics for the purpose of identifying and characterizing faulty packages from the point of view of researchers and practitioners in the context of OO software.

3.2 Hypotheses

The following hypotheses will be investigated in this research. The metric m, used in the below hypotheses, ranges over the set of package metrics under investigation in this research.

Hypothesis 1

- H0-PreFD (Null Hypothesis): There is no significant correlation between metric *m* and the Pre-release Fault Density (PreFD) of a package in a system.
- H1-PreFD (Alternative Hypothesis): There is significant correlation between metric *m* and the Pre-release Fault Density of a package in a system.

Hypothesis 2

- H0-PostFD (Null Hypothesis): There is no significant correlation between metric *m* and the Post-release Fault Density (PostFD) of a package in a system.

- H1-PostFD (Alternative Hypothesis): There is significant correlation between metric *m* and the Post-release Fault Density of a package in a system.

Hypothesis 3

- H0-PPreF (Null Hypothesis): There is no significant correlation between metric *m* and the Presence of a Pre-release Fault in a package (PPreF).
- H1-PPreF (Alternative Hypothesis): There is significant correlation between metric *m* and the Presence of a Pre-release Fault in a package.

Hypothesis 4

- H0-PPostF (Null Hypothesis): There is no significant correlation between metric *m* and the Presence of a Post-release Fault in a package (PPostF).
- H1-PPostF (Alternative Hypothesis): There is significant correlation between metric *m* and the Presence of a Post-release Fault in a package.

3.3 Subject

The subject of this empirical study is Eclipse system. Eclipse was selected for two reasons:

- Eclipse is well known system that is used in many researches and universities [46].
- Faults data for Eclipse releases were publically available [47].

Eclipse is an open source Integrated Development Environments (IDE) developed by IBM. It is written mainly in Java and can be used to create diverse end-to-end computing solutions for multiple execution environments. The platform consists of open source software components that tool vendors use to build solutions that plug in to integrated software workbenches [47]. The Eclipse platform incorporates technology expressed through a well-defined design and implementation framework. It includes extensible frameworks, tools and runtimes for building, deploying and managing software across the lifecycle. Great number of research organizations, major technology dealers, several universities, and individual researchers extended hands to support the Eclipse platform [48]. Eclipse releases 2.0, 2.1 and 3.0 were selected to be subjects for this study as the fault data were only available for those releases. Table 2 gives a brief description about these releases. It shows that release 3.0 has the maximum number of packages, classes, methods, LOC and faults whereas release 2.0 has the lowest.

Release	**Date**	**# of Packages**	**# of Classes**	**# of Methods**	**LOC**	**# of Pre-Release Faults**	**# of Post-Release Faults**
Eclipse 2.0	27-6-2002	378	7688	77617	524388	4297	917
Eclipse 2.1	27-3-2003	428	9243	94111	647864	2977	662
Eclipse 3.0	25-6-2004	645	12642	125532	870435	4579	1511

Table 2: Summary of Eclipse system

3.4 Experimental Variables

The independent variables are the metrics under investigation. The dependent variables in this study are:

- Pre- release Fault Density for a package (PreFD):

  The following formula was used for measuring the pre-release fault density for each package in a release i:

  $$\text{PreFD} = \frac{\text{Number of Pre - release Faults for a packge}}{\text{Number of LOC in this package}}$$

- Post- release Fault Density for a package (PostFD):

  PostFD for each package in a release i is calculated as follow:

  $$\text{PostFD} = \frac{\text{Number of Post - release Faults for a packge}}{\text{Number of LOC in this package}}$$

- Binary variable indicates the Presence of a Pre-release Fault in a package (PPreF).

- Binary variable indicates the Presence of a Post-release Fault in a package (PPostF).

3.5 Tool

The tool that was used to collect the package metrics in this study is JHawk [47]. JHawk is Eclipse plug-in supplied by Virtual Machinery. It measures various metrics at a system's package, class or method level on Java code. JHawk doesn't have any graphical view of the metrics, only text, but it is possible to export the results of an analyzed system to .xml or .csv files.

JHawk was used in this study to collect Martin metrics. In addition, this tool was extended to automate the extraction of C&K and MOOD metrics. MOOD and C&K metrics were redefined and implemented at the package-level by aggregating them. The implementation procedure used to extend this tool was as follow:

1- Calculating C&K and MOOD metrics at the package-level by adding new methods in the following classes:

- JavaParserMethodRecord: it has methods used to calculate metrics at method-level.
- JavaParserClassRecord: it contains methods used to calculate metrics at class-level. This class uses methods in JavaParserMethodRecord to calculate the class metrics.
- JavaParserPackageRecord: it contains methods that calculate the metrics at package-level. C&K and MOOD metrics were aggregated and defined in this class by aggregating the methods in *JavaParserClassRecord* Class. For example, Method *getMHF()* was used to calculate MHFP in *JavaParserPackageRecord* Class. Method *getMHF()* calculate the total number of hidden methods in a package. It uses *getPrivateMethods()* Method in *JavaParserClassRecord* Class to obtain the number of hidden methods in a class. To do this, getPrivateMethods() uses a Boolean method called isPrivate() in *JavaParserMethodRecord* Class to check if a method is private or not.

2- Adding the created metrics to the existing metrics in JHawk at the package-level. This was done by adding these metrics to "*PACKAGE_METRIC_CODES*" Array in *JHawkPreferenceConstants* Class.

3- Finally, these metrics were displayed in Eclipse by adding them in *JHawkDefaultPreferences*. Header name, description, position and threshold level (i.e. warning level) for each metrics can be defined or changed using this class.

4- Sort the metrics output: *PackageSorter* Class was used to sort metrics results. The objective of this step is to give a user the option to sort the metrics output according to their values or to the package names. In this research, the metrics results were sorted by package names and then exported to CSV format, which will make the results more organized and easier to deal with.

3.6 Data Collection

JHawk was used to extract the package-level metrics from Eclipse 2.0, 2.1 and 3.0. The faults data were taken from Zimmermann’s work [15]. Then, the dependent variables were calculated according to section 3.4. After that, univariate regression [49] was carried out to determine if each individual independent variable is significantly correlated to the dependent variables. Finally, multivariate regression [49] was used to build models for fault prediction.

3.7 Results and Discussion

The following sections present descriptive statistic for the collected metrics, results of univariate, multivariate analysis and the regression models built for fault prediction.

3.7.1 Descriptive statistics

Tables 3, 4 and 5 present descriptive statistics (minimum, maximum, mean, median, standard deviation) for the collected metrics in Eclipse releases 2.0, 2.1 and 3.0. Tables 6, 7 and 8 show descriptive statistics for faults found in Eclipse packages. Low variance (standard deviation) measures do not differentiate packages very well and therefore are not likely to be helpful predictors in our dataset. Analyzing and presenting the distribution of measures is important for comparison of the results with replicated studies [50]. It allows researchers to determine if the data gathered across studies come from similar populations. If not, this information will probably be useful to explain different findings across studies.

3.7.1.1 Descriptive statistics for metrics results

The studied metrics can be categorized into coupling, inheritance, cohesion, visibility and polymorphism. The following can be observed from the values of these metrics:

Coupling metrics:

- The largest maximum value is for TRFC in release 2.1, which also has the largest mean and standard deviation (StdDev). This may be explained by the fact that TRFC is the only measure to count indirect coupling, while all other coupling measures count connections to directly coupled classes only.

- Coupling between packages (i.e. Ca and Ce) was minimum in release 2.0 and maximum in release 3.0. Ca and Ce also have largest mean and standard deviation in 3.0. This might be due to the fact that the greatest number of packages was in

Eclipse release 3.0 (645 packages) whereas the minimum number of packages (378) was in release 2.0.

- Coupling between classes in packages were sometimes maximum in release 3.0 (as in CFP and ACBO) and sometimes in release 2.1 (as in TRFC and ARFC). Similarly mean and standard deviation were either maximum in 2.1 or 3.0. This is might be due to the fact that release 2.1 has, on average, the greatest number of classes in each package whereas release 3.0 has the greatest number of packages in Eclipse releases.
- The lowest mean, maximum and standard deviation values were for CFP compared to other coupling metrics. This is because CFP measures actual direct coupling between classes in a package over maximum possible number of coupling between these classes. This is because the denominator (counting a maximum number of possible coupling) for CFP grows proportionally faster than the numerator (actual number of coupling).

Inheritance metrics:

Distributions of the values of the inheritance metrics show that inheritance has been used cautiously within the three releases (i.e. low standard deviation, mean and median values for ADIT and ANOC). However, there is sufficient variance in TDIT and TNDC to proceed with the analysis.

Size metrics:

The highest mean, maximum and standard deviation values for No. classes, TWMP and AWMP were in Release 2.1. However, this release has fewer packages than 3.0, but more abstract classes.

Stability metrics:

The descriptive statistics for the stability measures do not show any interesting or surprising trends. Instability (I) values were maximum in release 3.0. This is because "I" depend on the number of classes inside this package that depend upon other packages, which is maximum in release 3.0 (i.e. Ce mean is 11.3). However, D values are not maximum in release 3.0. This is because D depends on "A" which was minimum in 3.0.

Cohesion metrics

TLCOM has the third largest mean, maximum and standard deviation values in Tables 3, 4 and 5. ALCOM indicates the average LCOM for a class in Eclipse. ALCOM is around 2 in each class, which indicates low cohesion. This is because of the presence of access methods which are typically only reference one attributes, and consequently increase the number of pairs of methods in the class that do not use attributes in common.

Visibility

Hidden attribute was used a lot in Eclipse releases (mean is between 0.52 and 0.62). However, hidden method was used cautiously within these releases.

Polymorphism

Packages in Eclipses releases have low PFP (mean is between 1.26 and 0.138). PFP measures the degree of method overriding in the class inheritance tree. This may be explained by the fact that PFP depend on inheritance, which was low in Eclipse packages, as discussed above.

Category		Minimum	Maximum	Mean	Median	StdDev
	Martin Metrics					
Size	No. Classes	1	242	20.39	13	24.91
	NOA	0	76	4.39	2	7.59
Stability	A	0	1	0.23	0.13	0.28
	I	0	1	0.61	0.69	0.34
	D	0	1	0.23	0.13	0.25
Coupling	Ca	0	242	10.33	3	23.78
	Ce	0	49	10.39	8	9.54
	C&K Suite					
Cohesion	TLCOM	0	1667.6	39.80	9.81	135.75
	ALCOM	0	94.87	2.12	0.65	7.88
Coupling	TRFC	1	16756	532.10	293	1055.11
	ARFC	1	147	24.88	23.05	15.41
	TCBO	0	6171	188.30	89	429.86
	ACBO	0	292.67	8.97	6.86	17.96
Size	TWMP	1	7970	206.18	107	463.99
	AWMP	1	64.29	9.65	8.24	6.90
Inheritance	TDIT	0	179	3.98	1	13.07
	ADIT	0	1.36	0.12	0.05	0.18
	TNDC	0	950	11.59	2.5	52.28
	ANDC	0	8.9	0.40	0.17	0.85
	MOOD Suite					
Visibility	MHFP	0	0.7	0.13	0.09	0.09
	AHFP	0	1	0.53	0.6	0.35
Polymorphism	PFP	0	3.65	0.11	0.01	0.01
Coupling	CFP	0	0.5	0.03	0.01	0.01
Inheritance	MIFP	0	0.93	0.47	0.53	0.53
	AIFP	0	0.98	0.40	0.41	0.31

Table 3: Descriptive statistics for the proposed metrics in Eclipse 2.0

Category		Minimum	Maximum	Mean	Median	StdDev
	Martin Metrics					
Size	No. Classes	1	247	21.60	14	26.84
	NOA	0	85	4.35	2	7.80
Stability	A	0	1	0.22	0.12	0.28
	I	0	1	0.60	0.67	0.34
	D	0	1	0.22	0.13	0.24
Coupling	Ca	0	283	11	3	26.34
	Ce	0	53	11.08	8	10.13
	C&K Suite					
Cohesion	TLCOM	0	1782.6	41.81	10.37	138.33
	ALCOM	0	99.53	2.01	0.69	6.92
Coupling	TRFC	1	18092	586.70	302	1131.54
	ARFC	1	149	220.19	23.66	15.37
	TCBO	0	6360	206.79	88	457.93
	ACBO	0	349	9.39	6.88	20.69
Size	TWMP	1	8549	220.19	115	478.90
	AWMP	1	50	9.47	8.4	5.5
Inheritance	TDIT	0	178	3.83	1	12.41
	ADIT	0	1.36	0.11	0.06	0.16
	TNDC	0	997	11.83	2	52.41
	ANDC	0	10.8	0.39	0.18	0.89
	MOOD Suite					
Visibility	MHFP	0	0.7	0.14	0.11	0.14
	AHFP	0	1	0.55	0.63	0.35
Polymorphism	PFP	0	3.65	0.11	0.03	0.23
Coupling	CFP	0	0.5	0.03	0.01	0.07
Inheritance	MIFP	0	0.96	0.51	0.59	0.31
	AIFP	0	0.99	0.42	0.44	0.31

Table 4: Descriptive statistics for the proposed metrics in Eclipse 2.1

Category		Minimum	Maximum	Mean	Median	StdDev
	Martin Metrics					
Size	No. Classes	1	245	19.60	12	24.53
	NOA	0	95	4.18	2	7.66
Stability	A	0	1	0.24	0.12	0.30
	I	0	1	0.62	0.72	0.34
	D	0	1	0.20	0.12	0.23
Coupling	Ca	0	40.3	11.21	3	31.08
	Ce	0	65	11.30	8	10.70
	C&K Suite					
Cohesion	TLCOM	0	2064.6	34.33	8.76	117.68
	ALCOM	0	113.2	1.92	0.65	7.50
Coupling	TRFC	0	17283	527.32	252	990.28
	ARFC	0	149	24.38	22.54	15.84
	TCBO	0	5981	181.80	69	428.23
	ACBO	0	456	8.94	5.94	24.22
Size	TWMP	0	7907	194.93	96	400.80
	AWMP	0	40	8.90	8	5.50
Inheritance	TDIT	0	148	3.05	1	10.06
	ADIT	0	1.45	0.10	0.04	0.15
	TNDC	0	745	9.25	2	36.52
	ANDC	0	11.5	0.33	0.14	0.80
	MOOD Suite					
Visibility	MHFP	0	0.81	0.13	0.10	0.13
	AHFP	0	1	0.57	0.66	0.35
Polymorphism	PFP	0	2.4	0.09	0	0.18
Coupling	CFP	0	1	0.04	0.01	0.09
Inheritance	MIFP	0	0.98	0.47	0.52	0.32
	AIFP	0	1	0.40	0.39	0.32

Table 5: Descriptive statistics for the proposed metrics in Eclipse 3.0

3.7.1.2 Descriptive statistics for Eclipse faults

Tables 6, 7 and 8 show descriptive statistics for faults found in Eclipse packages. It can be observed from these tables that post-release faults are much less than pre-release faults in all Eclipse releases. Consequently, PostFD and PPostF are less than PreFD and PPreF. Another observation is that PreFD and PostFD decrease when we go from release i to i+1. This is because LOC (denominator) increases dramatically from release i to i+1.

Dependent Variables	Minimum	Maximum	Mean	Median	StdDev
Pre-Release Faults	0	179	11.84	3	23.35
Post-Release Faults	0	88	2.60	1	7.61
PreFD	0	0.33	012	0.01	0.02
PostFD	0	0.13	0.002	0.0004	0.01
PPreF	0	1	75%	1	0.43
PPostF	0	1	50.52%	1	0.50

Table 6: Descriptive statistics for Fault in Eclipse 2.0

Dependent Variables	Minimum	Maximum	Mean	Median	StdDev
Pre-Release Faults	0	151	6.97	2	13.96
Post-Release Faults	0	71	1.57	0	4.48
PreFD	0	1	0.011	0.003	0.06
PostFD	0	0.125	0.002	0	0.008
PPreF	0	1	66.97%	1	0.47
PPostF	0	1	45.43%	0	0.5

Table 7: Descriptive statistics for Fault in Eclipse 2.1

Dependent Variables	Minimum	Maximum	Mean	Median	StdDev
Pre-Release Faults	0	220	7.12	2	15.90
Post-Release Faults	0	65	2.35	0	5.60
PreFD	0	1	0.01	0.003	0.04
PostFD	0	0.06	0.002	0	0.004
PPreF	0	1	63.45%	1	0.48
PPostF	0	1	47.74%	0	0.50

Table 8: Descriptive statistics for Fault in Eclipse 3.0

3.7.2 Univariate Analysis

To test the hypotheses defined in Section 3.2 statistically, we used Spearman's correlation analysis with a level of significance α = 0.05, which investigates the relationship between a dependent variable and the independent variables. Spearman's correlation was chosen because it does not require any underlying distribution in the data. By using the proposed metrics, the Spearman's correlation coefficient and the p-value were calculated. The correlation coefficient ranges from -1 for perfect negative correlation, through 0 when there is no correlation, to 1 for perfect positive correlation. The p-value determines whether the correlation is significant (if p-value < 0.05) or not. Tables 9, 10, 11 and 12 show the results of the Spearman's correlation with the p-value (shown between brackets) between dependant variables and independent variables for the 3 consecutive releases of Eclipse. The bold values indicate significant correlation. The results discussions of the correlations were organized based on the categories of the studied metrics (i.e. Martin, C&K and MOOD metrics).

3.7.2.1 PreFD and Package-Level Metrics

The correlation analysis shows that Martin metrics have positive significant correlation with PreFD except for Instability "**I**" and Distance from the main sequence (D), as shown in Table 9. Ca and Ce have positive significant correlation with PreFD in all Eclipse releases while "A" has negative significant correlation in release 2.1 and positive correlation in 2.0 and 3.0. The negative coefficient of "A" indicates that a package is more likely to have more faults when the ratio between abstract classes and total number of classes (i.e. NOA / No. Classes) in a package decreases whereas positive coefficient indicates that fault decreases when this ratio increases. This is because the numerator (NOA) has negative significant correlation in 2.1 and positive significant correlation in the others. An explanation for this is that, increasing abstract classes in a package reduces possibility of faults as abstract classes do not have implementation for methods defined in these classes. However, abstract classes need to be extended by other classes and consequently methods need to be implemented (either by classes in the same package or in different packages) which, in turn, increases possibility of faults. This might explains why NOA and "A" have negative and positive coefficients in Eclipse releases. On the other hand, No. Classes has positive significant correlation in two releases (2.0 and 3.0) and positive correlation (with p-value 0.13) in release 2.1, which is close to be significant. This is might be because there were some faults that could not be discovered through the testing phase (i.e. pre-release faults), however, they were found after releasing the system. In fact, No. classes were significant with PostFD in all releases, as this can be seen in PostFD section.

For C&K suite, all metrics have positive significant correlation with PreFD except for TDIT, ADIT and ANDC. In general, inheritance metrics did not have significant correlation with PreFD, except for TNDC in release 3.0. This might be because their mean values and standard deviations were relatively low. ADIT has negative correlation in all releases while TDIT, TNDC and ANDC have negative correlation in release 2.1 and positive for the others. The negative coefficient of DIT indicates that fault-proneness decreases with the depth of the class, which means that classes located deeper in the inheritance hierarchy are less fault-prone while the positive correlation indicates that classes located deeper in the inheritance hierarchy are more likely to have more faults.

The relationships of CBO metrics to PreFD are particularly strong. In fact, ACBO and TCBO have the highest correlation among all the package-level metrics in release 2.0, although that release 2.1 has the maximum statistical values (e.g. mean and variance), as shown in section 3.7.1.1. Moreover, ACBO has higher correlation than TRFC, which has the highs statistical values in Eclipses releases. This indicates that ACBO is very good indicator for faults in package. This might be due to the fact that CBO measures are the only measures that count both inward and outward coupling which increases the complexity of a package and so increases the possible faults in that package. On the other hand, MOOD metrics were not as good as C&K and Martin metrics. AHFP, MHFP and CFP have positive significant correlation only in one release (release 2.0 for AHFP, MHFP and 2.1 for CFP). In addition, MHFP and CFP have also positive correlation close to be significant (with p-value 0.0502 for MHFP and 0.063 for

CFP) in release 2.1 and positive correlation (not significant) in release 3.0. This can be explained by the fact that PreFD decreases from release i to i+1 (it was 0.333 in release 2.0, 0.1202 in 2.1 and 0.00944 in 3.0).

Since Ca, Ce, TLCOM, TRFC, TWMP, TCBO and ACBO metrics have significant correlation through all releases, the H1-PreFD hypothesis will be accepted with respect to these metrics and the H0-PreFD will be rejected.

In summary, the above results indicate that faults are more likely to increase when coupling between packages (e.g. Ca and Ce) increases. In addition, increasing coupling between object (CBO) and number of methods (as in TWMP and AWMP) in a package will increase the possibility of faults in this package. However, increasing cohesion in a package (i.e. decreasing TLCOM) will decrease the faults in that package. The result also shows that CBO metrics are better than the others. In fact, ACBO is the best metrics that can be used as indicator for PreFD. This is because the larger ACBO in a package, the more complexity there is to manage and so the more likely to have more faults in this package. TCBO comes second and Ce comes third. It can be also shown from Table 9 that C&K-Total metrics have higher correlation than C&K-Average. In addition, it can be observed that C&K suite is the best indicator for PreFD, Martin metrics come second and finally MOOD metrics come last.

Pair of Variables	Spearman Rank Order Correlations (**Eclipse** releases) Marked correlations are significant at p<.05000		
	2.0	2.1	3.0
Martin Metrics			
No. Classes	**0.114759 (0.026067)**	0.072051 (0.137164)	**0.150756 (0.000124)**
A	0.017345 (0.737442)	**-0.156452 (0.001188)**	0.007075 (0.856278)
I	0.016323 (0.752391)	0.090736 (0.061022)	-0.007131 (0.856781)
D	-0.049042 (0.737442)	0.056976 (0.240051)	0.040105 (0.309925)
Ca	**0.186075 (0.000286)**	**0.098509 (0.00)**	**0.166412 (0.000022)**
Ce	**0.251647 (0.000001)**	**0.264205 (0.00)**	**0.203499 (0.00)**
NOA	0.04791 (0.354155)	**-0.09591 (0.04761)**	**0.080308 (0.041775)**
C & K Suite			
TLCOM	**0.103963 (0.00001)**	**0.112725 (0.019809)**	**0.226685 (0.00)**
ALCOM	0.034956 (0.499182)	0.087355 (0.071348)	**0.201688 (0.00)**
TRFC	**0.171620 (0.000833)**	**0.171022 (0.000385)**	**0.150856 (0.000123)**
ARFC	**0.142826 (0.005529)**	**0.177732 (0.000223)**	0.070149 (0.075479)
TWMP	**0.147824** (0.004070)	**0.122087 (0.011575)**	**0.154015 (0.000088)**
AWMP	0.090175 (0.080759)	**0.121614 (0.011903)**	0.066803 (0.090544)
TDIT	0.003873 (0.940327)	-0.047089 (0.331678)	0.013970 (0.723660)
ADIT	-0.025421 (0.623162)	-0.076792 (0.113078)	-0.021685 (0.583088)
TNDC	0.024114 (0.641143)	-0.007688 (0.490616)	**0.085445 (0.030280)**
ANDC	-0.006622 (0.898165)	-0.033448 (0.490616)	0.063226 (0.109217)
TCBO	**0.256165 (0.00)**	**0.190235 (0.000076)**	**0.218063 (0.00)**
ACBO	**0.306274 (0.00)**	**0.257854 (0.00)**	**0.192641 (0.000001)**
MOOD Suite			
MHFP	**0.115380 (0.025265)**	0.094792 (0.050294)	0.038553 (0.329035)
MIFP	0.080125 (0.120900)	0.013340 (0.783430)	0.031895 (0.419429)
PFP	0.035858 (0.488177)	0.015751 (0.745522)	-0.012779 (0.746381)
CFP	**0.129160 (0.012187)**	0.089908 (0.063430)	0.009914 (0.801877)
AIFP	0.067760 (0.189841)	0.027344 (0.573106)	0.027287 (0.489743)
AHFP	0.190049 (0.752391)	**0.143569 (0.002945)**	0.045561 (0.248637)

Table 9: PreFD and package-level metrics

3.7.2.2 PostFD and Package-Level Metrics

Table 10 shows that all Martin metrics have positive significant correlation with PostFD except for "**I**". Number of classes (No. Classes), Ca and Ce have positive significant correlation with PostFD through all Eclipse releases whereas NOA has significant correlation only in one release (release 3.0). This might be due to the fact that packages in release 3.0 have more NOA than in 2.0 and 2.1 (as shown in Tables 3, 5 and 6). On the other hand, "A" has negative significant correlation in 2.0 and 2.1 but not significant in 3.0, which has the highest statistical values of "A" compared to 2.0 and 2.1 packages.

Table 10 shows also that TCBO, ACBO, TRFC, ARFC, TWMP, AWMP and TLCOM have positive significant correlation with PostFD in all Eclipse releases. As in PreFD, TCBO has the strongest correlation in the studied metrics. In addition, it is it is interesting to note that TCBO was highest (0.34) in release 2.0, although release 2.1 has the maximum statistical values, as shown in Table 4. Moreover, TCBO has higher correlation than TRFC, which has the highs statistical values in Eclipses releases. In fact, Empirical Study 3[4] shows that TRFC is more correlated to size than TCBO. This indicates that TCBO is very good indicator for faults in package. TWMP, which is a size metrics, has the second highest correlation in Table 10 while TRFC was the third. Although inheritance was not used that much in Eclipse releases, TDIT, ADIT, TNDC and ANDC have positive significant correlation with PostFD in release 2.0 and 3.0. However, ANDC and ADIT were not having significant correlation with PostFD. This

[4] Empirical Study 3 investigates correlation between the proposed metrics and effort in term of LOC

might be because packages in release 2.1 have, on average, more classes, which is the denominator for ANDC and ADIT, than the other releases.

As in PreFD, MOOD metrics were not as good as C&K and Martin metrics. AIFP, MHFP and CFP have positive significant correlation only in one release (release 2.0 for CFP and 3.0 for AIFP and MHFP) while PFP has positive significant correlation in 2.0 and 3.0 but not in 2.1. This is because the denominator (counting a maximum number of possible method overrides) for PFP grows proportionally faster than the numerator (actual number of method overrides).

Since No. Classes, Ca, Ce TLCOM, TRFC, ARFC, TWMP, AWMP, TCBO and ACBO metrics have significant correlation through all releases, H1-PostFD hypothesis will be accepted with respect to these metrics and H0-PostFD will be rejected.

In summary, as in PreFD, the results show that faults increase when coupling between packages increases. Also, increasing number of methods in a package will increase the possibility of faults in this package. However, increasing cohesion and abstractness in packages will decrease the faults. The results also show that CBO metrics are better than the others. Actually, TCBO was the best metrics that can be used as indicator for faults. TRFC was the second best and TWMP is the third. It can be also shown from Table 10 that C&K-Total metrics have higher correlation than C&K-Average. In addition, it can be observed that C&K suite is the best indicator for PostFD prediction, Martin metrics come second and finally MOOD metrics come last.

Pair of Variables	Spearman Rank Order Correlations (**Eclipse** releases) Marked correlations are significant at p<.05000		
	2.0	2.1	3.0
Martin Metrics			
No. Classes	**0.272478 (0.00)**	**0.194208 (0.000053)**	**0.261093 (0.00)**
A	**-0.155322 (0.002527)**	**-0.102007 (0.035145)**	-0.031102 (0.431259)
I	0.075723 (0.142773)	0.007659 (0.874606)	0.020804 (0.598493)
D	0.046665 (0.366868)	**0.142194 (0.003233)**	**0.102490 (0.009304)**
Ca	**0.135689 (0.008425)**	**0.155909 (0.001229)**	**0.212355 (0.00)**
Ce	**0.242739 (0.000002)**	**0.176263 (0.000252)**	**0.292377 (0.00)**
NOA	-0.00821 (0.873787)	0.0202 (0.676849)	**0.1222783 (0.001894)**
C & K Suite			
TLCOM	**0.236299 (0.000004)**	**0.211346 (0.000011)**	**0.251191 (0.00)**
ALCOM	**0.094456 (0.067315)**	**0.152813 (0.001540)**	**0.129151 (0.001030)**
TRFC	**0.337038 (0.00)**	**0.229355 (0.000002)**	**0.316878 (0.00)**
ARFC	**0.241714 (0.00)**	**0.143144 (0.003031)**	**0.234399 (0.00)**
TWMP	**0.316744 (0.000002)**	**0.218217 (0.000005)**	**0.296625 (0.00)**
AWMP	**0.199194 (0.000101)**	**0.130742 (0.006823)**	**0.201842 (0.00)**
TDIT	**0.156168 (0.002391)**	0.071119 (0.142335)	**0.149695 (0.000139)**
ADIT	**0.130693 (0.011191)**	0.032742 (0.499820)	**0.113200 (0.004052)**
TNDC	**0.161774 (0.001648)**	0.087055 (0.072330)	**0.204968 (0.00)**
ANDC	**0.106300 (0.039378)**	0.031281 (0.519153)	**0.169194 (0.000016)**
TCBO	**0.341808 (0.00)**	**0.255153 (0.0000)**	**0.323976 (0.00)**
ACBO	**0.267513 (0.00)**	**0.222677 (0.000003)**	**0.262954 (0.00)**
MOOD Suite			
MHFP	0.073681 (0.153895)	0.071282 (0.141420)	**0.129105 (0.001034)**
MIFP	0.072378 (0.161325)	-0.017112 (0.724394)	0.067319 (0.088075)
PFP	**0.116562 (0.023796)**	0.006266 (0.897276)	**0.140807 (0.000342)**
CFP	**0.128445 (0.012678)**	0.084003 (0.082953)	0.030365 (0.442097)
AIFP	0.100788 (0.050842)	0.026986 (0.578141)	**0.084591 (0.031979)**
AHFP	0.033832 (0.513090)	0.061673 (0.203412)	0.017183 (0.663632)

Table 10: PostFD and package-level metrics

3.7.2.3 PPreF and Package-Level Metrics:

Table 11 shows that all Martin metrics have positive significant correlation with PPreF except "**I**". No. of classes, Ca and Ce have positive significant correlation in all Eclipse releases while NOA (as in PostFD) has positive significant correlation in two releases 2.0 and 3.0. Table 11 also shows that all C&K metrics have positive significant correlation with PPreF in all Eclipse releases except TDIT, ADIT and ANDC in release 2.0 (as in PostFD). However, TDIT was very close to be significant (p-value 0.09). This is because, unlike PreFD and PostFD, PPreF has larger mean (0.699) than PreFD and PostFD, as shown in Table 7. For the same reason, MOOD metrics achieved better correlation than PreFD and PostFD. In fact, MHFP and PFP have positive significant correlation with PPreF in all releases. Moreover, MIFP, CFP and AIFP have positive significant correlation in tow releases of Eclipse.

Since No. Classes, Ca, Ce TLCOM, TRFC, ARFC, TWMP, AWMP, TCBO, ACBO, MHFP and PFP metrics have significant correlation through all releases, the H1-PPreF hypothesis will be accepted with respect to these metrics and the H0-PPreF will be rejected.

To summarize this, the above results show that faulty packages increase when coupling between packages increases. Also, increasing number of methods, private methods (i.e. MHFP), overriding methods (i.e. PFP) and classes in a package will increase the possibility of finding fault in this package and hence the number of faulty packages in a system. However, increasing cohesion and abstractness in packages will decrease possibility of finding faults in a package. Among all the metrics in Tables 11, TRFC, TWMP and TCBO have the strongest correlation (between 0.42 and 0.39). In addition, as in PostFD, C&K-Total metrics have higher correlation than C&K-Average. Furthermore, it can be observed from the above that C&K suite is the best indicator for PPreF, Martin metrics come second and finally MOOD metrics come last.

3.7.2.4 PPostF and Package-Level Metrics:

Table 12 shows the correlation result between PPostF and the proposed metrics. The result was very close to the one in Table 12. As in PPreF, No. of classes, Ca and Ce have positive significant correlation in all Eclipse releases with PPostF while NOA has positive significant correlation in two releases 2.0 and 3.0. In addition, all C&K metrics have positive significant correlation with PPostF in all Eclipse releases except ANDC in release 2.1, which has p-value very close to 0.05. MHFP and PFP have positive significant correlation with PPostF in all Eclipse releases. Furthermore, MIFP have positive significant correlation in tow releases of Eclipse and CFP and AIFP have positive significant correlation in one release and very close to be significant in the second release.

Since No. Classes, Ca, Ce TLCOM, TRFC, ARFC, TWMP, AWMP, TCBO, ACBO, MHFP and PFP metrics have significant correlation through all releases, the H1-PPostF hypothesis will be accepted with respect to these metrics and the H0-PPreF will be rejected.

Pair of Variables	Spearman Rank Order Correlations (**Eclipse** releases) Marked correlations are significant at p<.05000		
	2.0	2.1	3.0
Martin Metrics			
No. Classes	**0.343632 (0.00)**	**0.266185 (0.0)**	**0.333896 (0.00)**
A	0.016846 (0.744740)	**-0.154870 (0.001332)**	-0.036467 (0.356814)
I	-0.016143 (0.755042)	0.091381 (0.059199)	0.036170 (0.359828)
D	0.063606 (0.218511)	**0.140640 (0.003590)**	0.063663 (0.106781)
Ca	**0.234750 (0.00004)**	**0.169050 (0.000451)**	**0.199103 (0.00)**
Ce	**0.252351 (0.00001)**	**0.318676 (0.00)**	**0.318726 (0.00)**
NOA	**0.16237 (0.001582)**	-0.0019 (0.96834)	**0.15065627 (0.000126)**
C & K Suite			
TLCOM	**0.315914 (0.00000)**	**0.263833 (0.00)**	**0.352712 (0.00)**
ALCOM	**0.120579 (0.019342)**	**0.121894 (0.011708)**	**0.226378 (0.00)**
TRFC	**0.412864 (0.00)**	**0.393529 (0.00)**	**0.362755 (0.000)**
ARFC	**0.250246 (0.000001)**	**0.302832 (0.00)**	**0.219328 (0.00)**
TWMP	**0.426556 (0.00)**	**0.355638 (0.00)**	**0.361620 (0.00)**
AWMP	**0.273134 (0.00)**	**0.275768 (0.00)**	**0.211423 (0.00)**
TDIT	**0.182082 (0.000387)**	**0.092580 (0.055931)**	**0.160942 (0.000041)**
ADIT	**0.126213 (0.014325)**	0.032489 (0.503135)	**0.099353 (0.011713)**
TNDC	**0.225289 (0.00)**	**0.125595 (0.009378)**	**0.216120 (0.00)**
ANDC	**0.159166 (0.000257)**	0.052061 (0.283112)	**0.152336 (0.000105)**
TCBO	**0.390443 (0.00)**	**0.332088 (0.00)**	**0.381643 (0.00)**
ACBO	**0.274783 (0.00)**	**0.270274 (0.00)**	**0.263770 (0.00)**
MOOD Suite			
MHFP	**0.126311 (0.021310)**	**0.166955 (0.024853)**	**0.130447 (0.000915)**
MIFP	**0.103561 (0.005615)**	**0.055962 (0.248537)**	**0.114996 (0.003500)**
PFP	**0.177531 (0.000006)**	**0.155552 (0.001262)**	**0.126809 (0.001272)**
CFP	**0.171790 (0.011653)**	**0.136935 (0.004587)**	0.062103 (0.115668)
AIFP	**0.110975 (0.001721)**	**0.086627 (0.073748)**	**0.100842 (0.010508)**
AHFP	0.146627 (0.220699)	**0.126728 (0.008752)**	0.062082 (0.115787)

Table 11: PPreF and packages-level metrics

In summary, as in PPreF, the results show that faulty packages increase when coupling between packages increases. Also, increasing number of methods, private methods, overriding methods and classes in a package will increase the possibility of finding faults in this package and hence the number of faulty packages in a system. However, increasing cohesion and abstractness in packages will decrease possibility of finding faults in a package. Among all the metrics in Tables 12, TRFC, TWMP and TCBO have the strongest correlation (0.48, 0.46 and 0.44 respectively). In addition, as in PostFD, C&K-Total metrics have higher correlation than C&K-Average. It can be observed also from the above that C&K suite is the best indicator for predicting PPreF, Martin metrics come second and finally MOOD metrics come last.

3.7.2.5 Summary

The overall results showed that package-level metrics can be used as good indicator for fault prediction in term of PreFD, PostFD, PPreF and PPostF. Table 13 gives the best suites metrics that achieved significant correlation through all Eclipse releases. Among all Martin's metrics, No. Classes, Ca and Ce were the best. TLCOM, TRFC, ARFC, TWMP, AWMP, TCBO and ACBO were the best fault prediction indicators in C&K suite. PFP and MHFP were the best predictors for fault in MOOD suite.

An interesting observation is that coupling metrics were good indicators for faults in a package while inheritance metrics (e.g. TDIT, ADIT, TNDC and ANDC) were not always good indicators for PreFD and PostFD. However, they were good in predicting PPreF and PPostF. This is because PPreF and PPostF have higher mean and

Pair of Variables	Spearman Rank Order Correlations (**Eclipse** releases) Marked correlations are significant at p<.05000		
	2.0	2.1	3.0
Martin Metrics			
No. Classes	**0.404232 (0.00)**	**0.343007 (0.00)**	**0.388326 (0.00)**
A	-0.137226 (0.744740)	**-0.085528 (0.077560)**	-0.070385 (0.074536)
I	0.061301 (0.755042)	0.010973 (0.821133)	0.040367 (0.306771)
D	0.075594 (0.218511)	**0.160484 (0.000874)**	**0.119162 (0.002473)**
Ca	**0.202954 (0.00004)**	**0.214302 (0.00001)**	**0.242059 (0.00)**
Ce	**0.325386 (0.00001)**	**0.263478 (0.00)**	**0.381972 (0.00)**
NOA	**0.07863 (0.00)**	0.0117 (0.15566)	**0.15065627 (0.000047)**
C & K Suite			
TLCOM	**0.338205 (0.00)**	**0.325572 (0.00)**	**0.351577 (0.00)**
ALCOM	**0.114130 (0.00)**	**0.160773 (0.000856)**	**0.150157 (0.000132)**
TRFC	**0.487245 (0.00**	**0.401729 (0.00)**	**0.471944 (0.00)**
ARFC	**0.308976 (0.00)**	**0.242628 (0.00)**	**0.340752 (0.00)**
TWMP	**0.466300 (0.00)**	**0.386030 (0.00)**	**0.441722 (0.00)**
AWMP	**0.263057 (0.00)**	**0.219240 (0.000005)**	**0.295901 (0.00)**
TDIT	**0.256839 (0.00)**	**0.180083 (0.000183)**	**0.231721 (0.00)**
ADIT	**0.200034 (0.000094)**	**0.104051 (0.031584)**	**0.164170 (0.000029)**
TNDC	**0.271922 (0.00)**	**0.186192 (0.000109)**	**0.277250 (0.00)**
ANDC	**0.187444 (0.000257)**	0.082286 (0.089461)	**0.199737 (0.00)**
TCBO	**0.448345 (0.00)**	**0.388267 (0.00)**	**0.433005 (0.00)**
ACBO	**0.290382 (0.00)**	**0.273024 (0.00)**	**0.293932 (0.00)**
MOOD Suite			
MHFP	**0.118715 (0.021310)**	**0.125186 (0.009613)**	**0.204323 (0.00)**
MIFP	**0.142567 (0.00)**	0.037359 (0.441300)	**0.116301 (0.00)**
PFP	**0.230775 (0.00)**	**0.114159 (0.018286)**	**0.231204 (0.003143)**
CFP	**0.129967 (0.001721)**	0.080493 (0.096687)	0.059806 (0.129790)
AIFP	0.161137 **(0.220699)**	0.083852 (0.083507)	**0.132878 (0.000730)**
AHFP	0.063305 (0.235690)	0.061483 (0.204814)	0.030722 (0.436753)

Table 12: PPostF and package-level metrics

than PreFD and PostFD, as shown in Table 7. Therefore, they will be considered as good indicator for faults in package. Another interesting observation is that C&K suite is the best indicator for fault, Martin metrics come second and finally MOOD metrics come last. Among all the studied metrics, TRFC and TCBO are the best metrics that can be used as indicators for fault, as they have the strongest correlation across Eclipse releases.

Suites	Metrics	Category
Martin	Ca, Ce	Coupling
	No. Classes	Size
C&K	TLCOM	Cohesion
	TRFC, ARFC, TCBO, ACBO	Coupling
	TWMP, AWMP	Size
MOOD	MHFP	Visibility
	PFP	Polymorphism

Table 13: Best indicators for fault in a package

From the above analysis, the following points can be used to characterize faulty packages and improve the quality of packages in OOD:

- Size of package: the larger the package, the more likely to have more faults. Whether measured in number of methods or classes, the larger a package the more code it contains and the more things that could impose a fault.

- Cohesion: decreasing cohesion in a package will increase the possible faults in that package. Low cohesion in a package makes its elements (e.g. methods and variables) more difficult to understand and so harder to maintain. Consequently, this will increase the likelihood of errors during the development of this package.

- Number of Afferent Coupling: more packages that depend upon classes inside this package implies more possible faults. If there are a greater number of other packages that depended upon this package, this indicates that this package has more functions in the system and a lot of relationships with other packages which increases the likelihood of errors in this package.

- Number of Efferent Coupling: more classes inside this package that depend upon classes outside this package implies more possible faults in this package as this increases complexity between packages, which increases the likelihood of faults during development.

- Internal coupling (CBO): increasing coupling between classes inside a package increases the complexity of this package and hence increases the possible faults in that package.

- Abstractness: Package that has low abstractness is likely to have more fault than package with high Abstractness. This is because abstract classes do not contain implementation code for functions defined in this package.

- Polymorphism Factor: increasing total degree of methods overriding in the class inheritance tree in a package obviously increase it complexity. This in turn increases likelihood of faults in this package.

- Method Hiding Factor: increasing hidden (private) methods increases possibility of faults in this package. This is because private methods do not allow other classes to use them outside the class that defined them. Consequently, new methods need to be created, which increase the possibility of faults.
- Inheritance: the more of depth of inheritance tree and number of descendents classes in package, the more complicate the package will be and hence the more likely to have more faults.

3.7.3 Multivariate Analysis

Multivariate regression [49] is used to determine whether a particular combination of the package-level metrics provides the best estimate for the external quality attributes. Thus, the studied metrics were used to build the best models for each independent variable. These metrics were categorized into 6 groups:

- All-Suites: it includes the best subset of C&K, MOOD and Martin metrics.
- Martin: It contains the best subset of Martin metrics.
- MOOD: It contains the best subset of MOOD metrics.
- C&K-All: It contains the best subset of C&K metrics (Total and Average).
- C&K-Total: It contains the best subset of C&K-Total metrics.
- C&K-Average: It contains the best subset of C&K-Average metrics.

Stepwise regression was done using Minitab tool (using default setting: forward and backward with alpha to enter as 0.15 and alpha to remove as 0.15) [51], which is well known statistical tool, to determine the best subset of each group for PreFD, PostFD, PPreF and PPostF in Eclipse 2.0, 2.1 and 3.0 datasets. The best subset for each group is shown in Appendix A. Ca, Ce, TLCOM, TWMP, TCBO, TRFC, ARFC, CFP and MHFP were the most common metrics that appear in these subsets.

After that, the best subset of each group was used to build regression or classification model for faults prediction in Eclipse packages. Weka [52], which is an open source machine-learning tool, was used to build and validate the regression models. To estimate the accuracy of a classification or regression model, the dataset was divided into training and testing sets. More specific, Eclipse 2.0 dataset was used to train the model in Weka and Eclipse 2.1 dataset was used to test it. Then, Eclipse 2.0 and 2.1 datasets were combined together, to increase the size of training dataset, and used as training dataset. After that, Eclipse 3.0 dataset was used to test that model. These models are presented in Appendix B.

3.7.4 Evaluation Measures

This section presents evaluation measures used to evaluate the performance of prediction models. Evaluation measures can be divided into two groups: classification and regression. These measures have been selected based on the most commonly and widely used in the literature.

3.7.4.1 Regression Evaluation Measures

There are several measures used to evaluate regression prediction models. However, most of the commonly used regression measures are derived from Magnitude of Relative Error (MRE). MRE is the standard evaluation criterion to assess software prediction models. It is defined as follows [53]:

$$\text{MRE}_i = \frac{|\,\text{Actual}_\text{i} - \text{Predicted}_\text{i}\,|}{\text{Actual}_\text{i}}$$

The regression measures that were derived from MRE are MMRE and Pred (25).

MMRE

Mean Magnitude of Relative Error (MMRE) [54], which is well-known evaluation criteria, was used to evaluate the performance of regression prediction models. MMRE is the mean of MRE. In other words, it is the aggregation of MRE $_i$ values over multiple observations (N):

$$\text{MMRE} = \frac{1}{n}\sum_{i}^{N}\text{MRE}_\text{i}$$

Pred(25)

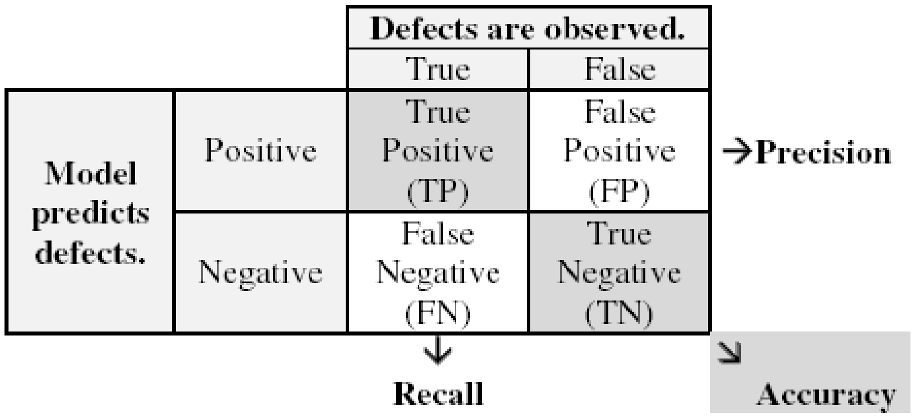
Another measure, which is derived from MRE, called Pred(25) was used to evaluate the regression model. Pre(25) is the percentage of observations whose MRE is less than or equal to level 0.25. It is calculated as follow [54]:

$$\text{Pred(25)} = \frac{C}{n}$$

Where C is the number of observations (cases) whose MRE is less than or equal to the 0.25, and n is the total number of observations in the dataset. Higher score of Pred(25%) implies better predictive accuracy. By contrast, lower score of MMRE indicate better predictive accuracy.

3.7.4.2 Classification Evaluation Measures

The evaluation measures of a prediction model for correctly or not correctly classified packages (as faulty or not-faulty) can be obtained from a confusion matrix, similar to the one shown in Table 14. The confusion matrix has four categories [15, 54]: True Positives (TP) are packages correctly classified as positives. False Negatives (FN) are positive packages incorrectly classified as negative. False Positives (FP) are negative packages incorrectly classified as positive. Finally, True Negatives (TN) refers to negative packages correctly classified as negative.



		Defects are observed.		
		True	False	
Model predicts defects.	Positive	True Positive (TP)	False Positive (FP)	**→Precision**
	Negative	False Negative (FN)	True Negative (TN)	
		↓ **Recall**		↘ **Accuracy**

Table 14: Confusion matrix for binary classification problems [27]

To assess the quality of a classification model, the following measures have been used:

Recall

Recall is the percentage of true positives that are classified correctly (predicted and observed as faulty packages). It is calculated as follows [55]:

$$\text{Recall} = \frac{TP}{TP + FN}$$

A value close to one is superlative and would mean that every package that had faults observed was predicted to have faults.

Precision

Precision is the ratio between the number of correctly identified positives (predicted and observed as faulty packages) and number of packages predicted as faulty. It is calculated as follows [54]:

$$\text{Precision} = \frac{TP}{TP + FP}$$

A value close to one is desirable and would mean that every package that was predicted to have faults actually had faults.

Accuracy

The classification accuracy, or correct classification rate, can be defined as number of correct classifications (i.e. both true positives and true negatives) to the total number of packages (i.e. classifications) [15, 55].

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

A value of one is best and would mean that the model classified perfectly (i.e. all cases have been correctly classified).

The evaluation measures for each model created in the previous sections are as follow:

3.7.5 PreFD, PostFD and package-level metrics

Table 15 presents the results obtained by building regression models for PreFD and PostFD based on the best subsets of the proposed metrics, with bold values as indication of best achieved result. The best Pred(25%) for PreFD model was achieved by using Martin metrics (28.36 %) in dataset 3.0 and the best MMRE was produced by C&K-Total (2.906). In addition, C&K-Total achieved the best Pred(25%) and MMRE (28.57% and 1.019 respectively) for PostFD in dataset 3.0. However, Martin was second best with a small MMRE difference of (0.009).

The performance of these models was not perfect. This is because of the low variance of PreFD and PostFD. In addition, this performance can be improved by using some of AI techniques such as Artificial Neural network and hybrid techniques [57]. However, since we are not interested here in getting the best accurate results, by applying different AI techniques, this will be considered as one direction for future work.

Models			2.1 (Train 2.0 Test2.1)		3.0 (Train 2.0-2.1 Test3.0)	
			PreFD	PostFD	PreFD	PostFD
All-Suites		PRED (25%)	14.33%	13.45%	12.56%	25.78%
		MMRE	3.451	1.775	2.916	1.276
Martin		PRED (25%)	10.13%	13.45%	**28.36%**	20.90%
		MMRE	3.644	1.761	5.996	1.028
MOOD		PRED (25%)	10.48%	15.78%	14.53%	24.04%
		MMRE	3.682	1.721	2.997	1.1233
C&K	All	PRED (25%)	11.18%	12.28%	13.05%	25.05%
		MMRE	3.464	2.00	3.042	1.376
	Total	PRED (25%)	11.18%	13.45%	12.31%	**28.57%**
		MMRE	3.464	1.763	**2.906**	**1.019**
	Average	PRED (25%)	11.18%	11.69%	12.56%	24.73%
		MMRE	3.464	1.903	2.934	1.2355

Table 15: Classification accuracy for PreFD, PostFD and package-level metrics

Figure 11 shows the MRE box plots of Eclipse 2.1 results obtained from PreFD regression models. Boxes sizes seem to be close to each other, and they are almost at the same level. However, C&K models were the best model as they had the narrowest boxes and smallest whiskers. On the other hand, Martin and MOOD models were the worst as they have the highest boxes and longest whiskers compared with others.

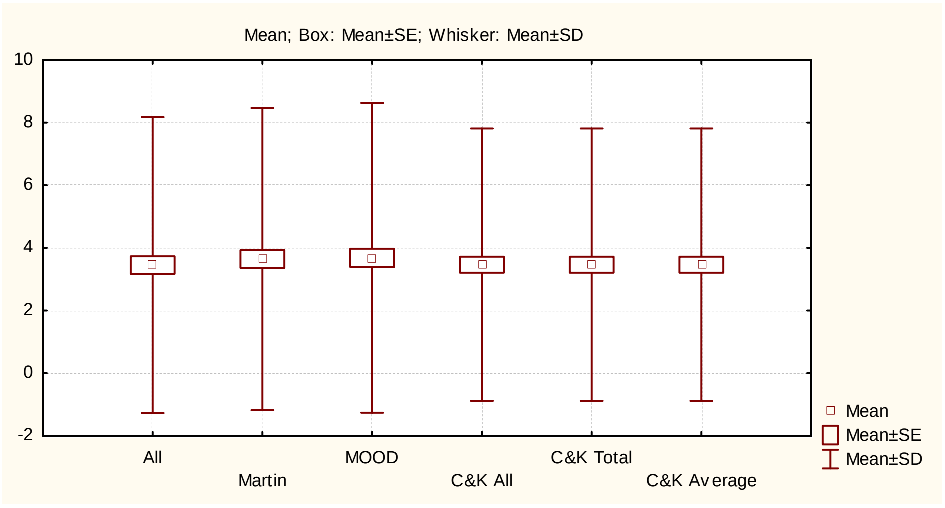


Figure 11: Box Plot for PreFD in Eclipse 2.1

Figure 12 box plots Eclipse 2.1 results obtained from PostFD regression models. C&K-Total and Martin metrics were the best, since they had the narrowest boxes and smallest whiskers. On the other hand, C&K-All and MOOD models were the worst as they have the highest boxes and longest whiskers compared with others.

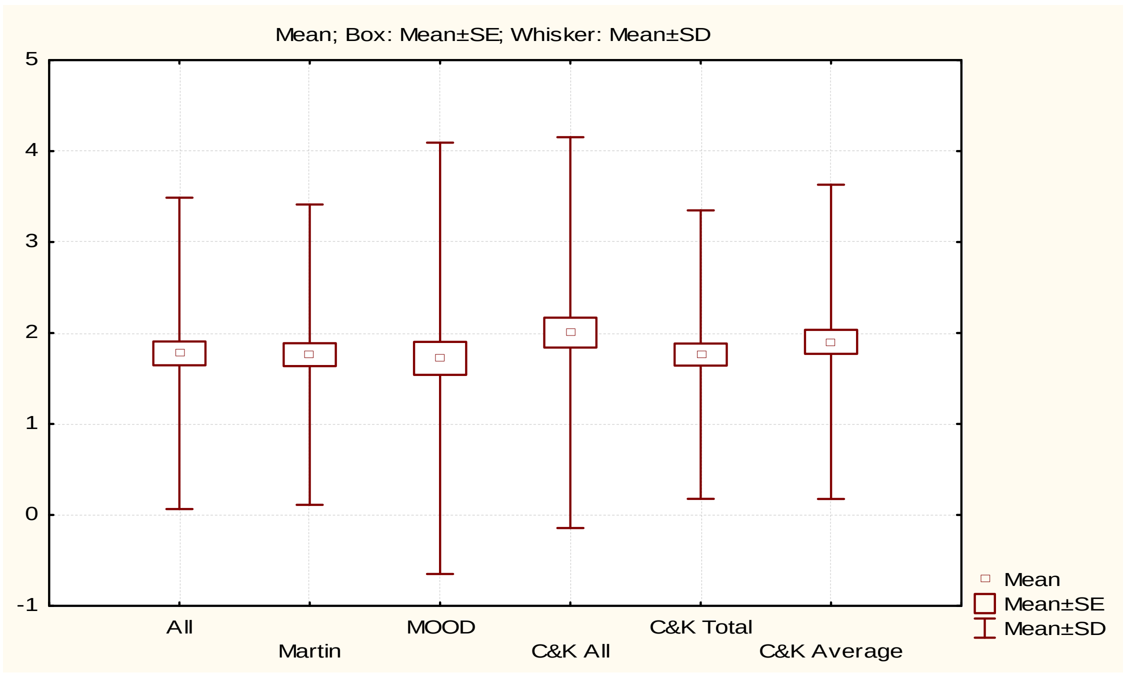


Figure 12: Figure Box plot for PostFD in Eclipse 2.1

Figure 13 box plots Eclipse 3.0 results obtained from PreFD regression models. Boxes sizes seem to be close to each other, and they are almost at the same level. However, C&K-Total was the best since it had the lowest box and smallest whisker.

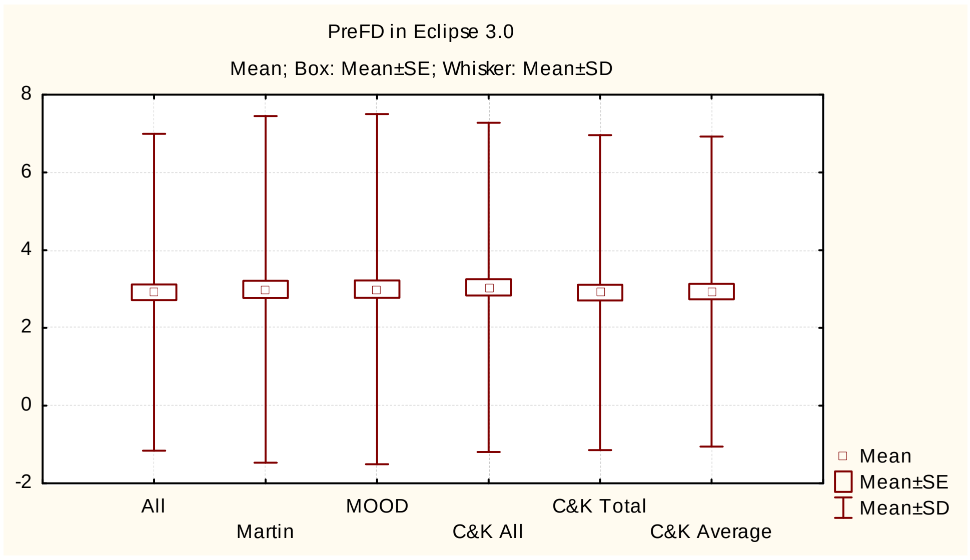


Figure 13: Box Plot for PreFD in Eclipse 3.0

Figure 14 box plots Eclipse 3.0 results obtained from PostFD regression models. C&K-Total has the narrowest box and the smallest whisker. However, Martin model was competitive with them in the lower box and whisker.

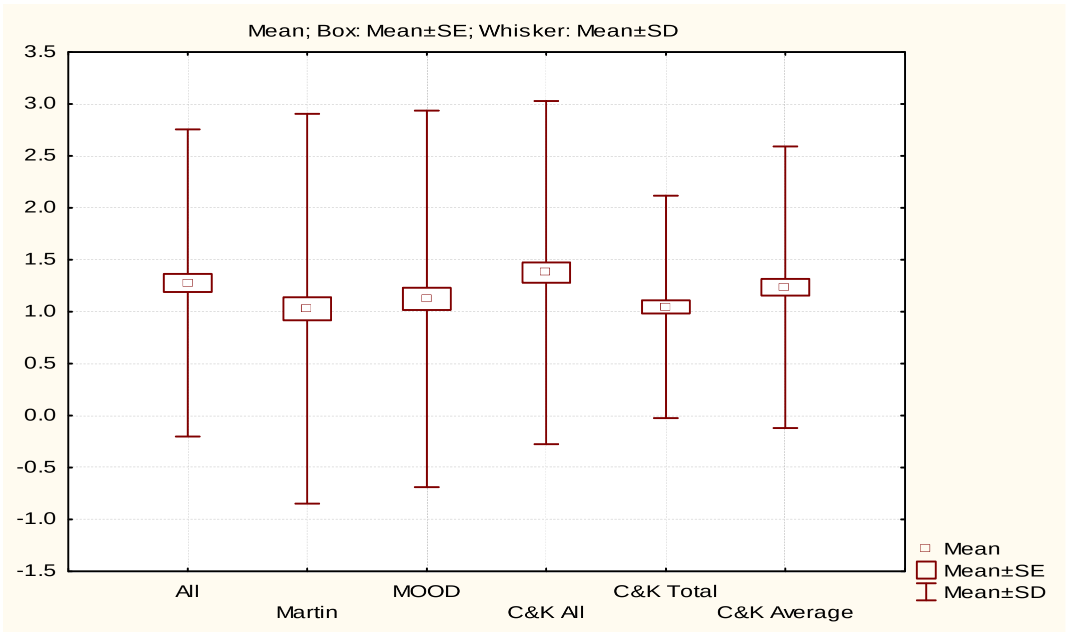


Figure 14: Box Plot for PostFD in Eclipse 3.0

In summary, the above results show that C&K models achieved the best performance in predicting PreFD and PostFD, which supports our finding. In addition, Martin models showed close and competitive results in PreFD and PostFD. However, MOOD was the worst model, which also supports our findings discussed in the univariate analysis. C&K-Total was chosen to be best regression model, since it achieved the best MMRE and Pred(0.25), as shown in Table 14 .

3.7.6 PPreF and package-level metrics

Table 16 presents the accuracy of the classification results obtained from applying classification regression on PPreF based on the best subsets of the proposed metrics, with bold values indicating the best achieved result. Overall results show a competition between the models, which is built from these subsets. The accuracy of these models was between 63.5% and 74.47% depending on the metrics used to build the model. The best accuracy was accomplished by using All-Suites metrics with 74.47 % in release 2.1. However, C&K-All produced the best precision and accuracy in release 3.0 (69.05%), as shown in table 16. C&K-Total scored the best precision and recall in 2.1 (with a minor difference from All Suite) and the best recall in 3.0.

Models		Measures	2.1 (Train 2.0 Test 2.1)	3.0 (Train 2.0 & 2.1 Test 3.0)
All-Suites		Accuracy	**74.47%**	69.05%
		Precision	0.755	0.691
		Recall	0.916	0.926
Martin		Accuracy	66.97%	66.71%
		Precision	0.67	0.668
		Recall	1	0.946
MOOD		Accuracy	67.2%	64.38%
		Precision	0.671	0.641
		Recall	0.997	0.998
C&K	**All**	Accuracy	70.72 %	**69.67%**
		Precision	0.707	**0.702**
		Recall	0.962	0.907
	Total	Accuracy	66.97%	63.45%
		Precision	**0.76**	0.635
		Recall	**1**	**1**
	Average	Accuracy	68.61%	66.56%
		Precision	0.685	0.673
		Recall	0.983	0.922

Table 16: Classification accuracy for Package-level Metrics

3.7.7 PPostF and package-level metrics

Table 17 presents the accuracy of the classification results obtained from applying classification regression on PPostF based on the best subsets of the proposed metrics. The accuracy results were between 57% and 68% depending on the metrics used to build the model. The best accuracy for this model was produced by using the C&K-Total and All-Suites models (67.91%) in release 2.1. C&K-Total also produced the best accuracy in release 3.0 (it correctly classifies 68.1 % of the 3.0 packages). Moreover, C&K-Total scored the best recall in 2.1 and 3.0. However, All Suite scored the best precision in 2.1 and 3.0.

Models		Measures	2.1 Train 2.0 Test 2.1	3.0 Train 2.0 & 2.1 Test 3.0
All-Suites		Accuracy	**67.91%**	66.09%
		Precision	**0.72**	**0.65**
		Recall	0.674	0.78
Martin		Accuracy	64.40%	66.25%
		Precision	0.65	0.64
		Recall	0.737	0.80
MOOD		Accuracy	55.26%	57.85%
		Precision	0.6	0.58
		Recall	0.541	0.67
C&K	**All**	Accuracy	67.49%	66.25%
		Precision	0.66	0.64
		Recall	0.77	0.83
	Total	Accuracy	**67.91%**	**67.03%**
		Precision	0.69	0.64
		Recall	**0.75**	**0.85**
	Average	Accuracy	60.88%	62.20%
		Precision	0.65	0.61
		Recall	0.61	0.75

Table 17: Classification accuracy for Package-level Metrics

In summary, overall results showed that these models gave competitive results against PPreF and PPostF. These can be shown from Figure 15 which compares the accuracy of these models against PPreF and PPostF in Eclipse datasets 2.1 and 3.0. All-Suites models produced the best accuracy. C&K-All comes second with a minor difference between it and both C&K-Total and Martin. An explanation for this is that, All-Suites group contains the best metrics in both Martin (e.g. No. Class, A, Ca and Ce) and C&K-All (e.g. TLCOM, TCBO, TWMP, TRFC, ARFC and ACBO) subsets. In addition, the majority of C&K-All subset contains the best metrics in C&K-Total (e.g. TCBO, TRFC, TLCOM and TWMP), as shown in Appendix B.

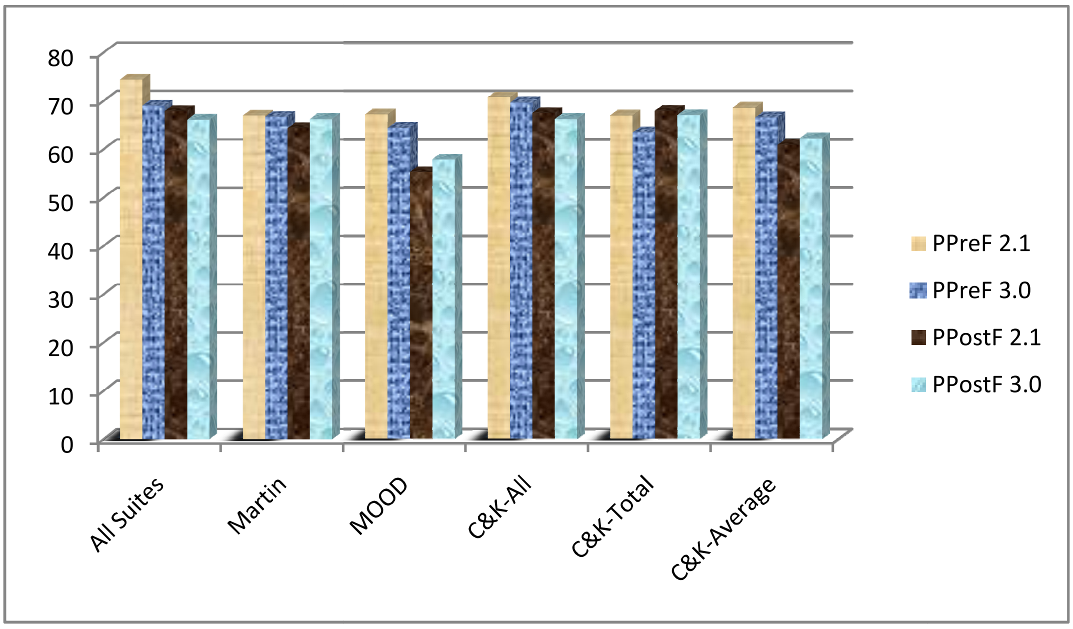


Figure 15: Accuracy of the classification results

3.8 Confounding Effect of Package Size

It has been proposed in many studies that size should be taken into account as a confounding variable when validating object-oriented metrics [58, 59]. The goal of such studies was to explore, after controlling the size confounder, whether all associations between investigated metrics and the dependent variables, such as fault and change-prediction, exist or not.

In this section, the confounding effect of package size in the result presented in section 3.7 will be examined to demonstrate whether the association between the investigated metrics and fault prediction is real or not. To do that, the prediction model has been created twice for each metric in section 3.7 to predict fault proneness of packages: first based on individual metric *m* and second based on *m* and size. Size is considered as number of classes in a package. Then, the Wilcoxon test has been performed to compare the results obtained from each model. The Wilcoxon test is a nonparametric test that compares two paired groups. It calculates the difference between each set of pairs, ranks them from smallest to largest by absolute value and analyzes them. The P value determines whether populations have different medians or not. If the P value is small (i.e. < 0.05), the idea that the difference is due to chance could be rejected and it can be concluded instead that the populations have different medians. However, If the P value is large (i.e. > 0.05), the data do not give any reason to conclude that the overall medians differ [60], which indicates that there is no confounding effect of package size with the investigated metrics.

Table 18 presents the results of Eclipse. The bolded values indicate those metrics which are significant at a p-value less than 0.05. The result of Wilcoxon test showed that all models built using metrics that found to be good indicators for faults in PreFD were significant (p-value < 0.05) with models built by using each one of these metrics and Number of classes in a package. This indicates that the association between these metrics and fault prediction of packages disappears (i.e. was not real) after controlling for the size confounding. Therefore, it can be concluded that size of a package in term of number of classes could be used as indicator for pre-release faults during the testing phase instead of these metrics.

	PreFD		Post FD	
Metrics	**Z**	**p-value**	**Z**	**p-value**
Ca	**7.155176**	**0.000000**	**3.323387**	**0.000889**
Ce	**4.641500**	**0.000003**	1.536532	0.124409
TLCOM	**6.240398**	**0.000000**	**1.976662**	**0.048081**
TRFC	**3.890027**	**0.000100**	1.972504	0.058553
TCBO	**5.770644**	**0.000000**	1.63527	0.10199
ACBO	**5.024547**	**0.000001**	1.841966	0.065481
TWMP	**4.863031**	**0.000001**	1.678096	0.093329

Table 18: Results of the model after controlling the size for Eclipse

In PostFD, the result of Wilcoxon test showed that models built using Ce, TRFC, TCBO, ACBO and TWMP were insignificant (p-value > 0.05) with models built by using each one of these metrics and number of classes in a package. This indicates that the association between these metrics and fault prediction of packages is real regardless of the package's size. On the other hand, the association disappears when Ca and TLCOM metrics were used to build the models. As a result, no conclusion can be obtained for these two metrics.

From the above results, it can be observed that association between PLM and fault- prediction disappears after controlling for the size confounding in PreFD.

Therefore, the null hypotheses (H0-PreFD) will be accepted and the alternative (H1-PreFD) hypothesis will be rejected. However, the results of PostFD showed that Ce, TRFC, TCBO, ACBO and TWMP have real association with post-release faults for a package after controlling the package size. Therefore, the alternative hypothesis (H1-PostFD) will be accepted and null hypotheses (H0-PostFD) will be rejected with respect to these metrics.

3.9 Conclusion

Overall empirical results, produced in this chapter, showed that some of the proposed metrics are good indicators for fault prediction in term of PreFD, PostFD, PPreF and PPostF. More specific, No. Classes, Ca and Ce were the best in Martin's metrics. In addition, TLCOM, TRFC, ARFC, TWMP, AWMP, TCBO and ACBO were the best fault prediction indicators in C&K suite. PFP and MHFP are the best predictors for fault in MOOD suite.

The results also showed that size of package in term of number of classes may affect the validity of the investigated metrics in predicting pre-release faults. On the other hand, some of these metrics have real association with post-release faults in a package. As a result, it can be concluded that number of classes in a package could be used to predict faults during testing face (i.e. pre-release faults) whereas Ce, TRFC, TCBO, ACBO and TWMP metrics can be used to predict faults after releasing the system. Among the investigated metrics, TCBO were the best metric that can be used as indicators for fault prediction as it has no confounding effect with size and has the strongest correlation with faults.

In general, C&K suite was the best indicator for fault prediction. Martin metrics come second and finally MOOD metrics come last. These results have been supported by our findings from the regression analyses. The accuracy of the predictions models reached to more than 74% in predicting faulty packages. Furthermore, the accuracy can be improved by using some of AI technique such as Artificial Neural network and hybrid techniques.

Chapter 4

Empirical Study 2: Change Density and Package Metrics

4.1 Goal

The goal of this empirical study can be defined as follows: investigate the correlation between change proneness of a package and sets of package-level metrics for the purpose of identifying and characterizing change-prone packages from the point of view of researchers and practitioners in the context of OO software.

4.2 Hypotheses

The following hypotheses will be investigated in this research. The metric m, used in the below hypotheses, ranges over the set of object-oriented metrics studied in this research.

- H0-ChD (Null Hypothesis): There is no correlation between metric *m* and the Change Density (ChD) of a package from releases *i* to release i+1.
- H1-ChD (Alternative Hypothesis): There is correlation between metric *m* and the Change Density (ChD) of a package from releases *i* to release i+1.

4.3 Subjects

As in Empirical Study 1, Eclipse was used as subject for this empirical study. In addition, a second system called GanttProject was used as another subject for this study.

GanttProject is a project scheduling application written in Java and featuring Gantt chart, resource management, calendaring, import/export (MS Project, HTML, PDF, spreadsheets) [61]. GanttProject has three releases. Table 18 contains a summary of GanttProject releases.

Release Date	Release	No. packages	No. Classes	No. Methods	LOC
2007-12-17	2.0.6	55	737	5450	29459
2008-06-10	2.0.7	55	737	5498	29772
2008-11-04	2.0.8	55	737	5518	29901

Table 19: Summary of GanttProject system

4.4 Experimental Variables

The independent variables are the metrics under investigation. The dependent variable in this study is the change density. Change density is measured by dividing the amount of lines of code (LOC) changed in a package from release i to i+1 by the total number of LOC of this package in release i [62]. More specific, the amount of change is defined as: LOC added, deleted and modified from release i to i +1.

Therefore, Change Density (ChD) =

$$\frac{\text{\# of LOC (added, deleted and modified) for a package from release i to i + 1}}{\text{Total LOC in that package in relase i}}$$

4.5 Data Collection

JHawk was used to extract the package-level metrics (independent variables), by considering Eclipse releases (2.0, 2.1 and 3.0) and GanttProject releases (2.0.6, 2.0.7 and 2.0.8). Then, DiffDocpro [63], which is visual file and directory comparison tool, was used to calculate the amount of change in a package between release i and i+1 in term of LOC. During the comparison between each two releases, comments, blank lines and white space in line were excluded from the comparison process (option in DiffDocpro tool). Then, change density was calculated by dividing this number by the total number of LOC of this package in release i. Change density was used as a proxy for measuring the change in a package from release i to release i+1. Next, structural properties of packages in release i, as measured by the studied metrics, were then associated with the change density between release i and i+1 (using univariate regression), to determine if each individual package-level metric is significantly correlated to the change density. Finally, multivariate regression was used to build models for change prediction in software packages.

4.6 Results and Discussion

The following sections present descriptive statistics for the collected metrics, results of univariate, multivariate analysis and the regression models for change prediction.

4.6.1 Descriptive statistics

The descriptive statistics for Eclipse has been discussed previously in section 3.7 (Empirical Study 1). This section presents descriptive statistics for GanttProject releases. In addition, it shows descriptive statistics for change density in Eclipse and GanttProject packages.

4.6.1.1 Descriptive statistics for metrics results

This section shows descriptive statistic for metrics results in GanttProject releases. Tables 19 and 20 present descriptive statistics (minimum, maximum, mean, median, and standard deviation) for the collected metrics in GanttProject releases 2.0.6 and 2.0.7. These metrics can be categorized into coupling, inheritance, cohesion, visibility and polymorphism. The following can be observed from these metrics:

Coupling metrics:

- As in Eclipse system, the largest maximum value is for TRFC compared to other metrics in Table 19 and 20. It also has the largest mean and standard deviation (variance). This may be explained by the fact that TRFC is the only measure to count indirect coupling, whereas all other coupling measures count connections to directly coupled classes only.
- The lowest mean, maximum and standard deviation values are for CFP compared to other coupling metrics. This is because CFP measures actual direct coupling between classes in a package (i.e. denominator) over maximum possible number of

coupling between these classes (i.e. numerator), which is grows proportionally faster than the numerator.

- In general, all coupling metrics were higher in 2.0.7 than in 2.06. This might be due to the fact that the 2.0.7 has more methods than in 2.0.6.

Inheritance metrics:

Distributions of the values of the metrics show that inheritance has been used cautiously within the two releases (i.e. low standard deviation, mean and median values for ADIT and ANOC). However, there is sufficient variance in TDIT and TNDC to proceed with the analysis.

Size metrics:

All size metrics were higher in 2.0.7 than in 2.06. This is due to the fact that the 2.0.7 has more methods and LOC than in 2.0.6.

Stability metrics:

It can be observed that packages in GanttProject have less Abstractness than in Eclipse. This is because Eclipse releases have more abstract classes than GanttProject. The descriptive statistics for the other stability measures do not show any interesting or surprising trends.

Cohesion metrics

ALCOM indicates the average LCOM for a class in GanttProject packages. ALCOM is bigger than 1 in each class (i.e. mean 1.09), which indicates relatively low cohesion. This is because of the presence of access methods which are typically only reference one attributes, and consequently increase the number of pairs of methods in the class that do not use attributes in common. Since Eclipse releases have much more classes and methods than in GanttProject, LCOM metrics were much less in GanttProject than in Eclipse releases.

Visibility

As in Eclipse, hidden attribute was used a lot in GanttProject releases (around 0.71). However, hidden method was used cautiously within the two releases.

Polymorphism

Packages in GanttProject releases have low PFP (mean is around 0.0692). This may be explained by the fact that PFP depend on inheritance, which was low in GanttProject, as discussed above.

Category		Minimum	Maximum	Mean	Median	StdDev
	Martin Metrics					
Size	No. Classes	1	104	13.4	8	18.15
	NOA	0	17	2.86	1	4.37
Stability	A	0	0.67	0.16	0.15	0.18
	I	0	1	0.58	0.67	0.38
	D	0	1	0.317	0.25	0.32
Coupling	Ca	0	30	4.33	1	6.64
	Ce	0	29	4.33	2	5.23
	C & K Suite					
Cohesion	TLCOM	0	159.48	11.16	4.55	23.70
	ALCOM	0	19.93	1.09	0.5	2.97
Coupling	TRFC	0	4114	273.51	117	601.32
	ARFC	0	39.56	15.36	14.5	8.65
	TCBO	0	484	41.87	16	78.23
	ACBO	0	27.67	3.05	2.2	3.84
Size	TWMP	0	1382	99.09	39	208.65
	AWMP	0	13.28	5.47	5	2.74
Inheritance	TDIT	0	5	0.8	0	1.13
	ADIT	0	0.5	0.062	0	0.095
	TNDC	0	27	3.05	1	5.22
	ANDC	0	2.67	0.22	0.05	0.42
	MOOD Suite					
Visibility	MHFP	0	0.35	0.09	0.59	0.99
	AHFP	0	1	0.71	0.8	0.34
Polymorphism	PFP	0	0.67	0.07	0	0.15
Coupling	CFP	0	1	0.05	0.01	0.15
Inheritance	MIFP	0	0.93	0.27	0.18	0.29
	AIFP	0	0.92	0.26	0. 10	0.30

Table 20: Descriptive statistics for the package-level metrics in GanttProject 2.0.6

Category		Minimum	Maximum	Mean	Median	StdDev
	Martin Metrics					
Size	No. Classes	1	104	13.4	8	18.15
	NOA	0	17	2.86	1	4.37
Stability	A	0	0.67	0.17	0.15	0.18
	I	0	1	0.58	0.67	0.38
	D	0	1	0.317	0.25	0.31
Coupling	Ca	0	30	4.33	1	6.64
	Ce	0	29	4.33	2	5.23
	C & K Suite					
Cohesion	TLCOM	0	159.48	11.19	4.55	23.37
	ALCOM	0	19.93	1.09	0.5	2.97
Coupling	TRFC	0	4126	276.02	118	605.88
	ARFC	0	39.67	15.45	14.5	8.65
	TCBO	0	488	42.33	16	78.66
	ACBO	0	28	3.08	2.2	3.87
Size	TWMP	0	1390	99.96	39	210.58
	AWMP	0	13.37	5.51	5	2.76
Inheritance	TDIT	0	6	0.87	0	1.35
	ADIT	0	0.5	0.07	0	0.104
	TNDC	0	27	3.05	1	5.22
	ANDC	0	2.67	0.22	0.05	0.42
	MOOD Suite					
Visibility	MHFP	0	0.35	0.09	0.59	0.99
	AHFP	0	1	0.71	0.81	0.35
Polymorphism	PFP	0	0.67	0.07	0	0.15
Coupling	CFP	0	1	0.05	0.01	0.15
Inheritance	MIFP	0	0.93	0.27	0.18	0.30
	AIFP	0	0.92	0.26	0.01	0.30

Table 21: Descriptive statistics for the package-level metrics in GanttProject 2.0.7

4.6.1.2 Descriptive statistics for change density

Table 21 shows descriptive statistics for change (LOC and change density) occurred in Eclipse and GanttProject packages from release i to i+1. It can be observed from this table that change density in Eclipse datasets were close to each other, although that number of LOC changed in 2.1-3.0 dataset is more than 2.0-2.1. This is because LOC of a package in Eclipse (i.e. denominator) increases dramatically from release i to i+1. Another observation is that change between releases 2.0.6 and 2.07 in GanttProject is more than the one between 2.0.7 and 2.0.8.

System	From - To	Amount of Change	Minimum	Maximum	Mean	Median	StdDev
Eclipse	2.0 - 2.1	LOC	0	51774	1438.09	492.5	3503.14
		ChD	0	26.67	1.18	0.82	2.06
	2.1-3.0	LOC	0	320338	2607.5	641.5	16332.86
		ChD	0	14.62	1.29	1	1.319
GanttProject	2.0.6-2.0.7	LOC	0	1158	49.13	0	187.30
		ChD	0	1.12	0.05	0	0.17
	2.0.7-2.0.8	LOC	0	136	5.29	0	20.36
		ChD	0	0.08	0.005	0	0.016

Table 22: Descriptive statistics for change in Eclipse and GanttProject releases.

4.6.2 Univariate Analysis

As in Empirical Study 1, Spearman’s correlation was performed with a level of significance α = 0.05 (95% level of confidence) between change density and package-level metrics. The results are presented in the following sections.

4.6.2.1 Change density and package-level metrics in Eclipse

Table 22 shows the correlation result between change density and the package-level metrics in Eclipse datasets (2.0-2.1 and 2.1-3.0). The result shows that all metrics have significant correlation with change density except for D, NOA, TDIT, TNDC and ANDC. The correlation was negative in “A” and positive for the others. The negative coefficient of “A” indicates that a package is more likely to have more change when the Abstractness of a package (the ratio between abstract classes and total number of classes in a package) decreases. This is actually consistent with the Stable Dependencies Principle which states that a package should be as abstract as it is stable [12]. On the other hand, Ce has positive significant correlation with change density in both datasets. Furthermore, Ce has the highest correlation (0.379) in Table 22, which indicates that the more classes in a package that depends on other packages, the more likely this package will be exposed to high amount of changes in the next release, as any change of these packages may propagate a change out to the depending package. Insatiability also has positive significant correlation with change density in both datasets whereas No. Classes and Ca have significant correlation in 2.0-2.1.

For C&K suite, all metrics have positive significant correlation with change density except for TDIT, ADIT, TNDC and ANDC. The correlations of these metrics were mixed (i.e. positive and negative) and not significant, except for ADIT in 2.1-3.0 which has negative significant correlation. This is might be because their mean and standard deviations values are relatively low. The negative coefficient of DIT indicates that change density decreases with the depth of the class, which means that packages that have classes located deeper in the inheritance hierarchy are less change-prone. TLCOM has positive significant correlation with change density in both dataset (2.0-2.1 and 2.1-3.0) while ALCOM has positive significant correlation with change density in

dataset (2.0-2.1). On the other hand, CBO metrics have positive significant correlation with change density in both dataset (2.0-2.1 and 2.1-3.0). In fact, ACBO and TCBO have the highest correlation among all the C&K metrics. They also have higher correlation than TRFC, which has the highs statistical values in Eclipses releases. This indicates that they are very good indicators for change density in a package.
For MOOD metrics, MHFP is the only metrics that has positive significant correlation in both datasets. All the other metrics have positive significant correlation with change density in only the first dataset (2.0-2.1). This is might be because of the low variance of these metrics.

In Summary, the above result shows that Ce, ”I”, ”A”, TLCOM, TRFC, ARFC, TWMP, AWMP, TCBO, ACBO and MHFP metrics have significant correlation through all datasets. Therefore, they are considered to be good indicators for change in software packages.

Pair of Variables	Spearman Rank Order Correlations (**Eclipse** releases) Marked correlations are significant at p<.05000	
	2.0 – 2.1	2.1 – 3.0
Martin Metrics		
No. Classes	**0.147337 (0.005031)**	0.085241 (0.104451)
A	**-0.136136 (0.009606)**	**-0.173565 (0.000883)**
I	**0.116729 (0.026572)**	**0.128895 (0.013856)**
D	0.056070 (0.288026)	-0.018180 (0.729582)
Ca	**0.162064 (0.002008)**	0.093020 (0.076319)
Ce	**0.379772 (0.00)**	**0.274763 (0.00)**
NOA	-0.027692 (0.599988)	-0.052633 (0.316631)
C & K Suite		
TLCOM	**0.171778 (0.001050)**	**0.124401 (0.017573)**
ALCOM	0.083426 (0.113567)	**0.134521 (0.010189)**
TRFC	**0.287708 (0.00)**	**0.185027 (0.000387)**
ARFC	**0.269651 (0.00)**	**0.232196 (0.027877)**
TWMP	**0.198172 (0.000151)**	**0.115271 (0.00008)**
AWMP	**0.119611 (0.023033)**	**0.123428 (0.018484)**
TDIT	0.043256 (0.412556)	-0.057724 (0.272012)
ADIT	-0.026868 (0.610888)	**-0.118369 (0.023913)**
TNDC	0.060332 (0.252882)	-0.020521 (0.696376)
ANDC	0.002016 (0.969544)	-0.083505 (0.111728)
TCBO	**0.314935 (0.00)**	**0.182254 (0.000475)**
ACBO	**0.318227 (0.00)**	**0.198364 (0.000139)**
MOOD Suite		
MHFP	**0.348752 (0.00)**	**0.213755 (0.000039)**
MIFP	**0.149813 (0.004335)**	0.008511 (0.871440)
PFP	**0.123973 (0.018453)**	-0.013098 (0.803324)
CFP	**0.165021 (0.001655)**	0.019383 (0.712456)
AIFP	**0.138974 (0.008189)**	0.027725 (0.598025)
AHFP	**0.285855 (0.00)**	0.090600 (0.084322)

Table 23: Correlation between change density in Eclipse

4.6.2.2 Change density and package-level metrics in GanttProject

Table 23 shows the correlation result between change density and the package-level metrics in GanttProject datasets (2.0.6-2.0.7 and 2.0.7-2.0.8). No. classes, Ca, Ce, and NOA have positive correlation with change density in all GanttProject datasets. No. Classes has the highest correlation in Martin metrics (i.e. 0.46). This is because the more classes in a package, the more code it contains and hence the more things that could require a change. Ce was the second highest (with 0.45) which indicates that the more classes in a package that depended on other packages, the more likely change will be in this package, as any change of these packages may propagate a change out to the depending package. NOA has also high correlation (i.e. 0.423) with change density. This indicates that the more abstract classes and interfaces in a package, the more likely to have change. This can be explained by the fact that if a package has more interfaces and abstract classes, then it will provide more services and hence it will have more functions requests, which increases the possibility of package change (e.g. adding new function or modifying existing function) in the next release. On the other hand, Ca has the lowest significant correlation in both datasets (coefficient is 0.34 and 0.27 respectively) in Table 23. The positive significant correlation of Ca can be explained as follow: if there are more packages outside this package that depend upon classes inside this package, then any change in this package will propagate and affect all other packages that depend upon it. However, according to Martin [12], this package (because of the high dependences) should be more stable, which might explain the low value of correlation coefficient compared to the other metrics.

TLCOM, TRFC, ARFC, TWMP, AWMP and TCBO have, as in Eclipse, positive significant correlation in all datasets. Among them, TWMP has the highest correlation values (i.e. 0.56), as the more methods in a package, the more code it contains and the more things that could demand a change. However, inheritance metrics were not good indicators for change density. This is might be because inheritance has been used cautiously in GanttProject.

Unlike Eclipse, MOOD metrics did not show to be good indicators for change. Only PFP has significant correlation with change density and in one dataset. The other metrics have only correlation that is not significant. This is might be because of the low mean and variance for change density in 2.0.7-2.0.8 dataset.

An interesting observation here is that the correlation result in GanttProject for some of C&K and Martin metrics is higher than in Eclipse even though the change density and size of packages that changed from release i to i+1 in Eclipse are larger than in GanttProject (more than 80% of packages in Eclipse has been changed whereas it is less than 20% in GanttProject). This indicates that some of C&K and Martin Metrics were good indicators for change and are not affected by the size of packages in the studied systems (this will be discussed in section 4.7).

In Summary, the above result shows that No. Classes, Ce, NOA, TLCOM, TRFC, ARFC, TWMP, AWMP and TCBO metrics have significant correlation through all datasets. Therefore, they can be considered as good indicators for package change.

Pair of Variables	Spearman Rank Order Correlations (**GanttProject** release) Marked correlations are significant at p<.05000	
	2.0.6 – 2.0.7	2.0.7 – 2.0.8
Martin Metrics		
No. Classes	**0.463109 (0.000370)**	**0.361074 (0.006763)**
A	0.230271 (0.090774)	0.218636 (0.108787)
I	-0.038635 (0.779443)	-0.003702 (0.978603)
D	0.059612 (0.665503)	0.110600 (0.421476)
Ca	**0.341454 (0.010732)**	**0.279790 (0.038561)**
Ce	**0.402989 (0.002285)**	**0.450860 (0.000551)**
NOA	**0.423210 (0.001285)**	**0.326605 (0.014946)**
C & K Suite		
TLCOM	**0.405501 (0.002131)**	**0.321255 (0.016778)**
ALCOM	0.094397 (0.493013)	0.088346 (0.521263)
TRFC	**0.520509 (0.000046)**	**0.434276 (0.000924)**
ARFC	**0.390257 (0.003224)**	**0.416643 (0.001555)**
TWMP	**0.563665 (0.000007)**	**0.415100 (0.001626)**
AWMP	**0.542573 (0.000019)**	**0.384456 (0.003756)**
TDIT	0.253914 (0.061399)	0.179911 (0.188724)
ADIT	0.121855 (0.375482)	0.058523 (0.671263)
TNDC	0.242265 (0.074738)	0.259990 (0.055243)
ANDC	0.095214 (0.489262)	0.134831 (0.326368)
TCBO	**0.391075 (0.003155)**	**0.491056 (0.000141)**
ACBO	068372 (0.619898)	**0.409911 (0.001884)**
MOOD Suite		
MHFP	0.060908 (0.658678)	0.223947 (0.100252)
MIFP	0.073931 (0.591665)	0.019111 (0.889855)
PFP	**0.348525 (0.009116)**	0.092093 (0.503675)
CFP	0.140364 (0.306721)	0.052329 (0.704371)
AIFP	0.102043 (0.458493)	0.002268 (0.986887)
AHFP	-0.144061 (0.294026)	-0.179973 (0.188568)

Table 24: Correlation between change density in GanttProject

4.6.2.3 Summary

From the correlation results, we can conclude that change is more likely to increase when coupling between packages (e.g. Ce and Ca) increases. In addition, increasing coupling between object and number of methods in a package will increase the possibility of change in this package. On the other hand, increasing cohesion in a package (i.e. decreasing TLCOM), will decrease likelihood of change in this package. The results also show that TWMP is the best indicator for change, as it has the highest correlation (i.e. 0.56 in GanttProject). An explanation for this is that, when system moves from release i to i+1, new functions will be added in the next release. This will lead to creating new methods or modifying the existing methods to implement these functions in the next release. In addition, any additional features or change in requirements (e.g. functions or features requested by the users) leads to modifying the existing methods or creating new methods in the next release.

Table 24 presents the best indicators for change density in Eclipse and GanttProject. Bold values indicate the common metrics in Eclipse and GanttProject. It can be observed from this table that No. Classes, Ce, TLCOM, TRFC, ARFC, TWMP, AWMP and TCBO were the best indicators for change as they have high significant correlation with change density in all releases of the studied systems. It can be observed also that MOOD metrics were not good indicators for change as they did not show significant correlation in all systems datasets. Only MHFP and PFP were good indicators in Eclipse but not in GanttProject. This is might be because of the low variance and mean of MOOD metrics and the low value of change density in GanttProject system.

It can be observed from the overall results that C&K-Total metrics have higher correlation than C&K-Average. In addition, it can be observed that C&K suite is the best indicator for change. Martin metrics were the second best and competitive to C&K. However, MOOD metrics were not as good as C&K and Martin.

Suites	**Metrics**		**Category**
	Eclipse	**GanttProject**	
Martin	**Ce**	Ca**, Ce**	Coupling
		No. Classes	Size
	A	NOA	Abstractness
	I		Stability
C&K	**TLCOM**	**TLCOM**	Cohesion
	TRFC, ARFC,TCBO and ACBO	**TRFC, ARFC and TCBO**	Coupling
	TWMP, AWMP	**TWMP, AWMP**	Size
MOOD	MHFP		Visibility

Table 25: Best indicators for change density

From the above analysis, the following points can be used to characterize packages that are more likely to change in the next release of a system:

- Size of package: the larger the package, the more likely to have more change. Whether measured in number of methods or classes, the larger a package the more code it contains and the more things that could impose a change.

- Number of Afferent Coupling: more packages that depend upon classes inside this package implies more possible change. If there are a greater number of other packages that depended upon this package, this indicates that this package has more functions in the system and a lot of relationships with other packages. Any change of this package may propagate a change out to the depending packages which in turn increases the likelihood of change in those packages.

- Number of Efferent Coupling: more classes inside this package that depend upon classes outside this package implies more possible change in this package. This is because there will be more reasons that they may change as requesting changes from the depended upon package increases the likelihood of change of this package.

- Internal coupling (CBO): increasing coupling between classes in a package increases likelihood of change. This is because increasing coupling increases the dependency between classes in package. Any change in the depended upon class propagate a change out to the depending classes which in turn increases the likelihood of change in this package. In addition, increasing coupling between classes increases the likelihood of faults in a package, as seen in Empirical Study 1, and so increases the likelihood of change, to fix these faults.

- Cohesion: decreasing cohesion in a package will increase the likelihood of change for this package. This is because low cohesion in a class indicates that it might be a good idea to split the class into two or more sub-classes, which increases the change in a package.

4.6.3 Multivariate Analysis

As in Empirical Study 1, stepwise regression was used to determine the best subset of All-Suites, Martin, MOOD, C&K-All, C&K-Average and C&K-Total (Shown in Appendix A). Ca, Ce, TLCOM, TWMP, TCBO, TRFC, ARFC, CFP and MHFP were the most common metrics that appear in these subsets. Then, the best subset is used to build prediction model for each group (shown in Appendix B). Eclipse 2.0-2.1 dataset was used to train the regression models and Eclipse 2.1-3.0 dataset was used to test them. Similarly, GanttProject 2.0.6-2.07 was used to train the models and GanttProject 2.0.7-2.0.8 was used to test them. The results are presented in Tables 25 and 26.

Table 25 presents the results obtained by building regression models for change density in Eclipse based on the best subsets of the studied metrics, with bold values as indication of best achieved result. Overall results showed that the performance of these models was not perfect. This is because of the low variance of change density. In addition, the results showed a competition between metrics suites (Pred(0.25) was between 24% and 25%). However, All-Suites achieved the best performance, since it has the best Pred(0.25) and MMRE values, with minor difference between it and Martin.

Models			(Train 2.0-2.1 Test 2.1- 3.0)
All-Suites		PRED (25%)	**25%**
		MMRE	**2.17**
Martin		PRED (25 %)	24.94%
		MMRE	2.25
MOOD		PRED (25 %)	24.33 %
		MMRE	2.28
C&K	All	PRED (25%)	24.85%
		MMRE	2.41
	Total	PRED (25%)	24.85%
		MMRE	2.41
	Average	PRED (25%)	24.64%
		MMRE	2.41

Table 26: Classification accuracy for change density in Eclipse

Figure 16 box plots Eclipse results obtained from change density regression models. Boxes sizes seem to be close to each other, and they are almost at the same level. However, All-Suites model was the best, since it had the lowest box and smallest whisker, with minor difference between it and Martin.

The performance of these models was not perfect. This is because of the low variance of change density. In addition, as mentioned in Empirical Study 1, the performance of these models can be improved by using some AI techniques such as Artificial Neural network and hybrid techniques, which could be one direction for future work.

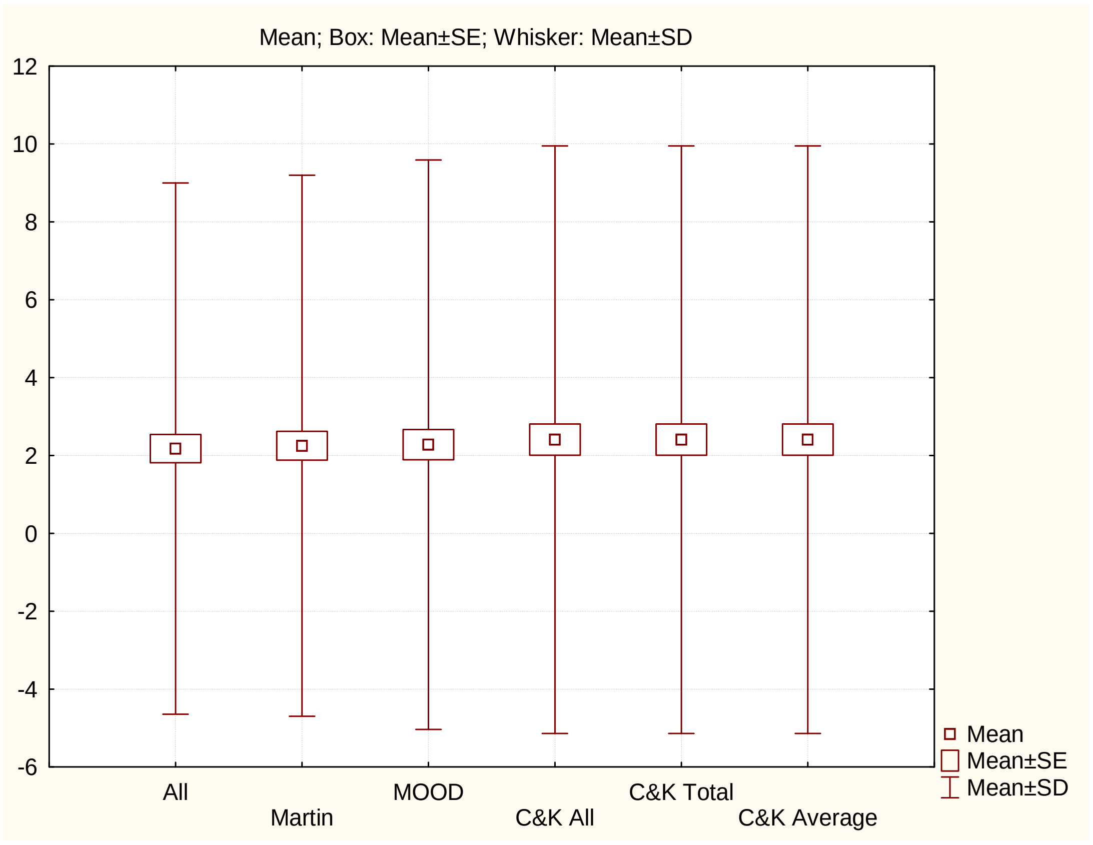


Figure 16: Box Plot for change density models in Eclipse

Table 26 presents the results obtained from evaluating regression models created in multivariate analysis. The performance measures of the created models were not good enough (best value for Pred(0.25) was 11.11%). This is because of the low variance of change density and small size of the dataset (around 10% of the packages have been changed from release 2.0.7 to 2.0.8). All-Suites was superior over other models with Pred(0.25) value of 11.11%. However, Martin, C&K-Total were the best in performance since it has the best MMRE values (i.e. 21.6).

Models			2.08 (Train 2.0.7 Test 2.0.8)
All-Suites		PRED (25%)	11.11%
		MMRE	29.18
Martin		PRED (25 %)	0
		MMRE	**21.60**
MOOD		PRED (25 %)	**0**
		MMRE	30.18
C&K	All	PRED (25%)	0
		MMRE	98.72
	Total	PRED (25%)	0
		MMRE	**21.60**
	Average	PRED (25%)	0
		MMRE	98.72

Table 27: Classification accuracy for change density in GanttProject

Figure 17 box plots GanttProject results obtained from change density regression models. C&K-Total and Martin metrics were the best, since they had the narrowest boxes and smallest whiskers. On the other hand, C&K-All and C&K-Average models were the worst as they have the highest boxes and longest whiskers compared with others.

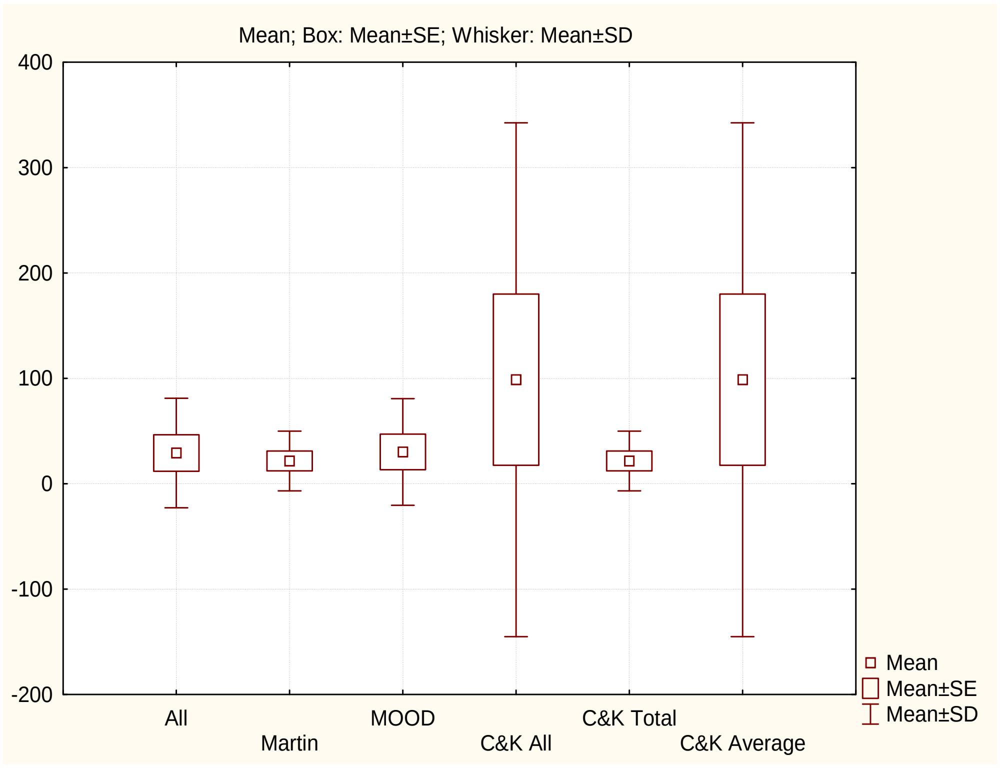


Figure 17: Box Plot for change density models in GanttProject

In summary, results were close and competitive. However, Martin was slightly better than C&K-Total and All-Suites.

4.7 Confounding Effect of Package Size

The objective of this analysis is to explore whether the association between the investigated metrics and change-density of packages (refer to section 3.8 for more detail) was real or not.

Table 28 shows the results for Eclipse and GanttProject respectively after controlling the package size. These results are only for those metrics found to be significant in univariate analysis (section 4.6.2). The bolded values in the tables indicate those metrics whose p-value is less than 0.05.

	Eclipse		GanttProject	
Metrics	**Z**	**p-value**	**Z**	**p-value**
Ce	**2.573731**	**0.010061**	**2.100420**	**0.035693**
TLCOM	**2.573731**	**0.010061**	0.050965	0.959354
TRFC	1.097063	0.272615	1.274118	0.202623
ARFC	0.279611	0.779776	0.050965	0.959354
TCBO	1.841889	0.065492	1.540308	0.123486
TWMP	0.449433	0.653120	1.579906	0.114129
AWMP	1.311646	0.189641	1.172189	0.241122

Table 28: Result of the model after controlling the size for Eclipse and GanttProject

The results show that all the investigated metrics were found to have no confounding effect of package-size except Ce in Eclipse, GanttProject and TLCOM in Eclipse (i.e. p-value < 0.05). The association with the change density of packages disappears for Ce and TLCOM, after controlling the size, while it remains for the other metrics (i.e. Ca, TRFC, ARFC, TCBO, TWMP and AWMP). Since the association of these metrics was real, the alternative hypothesis (H1-ChD) will be accepted with respect to these metrics and null hypothesis will be rejected.

4.8 Conclusion

Overall empirical results, produced in this chapter, showed that package-level metrics can be used as good indicators of change density. More specific, TLCOM, TRFC, ARFC, TWMP, AWMP, TCBO and ACBO are the best indicators for change in C&K suite. MHFP is the best indicator for change in MOOD suite. Ca, Ce, NOA, Abstractness and No. classes are the best indicators in Martin metrics. In addition, it can be observed that all of the correlations were positive except for Abstractness, which was negative. In fact, the negative correlation indicates that the amount of change in a package will decrease if the abstractness of that package increases, which is true according to Stable Abstraction Principle.

The results also showed that Ca, TRFC, ARFC, TWMP, AWMP and TCBO were the best metrics that can be used as indicators for change in this study, as they have significant correlation with change density in all datasets and they have no confounding effect of package-size. C&K metrics were the best indicators for change. Furthermore, C&K-Total metrics were better indicators of change than C&K-Average. Martin metrics come second, and they were competitive to C&K. Finally MOOD metrics come last.

The evaluation results of regression models showed that models built using Martin metrics, C&K-Total and All-Suites outperforms other models. The performance of these models was not perfect. This is because of the low variance of change density and small size of the dataset. In addition, as mentioned in Empirical Study 1, the performance of these models can be improved by using some AI techniques such as Artificial Neural network and hybrid techniques. However, as we are not interested here in getting the best accurate results by applying different AI techniques, this could be considered as one direction for future work.

Chapter 5

Empirical Study 3: Implementation Effort & Package Metrics

5.1 Goal

The goal of this empirical study can be defined as follows: investigate the correlation between implementation effort of a package and the set of package-level metrics for the purpose of estimating implementation effort of software packages from the point of view of researchers and practitioners in the context of OO software.

5.2 Hypotheses

The following hypotheses will be investigated in this research. The metric m, used in the below hypotheses, ranges over the set of package metrics studied in this research.

- H0-Eff (Null Hypothesis): There is no correlation between metric *m* and the implementation effort (in term of LOC) done on a package in a system.
- H1-Eff (Alternative Hypothesis): There is correlation between metric *m* and the implementation effort (in term of LOC) done on a package in a system.

5.3 Subjects

Since we are interested here in estimating the implementation effort, and not the modification effort, the first release of Eclipse and GanttProject (discussed in Empirical

Study 2) was measured to estimate the effort required to implement system packages. Hence, Eclipse 2.0 and GanttProject 2.0.6 will be the subjects of this study.

5.4 Experimental Variables

The independent variables are the package-level metrics under investigation. The dependent variable in this study is the implementation effort. The total LOC for each package of the system will be used as a proxy for measuring implementation effort that has been done on that package.

5.5 Tool

As in the previous studies, JHawk was used to collect the independent and dependant variables of this study.

5.6 Data Collection

JHawk was used to extract the package-level metrics (independent variables) and total LOC of a package (dependent variable), by considering Eclipse releases 2.0 and GanttProject releases 2.0.6. Then, these metrics were associated with the total LOC of a package (using univariate regression), to determine if each individual package-level metric is correlated to the implementation effort. Finally, multivariate regression was used to build prediction models to estimate the implementation effort for each package of the system.

5.7 Results and Discussion

The results of descriptive statistics for Eclipse 2.0 and GanttProject 2.0.6 have been discussed previously in Empirical Study 1 and 2. The following sections present results of univariate, multivariate analysis and the regression models built for predicting the implementation effort of software packages.

5.7.1 Univariate Analysis

Spearman's correlation was performed with a level of significance $\alpha = 0.05$ (95% level of confidence) between implementation effort and the package-level metrics in Eclipse and GanttProject. The results are shown as follow:

5.7.1.1 Implementation effort and package-level metrics in Eclipse

The correlation result between implementation effort and the package-level metrics in Eclipse is shown in Table 27. The bold values indicate significant correlation. The result shows that all metrics have significant correlation with implementation effort except “A” and CFP. The result also shows that all correlations were positive except “A”. The negative coefficient of “A” indicates that a package is more likely to need less effort when the Abstractness of this package increases. An explanation for this is that abstract classes (or interfaces) don't need to have implementation for all of their methods, as they are abstract methods. Thus, they required less effort than concrete classes.

It can be observed from Table 27 that No. classes has the strongest correlation in Martin metrics, as the more classes in a system the more effort will be needed to implement these classes. In addition, it can be observed that TRFC has the strongest correlation (0.96) in C&K metrics. In fact, it has the strongest correlation in Table 27. This may be explained by the fact that TRFC has the largest mean and variance in Eclipse. In addition, TRFC is the only measure to count direct and indirect coupling. TWMP has the second largest correlation value (0.91) in Table 27. This is because TWMP counts the number of methods in a package. Consequently, increasing the methods will increase the implementation effort required to implement these methods.

PFP has the strongest correlation value (0.5134) in MOOD metrics. This is because increasing PFP will increase the degree of methods overriding in the package. This in turn increases the implementation effort required to redefine these methods.

Pair of Variables	Spearman Rank Order Correlations (Eclipse release 2.0) Marked correlations are significant at p<.05000
Martin Metrics	
No.Classes	**0.801613** **(0.00)**
A	-0.071265 (0.167883)
I	**0.130675** **(0.011202)**
D	**0.129384** **(0.012037)**
Ca	**0. 244191** **(0.000002)**
Ce	**0.608720** **(0.00)**
NOA	**0.339046** **(0.00)**
C & K Suite	
TLCOM	**0.669142** **(0.00)**
ALCOM	**0.213866** **(0.000029)**
TRFC	**0.964776** **(0.00)**
ARFC	**0.557107** **(0.00)**
TWMP	**0.917208** **(0.00)**
AWMP	**0.500863** **(0.00)**
TDIT	**0.537819** **(0.00)**
ADIT	**0.376302** **(0.00)**
TNDC	**0.508403** **(0.00)**
ANDC	**0.308339** **(0.00)**
TCBO	**0.760968** **(0.00)**
ACBO	**0.350366** **(0.00)**
MOOD Suite	
MHFP	**0.354035** **(0.00)**
MIFP	**0.313531** **(0.00)**
PFP	**0.513480** **(0.00)**
CFP	0.076777 (0.097285)
AIFP	**0.347604** **(0.00)**
AHFP	**0.112976** **(0.00)**

Table 29: Correlation between implementation effort in Eclipse 2.0

In summary, the above result indicates that the studied metrics are very good indicators for implementation efforts. In addition, it can be observed that C&K-Total metrics have higher correlation than C&K-Average. In addition, it can be observed from the above that C&K suite is the best indicator for implementation effort, Martin metrics come second and finally MOOD metrics come last.

5.7.1.2 Implementation effort and package-level metrics in GanttProject

Table 28 shows the correlation between implementation effort and the package-level metrics in GanttProject. All metrics have positive significant correlation with the implementation effort except I, D, MIFP, CFP, AIFP and AHFP.

As in Eclipse, No. Classes has the strongest correlation in Martin metrics. In addition, TRFC has the highest correlation in C&K metrics and MHFP has the highest correlation in MOOD metrics, with minor difference between it and PFP. Furthermore, as in Eclipse, TRFC has the highest correlation value (0.99) among the studied metrics in GanttProject. TWMP was the second (0.96) and No. Classes was the third (0.90).

In summary, the above result indicates that the studied metrics are very good indicators for implementation efforts. In addition, it can be observed that C&K-Total metrics have higher correlation than C&K-Average. Moreover, it can be observed from the above that C&K suite is the best indicator for implementation effort, Martin metrics come second and finally MOOD metrics come last.

Pair of Variables	Spearman Rank Order Correlations (GanttProject 2.0.6) Marked correlations are significant at p<.05000
Martin Metrics	
No.Classes	**0.906805 (0.00)**
A	**0.386193 (0.003539)**
I	0.004825 (0.972112)
D	0.024976 (0.856366)
Ca	**0.528753 (0.000033)**
Ce	**0.756672 (0.00)**
NOA	**0.678673 (0.00)**
C & K Suite	
TLCOM	**0.783153 (0.00)**
ALCOM	**0.282037 (0.036964)**
TRFC	**0.991052 (0.00)**
ARFC	**0.675434 (0.00)**
TWMP	**0.947663 (0.00)**
AWMP	**0.607690 (0.00)**
TDIT	**0.566061 (0.00)**
ADIT	**0.346570 (0.00)**
TNDC	**0.610171 (0.00)**
ANDC	**0.391424 (0.00)**
TCBO	**0.803520 (0.00)**
ACBO	**0.306522 (0.00)**
MOOD Suite	
MHFP	**0.571959 (0.00)**
MIFP	0.143074 (0.297379)
PFP	**0.499695 (0.000103)**
CFP	0.186760 (0.172163)
AIFP	0.178931 (0.191185)
AHFP	0.152171 (0.267394)

Table 30: Implementation effort and package-level metrics in GanttProject

5.7.1.3 Summary

The above results showed that package-level metrics are good indicators for implementation effort estimation. More specific, the results showed that C&K suite was the best indicator for implementation effort, as it has the highest correlation, Martin metrics come second and finally MOOD metrics come last. In addition, C&K-Total metrics where better than C&K-Average. This is might be because C&K-Total metrics have larger variance and mean than C&K-Average. Among all the studied metrics, TRFC, TWMP and No. classes are the best metrics that can be used to estimate the implementation effort, as they have the strongest correlation across the studied systems (between 0.99 and 0.90).

Table 29 gives the best metrics that achieved significant correlation through Eclipse and GanttProject. Bold values indicate the common metrics in Eclipse and GanttProject. An interesting observation in this table is that PFP and MHFP were the only metrics in MOOD metrics that have significant correlation in both systems. An explanation of this is that increasing PFP will increase the degree of methods overriding in the package which will increase the effort needed to implements these methods. In addition, increasing hidden methods will prevent developers from reusing these methods, which will increase effort needed to implement new methods in a package. Unlike MHFP, AHFP was significant in Eclipse and close to significant in GanttProject. This is might be because defining new hidden variables is much easier than methods and hence it will need less effort than MHFP. Another interesting observation is that Abstractness had negative correlation in Eclipse and significant positive correlation in

GanttProject. This indicates that this metric is not good indicator for estimating implementation effort. This is because increasing abstractness in a package might decrease the implementation effort in this package, as its classes defined as abstract classes, however, this might also increase the effort needed to implement concrete classes that extend the abstract classes in this package.

Suites	Metrics		Category
	Eclipse	GanttProject	
Martin	**Ca and Ce**	**Ca and Ce**	Coupling
	No. Classes	**No. Classes**	Size
	A	NOA	Abstractness
		I	Stability
C&K	**TLCOM, ALCOM**	**TLCOM, ALCOM**	Cohesion
	TRFC,ARFC, TCBO and ACBO	**TRFC,ARFC, TCBO and ACBO**	Coupling
	TWMP, AWMP	**TWMP, AWMP**	Size
	TDIT,ADIT TNDC, ANDC	**TDIT,ADIT TNDC, ANDC**	Inheritance
MOOD	MIFP, AIFP		Inheritance
	MHFP,AHFP	**MHFP**	Visibility
	PFP	**PFP**	Polymorphism

Table 31: Best metrics for implementations effort estimation

From the above analysis, the following points can be used to estimate implementation effort in software packages:

- Size of package: the larger the package, the more likely to require more implementation effort. Whether measured in number of methods or classes, clearly the larger a package the more effort it needs to implement this package.

- Cohesion: decreasing cohesion in a package will increase the likelihood of change for this package (as shown in Empirical Study 2). This is because low cohesion in a class indicates that it might be a good idea to split the class into two or more sub-classes, which increases the implementation effort to split such classes.

- Number of Afferent Coupling: more packages that depend upon classes inside this package implies more possible faults and change (as discussed in Empirical Study 1 and 2), which increase implementation effort needed to fix these faults and maintain changes in this package , as any change of this package may propagate a change out to the depending packages.

- Number of Efferent Coupling: more classes inside this package that depend upon classes outside this package implies more possible faults and changes in this package, as this increases complexity between packages, which in turn increases implementation effort for that package required to fix these faults and maintain change in the package.

- Coupling between Object: increasing coupling between classes in a package increases the complexity of this package and hence increases the possible faults and change in that package (as discussed in Empirical Study 1 and 2). As a result, this will increase implementation effort for this package.

- Polymorphism Factor: increasing degree of methods overriding in the package will increase the implementation effort since these methods need to be redefined. In addition, increasing Polymorphism Factor in a package obviously increase it complexity. This in turn increases likelihood of faults and hence the implementation effort needed to fix these faults in this package.
- Method Hiding Factor: increasing hidden methods increases implementation effort in this package. This is because private methods do not allow other classes to use them outside the class that defined them. Consequently, new methods need to be created, which obviously increases the implementation effort for that package.
- Inheritance: the more of depth of inheritance tree and number of descendents classes in package, the more effort is needed to implements these classes.

5.7.2 Multivariate Analysis

As in Empirical Study 1 and 2, stepwise regression was used to determine the best subset from the following groups: All-Suites, Martin, MOOD, C&K-All, C&K-Average and C&K-Total metrics. The results are shown in Appendix B. No. Classes, TNDC, TWMP, TCBO, ACBO, TRFC, ARFC, CFP and AHFP were the most common metrics that appear in these subsets. Then, the best subset of each group is used to build prediction model (using regression) for each group as in Empirical Study 1 and 2. However, since only one dataset (i.e. dataset for the first release) was used in this study, dataset was split into two parts: 66% as training dataset and 34% for testing (default setting in Weka).

Table 30 presents the results obtained by building regression models for implementation effort based on the best subsets of the studied metrics, with bold values as indication of best achieved result. It can be observed from this table that the best Pred (0.25) was achieved by using the C&K-All subsets (49.2%), with minor difference (0.73) between it and All-Suites. However, the best MMRE value was for C&K-Total (i.e. 1.19).

Models			Eclipse 2.0
All-Suites		PRED (25%)	**49.20%**
		MMRE	1.67
Martin		PRED (25 %)	18.61%
		MMRE	2.16
MOOD		PRED (25 %)	17.81%
		MMRE	8.73
C&K	All	PRED (25%)	48.47%
		MMRE	1.66
	Total	PRED (25%)	45.6%
		MMRE	**1.19**
	Average	PRED (25%)	17.55%
		MMRE	6.85

Table 32: Classification accuracy for implementation effort in Eclipse

Figure 18 box plots Eclipse accuracy results obtained from implementation effort regression models. C&K-Total model was the best, since it has the narrowest box and smallest whisker, with minor difference between it and Martin. On the other hand, MOOD model was the worst as it has the highest box and longest whisker compared with others.

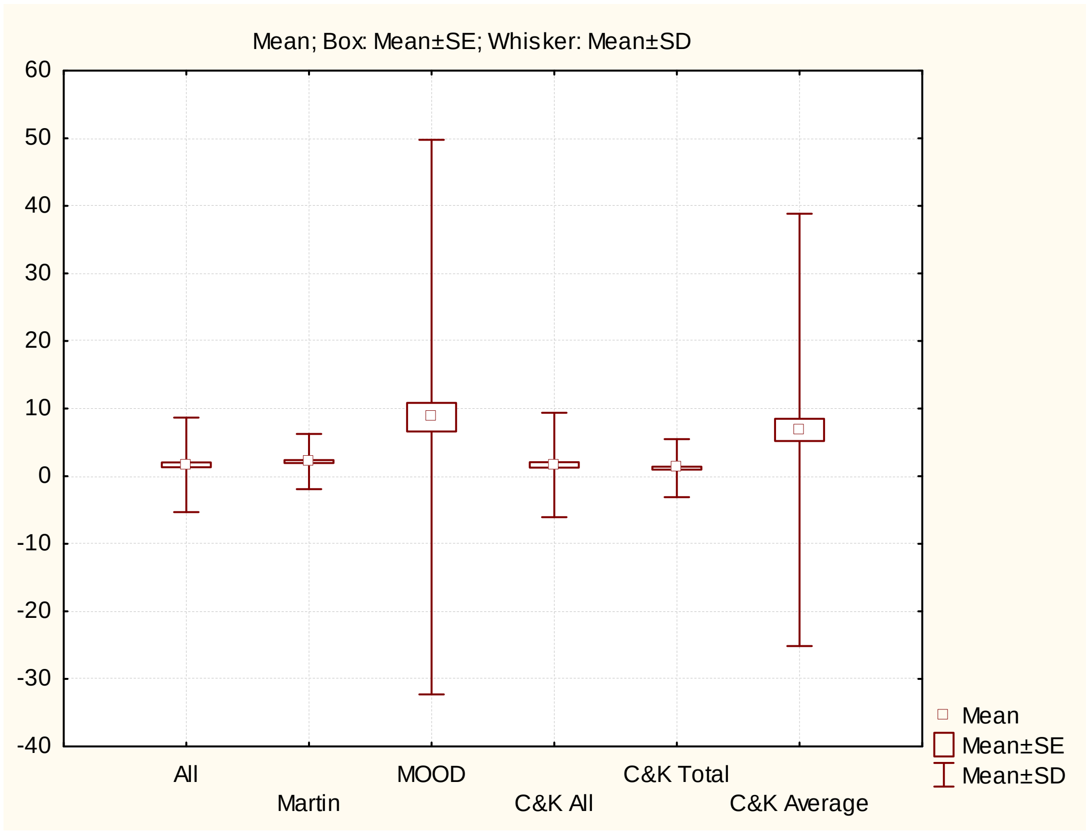


Figure 18: Box Plot for implementation effort models in Eclipse

Table 31 shows the results obtained by building regression models for implementation effort based on the best subsets in each group of the studied metrics in GanttProject. C&K-All model achieved the best Pred(25%) value (i.e. 73.21%), which means that 73% of cases estimates are within the inside 25% of its actual value. However, the best MMRE value was for C&K-Total (0.47), which has also the second best Pred(25) value (i.e. 69%). This can be explained by the fact that CK-All subset consists of all of C&K-Total metrics (i.e. TRFC, TCBO and TNDC) and ANDC, as shown in Appendix A.

Models			GanttProject 2.0.6
All-Suites		PRED (25%)	63.63%
		MMRE	0.81
Martin		PRED (25 %)	18.18%
		MMRE	8.13
MOOD		PRED (25 %)	9.09%
		MMRE	17.38
C&K	All	PRED (25%)	**73.21**%
		MMRE	2.99
	Total	PRED (25%)	69.09%
		MMRE	**0.48**
	Average	PRED (25%)	16.36%
		MMRE	27.52

Table 33: Classification accuracy for implementation effort in GanttProject

Figure 19 box plots Eclipse results obtained from implementation effort regression models. C&K-Total model was the best, since it has the narrowest box and smallest whisker, with minor difference between it and All-Suite. On the other hand, C&K-Average model was the worst as it has the highest box and longest whisker compared with others.

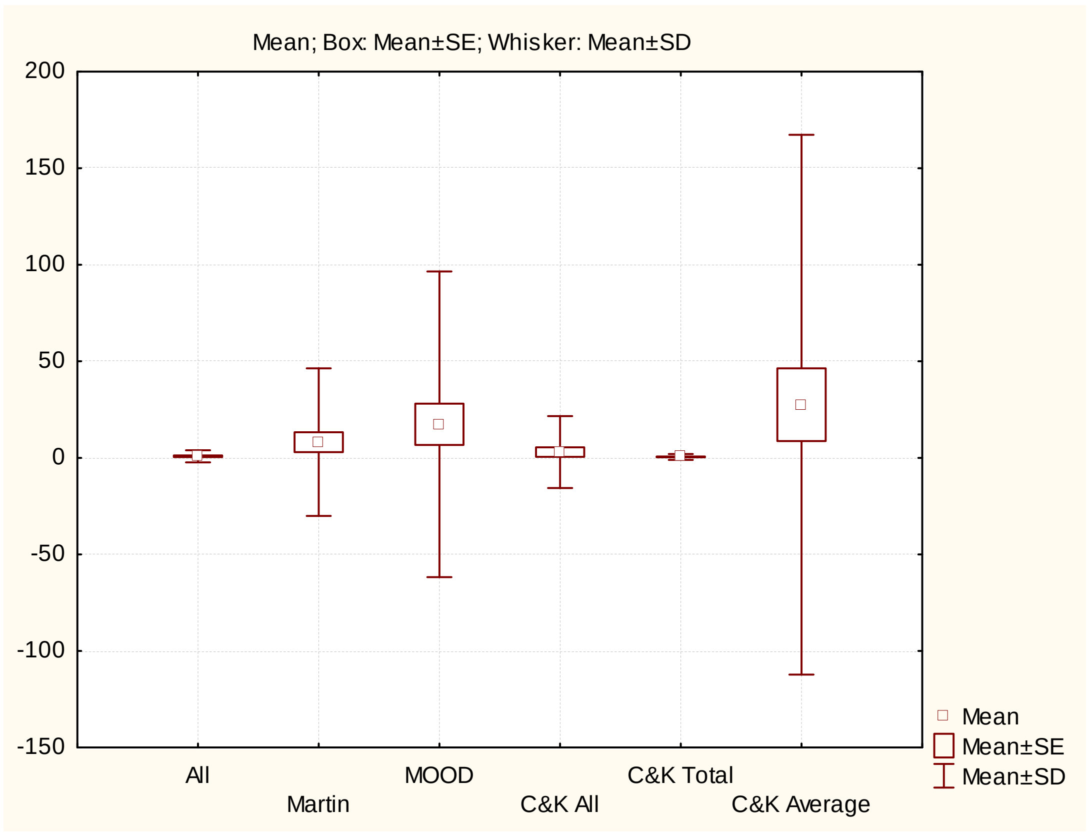


Figure 19: Box Plot for implementation effort models in GanttProject

In summary, the above results showed that C&K-Total models achieved the best performance in estimating effort in Eclipse and GanttProject, which supports our finding in the univariate regression. In addition, Martin models showed close and competitive results. However, C&K-Average and MOOD were the worst models. C&K-Total was chosen to be best regression model, since it achieved the best MMRE value in the studied systems, as shown in Table 30 and 31.

5.8 Confounding Effect of Package Size

As in sections 3.8 and 4.7, this section will explore whether the association between the investigated metrics and implementation effort of packages was real or not. Table 34 shows the results for Eclipse and GanttProject respectively after controlling the package size. These results are for those metrics found to be significant in univariate analysis (section 5.7.1). The bolded values in the tables indicate those metrics whose p-value is less than 0.05.

Eclipse's result shows that all metrics which showed a significant association with the implementation effort of packages in univariate analysis were found to have no confounding effect of package-size. However, some of these metrics were found to have confounding effect of package-size in GanttProject.

Since the association between some of the investigated metrics (Ca, Ce, TLCOM, TRFC, ARFC, TCBO, ACBO and AWMP) and implementation effort of packages is real, the alternative hypothesis (H1-Eff) will be accepted with respect to these metrics and the null hypothesis (H0-Eff) will be rejected.

	Eclipse		GanttProject	
Metrics	**Z**	**p-value**	**Z**	**p-value**
Ca	0.125678	0.899987	1.382460	0.166831
Ce	1.550031	0.121135	0.108921	0.913265
TLCOM	0.083785	0.933227	0.281695	0.778177
ALCOM	0.527849	0.597605	**2.404643**	**0.016189**
TRFC	0.041893	0.966584	0.204943	0.837617
ARFC	0.511091	0.609287	0.965812	0.334139
TCBO	1.927066	0.053972	0.930019	0.352362
ACBO	0.527849	0.597605	0.321937	0.747500
TWMP	0.502713	0.615166	**3.619533**	**0.000295**
AWMP	0.469199	0.638928	0.695420	0.486793
TDIT	1.876795	0.060547	**2.345994**	**0.018977**
ADIT	1.323811	0.185567	**2.404643**	**0.016189**
TNDC	1.524896	0.127286	**2.790057**	**0.005270**
ANDC	0.527849	0.597605	**2.404643**	**0.016189**
MHF	0.636770	0.524275	**2.429779**	**0.015109**
PF	0.779205	0.435860	**2.723028**	**0.006469**

Table 34: Result of the model after controlling the size for Eclipse and GanttProject

5.9 Conclusion

Overall empirical results, produced in this chapter, showed that package-level metrics can be used as a good indicator to estimate implementation effort in software packages. More specific, Ca, Ce, TLCOM, TRFC, ARFC, TCBO, ACBO and AWMP are the best indicators for implementation effort, as they showed significant correlation in both Eclipse and GanttProject systems and have no confounding effect of package-size. TRFC and TWMP are the best metrics that can be used to estimate the implementation effort, as they have the strongest correlation across the studied systems (between 0.99 and 0.90). In addition, the results showed that C&K-Total suite was the best indicator for implementation effort. Martin metrics come second. C&K-Average come third and MOOD metrics come last. These results have been supported by the evaluation results of the regression models, created by using the above metrics. The evaluation results of regression models showed that models built using C&K-Total metrics outperforms other models, as they achieved the best MMRE in the studied systems. Martin and All-Suite models showed close and competitive results to C&K-Total. On the other hand, MOOD and C&K-Averages were the worst and not as good as the others.

CHAPTER 6

Conclusions and Future Work

The goal of this study was to empirically investigate whether a set of package-level metrics are good indicators for implementation and quality assessment attributes such as change prediction, fault prediction and implementation effort estimation. The package-level metrics used in this study were Martin related metrics, C&K and MOOD suites. C&K and MOOD metrics were redefined here (by aggregating them) to be at the package-level. After that, JHawk tool was extended to automate the extraction of these metrics from two java open source systems. Then, each one of the collected metrics was correlated with change density of a package between release i and i+1, faults (i.e. fault density and whether a package is faulty or not) and implementation effort. Finally, regression models were created for each one of them.

Overall empirical results produced in this thesis showed that package-level metrics are good indicators for change prediction, fault prediction and implementation effort estimation. It showed also that C&K metrics were the best. More specific, C&K-Total metrics were the best metrics. Martin metrics come second and MOOD metrics come last.

Based on the above studies, the following guidelines can be considered to enhance the quality of the design in OOD:

- Size of package: the larger the package, the more likely to have more faults (more specific pre-release fault), change and the more likely to require more implementation effort. Whether measured in number of methods or classes, the larger a package the more code it contains, the more effort it needs and more things that could impose a fault and change.
- Cohesion: decreasing cohesion in a package will increase the possible faults, change and implementation effort in that package. Low cohesion in a package makes its elements (e.g. methods and variables) more difficult to understand and so harder to maintain. Consequently, this will increase the likelihood of faults, change and effort during the development of this package. A package that has classes with High LCOM could probably be subdivided into two or more subclasses with increased cohesion.
- Number of Efferent Coupling: when the number of classes inside a package that depend upon classes outside this package increases (outgoing coupling), the possible number of faults, changes and implementation effort will increase. Hence, designers should put this into consideration when designing a package.
- Number of Afferent Coupling: more packages that depend upon classes inside this package implies more possible faults and change, which increases implementation effort to fix these faults and update change in this package , as any change of this package may propagate a change out to the depending packages. Therefore, designers should give extra care about such packages.
- The TCBO/ACBO metrics give a good insight in where the in- and outgoing coupling resides in a package. This metric doesn’t show that there is a design

flaw but it demonstrates where it might be difficult to alter the code due to dependencies. Results show this metrics is the best indicator for faults. Too high TCBO increases the possible number of faults, changes and implementation effort required in a package and could be an indication of misplaced methods. TCBO can be minimized by moving the misplaced method to the class it mostly relates to.

- Designers should try to minimize TRFC /ARFC, which is the total number of methods that are executed in response to a message being received by an object of that class. Results show that TRFC is the best metrics that can be used as an indicator for change prediction and implementation effort estimation. When TRFC/ARFC increases, the overall design complexity of the package will increase and becomes hard to understand. It also increases the possible number of faults, changes and implementation effort required in a package. If a package has high value of TRFC/ARFC, this may be an indicator of misplaced methods. Designer has to review the design to identify these methods.
- Abstractness: Package that has low abstractness is more likely to have more faults than package with high Abstractness. This is because abstract classes do not contain implementation code for functions defined in this package.
- Polymorphism Factor: increasing total degree of methods overriding in the class inheritance tree in a package obviously increase it complexity. This in turn increases likelihood of faults in this package. In addition, increasing overriding methods will increase change and the implementation effort in this package.

Therefore, designers should put this into consideration when designing a package.

- Method Hiding Factor: increasing hidden (private) methods increases possibility of faults in this package. This is because private methods do not allow other classes to use them outside the class that defined them. Consequently, new methods need to be created, which increases implementation effort and the possibility of faults.
- Inheritance: the more of depth of inheritance tree and number of descendents classes in package, the more complicate the package will be and hence the more likely to have more faults.

6.1 Threats to Validity

The four threats to validity (conclusion, internal, construct and external) outlined in [64] is used in this section to describe the validity of the studies.

6.1.1 Conclusion Validity

Conclusion validity concerns issues that affect the ability to draw the correct conclusion about relations between the treatment and the outcome. The total number of packages that changed in GanttProject from release 2.0.6 to 2.0.7 was about 20% of the overall packages in 2.0.6 and about 10% from release 2.0.7 to 2.0.8. The statistical validity of this study is the size of the sample data (only 10% of packages changed in GanttProject) perhaps not enough for the statistic tests [65]. However, it was interesting to use case (as in GanttProject) where size of system releases (in term of LOC) has been relatively stable and see how change increase or decreases over releases.

6.1.2 Internal Validity

Internal validity is the degree to which conclusions can be drawn regarding the causal effect of independent variables on the dependent variable. The following possible threats have been identified:

1. Internal validity issues arise when there are errors in measurement. This is negated to an extent by the fact that the entire data collection process is automated via the JHawk and DiffDocpro tools. However, DiffDocpro does not calculate the changes at the package-level. It calculated them at the class level. Therefore, spread sheet was used to aggregate the changes at the package-level. This concern is alleviated by the cross check among the measures to identify abnormal values for any of the measures.
2. During the comparison between each two releases in Empirical Study 2, comments, blank lines and white space in line were excluded from the comparison process (option in DiffDocpro tool).
3. Since writing comments takes some effort, comments were taken in consideration when measuring the implementation effort (i.e. LOC) for a package.

6.1.3 Construct Validity

Construct validity refers to the extent to which the setting actually reflects the construct under study. It measures the degree to which the independent and dependent variables accurately measure the concepts they purport to measure. The independent variables used in these studies are martin related metrics, C&K and MOOD suites. As mentioned earlier, Martin metrics are well known metrics at the package-level. C&K and MOOD suites are considered the best object-oriented design metrics. C&K metrics are class-level metrics. They were adapted to be at package-level by aggregating them at package-level. The aggregated metrics give the overall view of C&K metrics at package-level. However, there might be better ways to redefine these metrics at the package-level. On the other hand, since MOOD metrics are system-level metrics, they were redefined to be at package-level by considering a package as small system and so measuring its attributes.

The dependent variables for the first empirical study are pre and post-release fault density and binary variable that indicate the presence of a fault in package. The faults were obtained from Zimmermann, Premraj and Zeller’s work [15]. They used very systematic procedure to calculated faults. In addition, cross checking among the measures was done to enhance the quality of fault dataset.

The density in the first and second studies was obtained by dividing the number of faults and the amount of changes in a package by total LOC for that package. This was done to limit the influence of the possible positive correlation between the package size and number of faults or amount of changes in that package. In the third study, LOC was used as a proxy for measuring implementation effort. LOC have been used as measure for effort in many researches, their advantages and drawbacks are well documented in [66]. LOC may not indicate the actual effort done on a system; this is because there are many tools that can generate tens of lines of code within few clicks. In addition, the implementing effort required for each LOC in a system is not consistence. For example, the required effort to define a variable is not as the one for creating for loop. However, measuring process of LOC is simple comparable to other measures: it can be specified unambiguously, so that when applied to the same code by different people, the results are guaranteed to be identical [66].

6.1.4 External Validity

External validity concerns the ability to generalize results outside the settings used. It is the degree to which the results of the research can be generalized to the population under study and other research settings. Our finding is limited to systems implemented using java language. However, since java shares the same concepts of object-oriented design (e.g. inheritance, coupling dependencies) with other OO languages, it is expected to have similar results when the empirical study is repeated with other systems implemented in other OO languages. However, this issue could be addressed as future work.

6.2 Major Contribution

1- Several OOD metrics at the package-level have been empirically investigated to determine the usefulness of these metrics in identifying and predicting faults of object-oriented packages.

2- Several OOD metrics at the package-level have been empirically investigated to determine the usefulness of these metrics in identifying and predicting change of object-oriented packages.

3- Several OOD metrics at the package-level have been empirically investigated to determine the usefulness of these metrics in estimating implementation effort of object-oriented packages.

4- Sets of well known design metrics (i.e. C&K and MOOD suites) have been redefined and implemented at the package-level to automate the extraction of these metrics.

6.3 Future Work

Several paths can be taken to extend on this work. Such paths include:

- Conducting additional empirical studies with different open sources and commercial systems to support findings of this thesis.
- Investigating the correlation of change-proneness, fault prediction and implementation effort estimation with other package-level metrics.
- In this study, linear regression was used to build models for change prediction, fault prediction and implementation effort estimation. It would also be of interest to

investigate the use of decision trees, neural network, support vector machine and other artificial intelligence techniques in order to increase the accuracy of the prediction models.

- Experiments might be done on the package-level metrics to determine proper threshold for the collected metrics. This threshold could be utilized by project managers to increase the quality of software design.

References

[1] R. Harrison, S. J. Counsell and R.V. Nithi, “An Evaluation of the MOOD Set of Object-Oriented Software Metrics,” *in IEEE Transactions on Software Engineering*, 1998, pp. 491-496.

[2] A. Koru and H. Liu, "Identifying and characterizing change-prone classes in two large-scale open-source products," *Journal of Systems and Software*, 2006.

[3] I. Jacobson, M. Christerson, P. Jonsson and G. Overgaard, *Object-Oriented Software Engineering: A Use-Case Driven Approach*: Addison- Wesley, 1992.

[4] Desmond F. D’Souza and Alan C. Wills. *Objects, Components and Frameworks with UML: The Catalysis Approach*: Addison-Wesley, 1998.

[5] P. Niemeyer, J. Knudsen, *Learning Java*, 3rd ed, 2005, pp. 38.

[6] V. Basili, L. Briand, and W. Melo, “A Validation of Object- Oriented Design Metrics as Quality Indicators,” *IEEE Transactions on Software Engineering*, vol. 22, no. 10, pp. 751-761, 1996.

[7] K. El Emam, S. Benlarbi, N. Goel and S. Rai, “A Validation of Object-Oriented Metrics,” *Technical Report ERB-1063*, National Research Council of Canada (NRC), 1999.

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

[9] F. Abreu, S. Esteves and M. Goulao, “The Design of Eiffel Programs: Quantitative Evaluation Using the MOOD Metrics,” in *Proceedings of TOOLS'96*, Santa Barbara, CA, USA, 1996.

[10] H. Melton, E. Tempero, ”The CRSS metric for package design quality,” *in Proceedings of the thirtieth Australasian conference on Computer science*, January 30-February 02, 2007, Ballarat, pp. 201-210.

[11**]** H. Kabaili , R. Keller , F. Lustman, Cohesion as Changeability Indicator in Object-Oriented Systems, in Proceedings of the Fifth European Conference on Software Maintenance and Reengineering, 2001, p.39.

[12] R. Martin, "Stability," *C+ + Report*, Vol. 9, No. 2 (Feb 1997).

[13] S. Roger, *Software Engineering*, Fifth edition, McGraw-Hill, 2000.

[14] S. Ducasse, M. Lanza and L. Ponisio, “Butterflies: A Visual Approach to Characterize Packages,” *in Proceedings of the 11th IEEE International Software Metrics Symposium (METRICS’05)*, IEEE Computer Society, 2005, pp. 70-77.

[15] T. Zimmermann, R. Premraj, and A. Zeller, “Predicting Defects for Eclipse,” *in PROMISE ’07: Proceedings of the Third Intrnational Workshop on Predictor Models in Software Engineering*, Washington, DC, USA, 2007, pp. 9+.

[16] M. Elish and D. Rine, "Investigation of Metrics for Object-Oriented Design Logical Stability," *in Proceedings of the Seventh European Conference on Software Maintenance And Reengineering (CSMR’03)*, 2003.

[17] F. Fioravanti, P. Nesi, and S. Perli, “Assessment of System Evolution Through Characterization,” *Proc. Int’l Conf. Software Eng. (ICSE ’98)*, 1998.

[18] K. Welker, W. Oman, Software Maintainability Metrics Models in Practise, 2004. [Online]. Available: http://www.stsc.hill.af.mil/crosstalk/1995/11/Maintain.asp.

[19] S. Henry and D. Kafura, “Software Structure Metrics Based on Information Flow,” *IEEE Transactions on Software Engineering*, vol. 7, no. 5, pp. 510- 518, Sept. 1981.

[20] D. Kafura and S. Henry, “Software Quality Metrics Based on Interconnectivity,” *J. Systems and Software*, vol. 2, pp. 121-131, Feb. 1982.

[21] B. Kitchenham, L. M. Pickard, and S.J. Linkman, “An Evaluation of Some Design Metrics,” *IEEE Software Eng. Journal*, vol. 5, no. 1, pp. 50- 58, Jan. 1990.

[22] S. R. Chidamber, D.P. Darcy, and C.F. Kemerer. “Managerial Use of Metrics for Object Oriented Software: An Exploratory Analysis,” *IEEE Transactions on Software Engineering*, pp. 629-639, 1998.

[23] W. Li and S. Henry, “Object-Oriented Metrics that Predict Maintainability,” *J. Systems and Software*, vol. 23, no. 2, pp. 111-122, 1993.

[24] R. Subramanyam and M. S. Krishnan, “Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects,” *IEEE Transactions on Software Engineering*, pp.297–310, 2003.

[25] A. B. Binkley and S. R. Schach, “Validation of the coupling dependency metric as a predictor of run-time failures and maintenance measures,” *in Proceedings of the International Conference on Software Engineering*, 1998, pp. 452–455.

[26] N. Nagappan, T. Ball, and A. Zeller, “Mining metrics to predict component failure,” *in Proceedings of the International Conference on Software Engineering (ICSE 2006)*, ACM, May 2006.

[27] A. Schröter, T. Zimmermann, A. Zeller, “Predicting Component Failures at Design Time,” *in Proceedings of the 5th International Symposium on Empirical Software Engineering (ISESE)*, Rio de Janeiro, Brazil, September 2006.

[28] P. Nesi and T. Querci, “Effort Estimation and Prediction of Object-Oriented Systems,” *Journal of Systems and Software*, pp. 89-102, 1998.

[29] R. Lind, K. Vairavan, “An experimental investigation of software metrics and their relationship to software development effort,” *IEEE Transactions on Software Engineering*, pp. 649–653.

[30] S. Yau and J. Collofello, “Design Stability Measures for Software Maintenance,” IEEE Transactions on Software Engineering, vol. SE-11, no. 9, pp. 849-856, Sep. 1985.

[31] M. D 'Ambros and M. Lanza, “Reverse engineering with logical coupling,” *in Working Conference on Reverse Engineering*, 2006, pp. 189 - 198.

[32] M. Wilhelm, S. Diehl,DependencyViewer, “A Tool for Visualizing Package Design Quality Metrics,” *IEEE*, pp.125-126, 2005.

[33] R. Reibing, “Towards a Model for Object-Oriented Design Measurement,” *Institute of Computer Science, University of Stuttgart*, 2002.

[34] P. Sandhu and H. Singh, "A Critical Suggestive Evaluation of C&K Metric," *in PACIS 2005 Proceedings*, 2005.

[35] G. Booch, *“Object-Oriented Analysis and Design with Applications”*, 2nd ed.: Benjamin Cummings, 1994.

[36] H. S. Chae, Y. R. Kwon, D. Bae, “A Cohesion Measure for Object-Oriented classes," *Software-Practice & Experience*, v.30, n.12, pp.1405-1431, Oct. 2000.

[37] C. Larman, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*, 3rd edition: Prentice-Hall, 2002.

[38] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy and W. Lorenses, *Object-Oriented. Modeling and Design*, Prentice Hall, 1991.

[39] F. Brito, E. Abreu and W. Melo, “Evaluating the Impact of Object- Oriented Design on Software Quality,” *Proc. Third Int’l Software Metrics Symp.*, 1996, pp. 90-99.

[40] N. Fenton and M. Neil, “Software metrics: roadmap,” *in ICSE – Future of SE Track* 2000, pp. 357–370.

[41] N. Fenton and S. L. Pfleeger, *Software Metrics: A Rigorous and Practical Approach.* London: International Thomson Computer Press, 1997.

[42] T. Mens and S. Demeyer, “Evolution Metrics,” *in Proceedings of the 4th international workshop on Principles of software evolution*, September 10-11, 2001.

[43] S. R. Chidamber, and R. F. Kemerer, “A metrics suite for object oriented design,” *IEEE Transactions on Software Engineering*, pp. 476-493, 1994.

[44] B. H. Sellers, *Object-Oriented Metrics-Measures of Complexity*: Prentice Hall, 1996.

[45] V. Basili and H. Rombach, "The TAME Project: Towards Improvement-Oriented Software Environment," *IEEE Transactions on Software Engineering*, vol. 14, pp. 758-773, 1988.

[46] http://www.eclipse.org

[47] http://www.st.cs.uni-sb.de/softevo/

[48] www.virtualmachinery.com/JHawkprod.htm

[49] M. H. Olague, L. H. Etzkorn, S. Gholston and S. Quattlebaum, “Empirical Validation of Three Software Metrics Suites to Predict Fault-Proneness of Object-Oriented Classes Developed Using Highly Iterative or Agile Software Development Processes,” *IEEE Transactions on Software Engineering*, pp. 402-419, 2007.

[50] L. C. Briand , J. Wüst , J. W. Daly and D. V. Porter, “Exploring the relationship between design measures and software quality in object-oriented systems,” *Journal of Systems and Software*, v.51 n.3, pp. 245-273, 2000.

[51] http://www.minitab.com/en-US/products/minitab/documentation.aspx

[52] http://www.cs.waikato.ac.nz/ml/weka/

[53] S. D. Conte, H. E. Dunsmore, and V. Y. Shen, Software engineering metrics and models: *Benjamin-Cummings Publishing Co.*, Inc., 1986.

[54] B. A. Kitchenham, L. M. Pickard, S. G. MacDonell, and M. J. Shepperd,"What accuracy statistics really measure," IEEE Software, vol. 148, pp. 81-85, 2001.

[55] J. Davis and M. Goadrich, "The Relationship Between Precision-Recall and ROC Curves," *in 23rd international conference on Machine learning*, 2006, pp. 233-240.

[56] I. H. Witten and E. Frank, *Data Mining: Practical Machine Learning Tools and Techniques,* second ed. San Francisco: Morgan Kaufmann, 2005.

[57] M. M. Thwin and T. S. Quah, "Application of Neural Networks for Software Quality Prediction Using Object-Oriented Metrics," *Journal of Systems and Software,* vol. 76, pp. 147 - 156, 2005.

[58] S. El Emam, N. Benlarbi and S. Rai, “The Confounding Effect of
Class Size on the Validity of Object-Oriented Metrics,” IEEE Transactions on
Software Engineering, vol. 27, pp. 630-650, 2001.

[59] W.M. Evanco, “Comments on ‘the Confounding Effect of Class Size on the Validity of Object-Oriented Metrics’,” Transactions on Software Engineering. vol. 29, no. 7, pp. 670-672, July 2003.

[60] G. Kanji, *100 Statistical Tests. Thousand Oaks*, CA: SAGE Publications, p. 110, 1999.

[61] http://www.ganttproject.biz/

[62] P. Mohagheghi, R. Conradi, O. M. Killi and H. Schwarz, “An Empirical Study of Software Reuse vs. Defect Density and Stability,” *in Proc. 26th Int’l Conference o Software Engineering (ICSE’2004),* 23-28 May 2004, Edinburgh, Scotland, pp. 282-291.

[63] http://www.prestosoft.com/ps.asp?page=edpexam

[64] T. D. Cook and D. T. Campbell, *Quasi-Experimentation: Design and Analysis Issues for Field Settings*. New York: Houghton Mifflin Company, 1979.

[65] L. Briand, K. El Emam, and S. Morasca, “Theoretical and empirical validation of software product measures,” *International Software Engineering Research Network,* 1995.

[66] J. S. Poulin, *Measuring Software Reuse: Principles, Practices, and Economic Model:* Addison-Wesley, 1997.

Appendix A – Best Subsets

Table 35: Best subsets of package-level metrics in Empirical Study 1

Eclipse Model			Metrics		Train 2.0 &Test 2.1	Train 2.0-2.1 & Test 3.0	Common
Fault	Pre – release	PPreF	All		No Class, A ,D, I, CE, Ca, TLCOM, ARFC, ACBO, TNDC, ANDC, MIFP and CFP	Ca, Ce, TLCOM, AWMP, ANDC , AHFP and CFP	Ca, Ce, ANDC and TLCOM
			Martin		D, I, Ca and Ce	No. Class, A ,Ca and Ce	Ca and Ce
			C&K	All	TLCOM, TCBO and ARFC	TLCOM, TWMP, TRFC, TCBO, ANDC and AWMP	TLCOM and TCBO
				Total	TLCOM, TWMP, TRFC and TNDC	TLCOM, TWMP, TRFC and TCBO	TLCOM, TRFC and TWMP
				Average	ARFC and ANDC	ALCOM, ARFC, ANDC and AWMP	ARFC AND ANDC
			MOOD		AHFP and CFP	AHFP, AIFP, MIFP and CFP	AHFP and CFP
		PreFD	All		I, Ca, TLCOM, AWMP, MIFP, AHFP, AIFP and PFP	A,NOA, ALCOM, AWMP and TWMP	AWMP
			Martin		I, Ca, and No. Class	No. Class, NOA and Ce	No. CLASS
			C&K	All	ARFC	TLCOM, TRFC, TNDC, ANDC and TCBO	None
				Total	TLCOM, TWMP, TNDC and TCBO	TLCOM, TRFC, TNDC and TCBO	TLCOM, TNDC and TCBO
				Average	ARFC	AWMP and ADIT	None
			MOOD		AHFP and CFP	AIFP, MHFP, AHFP, CFP and PFP	CFP
	Post –	PPostF	All		No Class, D, CE, TLCOM and ARFC	No. Class, A, Ca, Ce, TLCOM, AWMP, ANDC, TNDC, ALCOM, ACBO and CFP	No. Class, Ca, Ce, and

	release						TLCOM.
			Martin		No Class, A,D and Ce	No. Class, A ,Ca and Ce	No. Class, A and Ce
			C&K	**All**	TLCOM, TWMP, TRFC, ARFC and TCBO	TLCOM, TWMP, TRFC, TCBO,TNDC, ANDC and ACBO	TLCOM, TWMP, TRFC and TCBO
				Total	TLCOM, TWMP, TRFC and TCBO	TLCOM, TWMP, TRFC and TCBO	TLCOM, TWMP, TRFC and TCBO
				Average	ARFC and ADIT	ARFC, and ANDC	ARFC
			MOOD		MHFP	MHFP, MIFP,PFP and CFP	MHFP
			All		A, TLCOM, ARFC, MIFP, AHFP, MIFP and PFP	No. Class, Ce, TRFC, ALCOM, ARFC and MIFP	ARFC and MIFP
		PostFD	**Martin**		Ca and Ce	No. Class, A and Ce	Ce
			C&K	**All**	ARFC and ADIT	TRFC, TDIT,TCBO,TNDC, AWMP and ARFC	None
				Total	TRFC	TRFC and TWMP	TRFC
				Average	ARFC and ADIT	ARFC and LCOM	ARFC
			MOOD		MIFP, MHFP, AHFP, CFP and PFP	MIFP, MHFP and PFP	MIFP, MHFP and PFP

Table 36: Best subsets of package-level metrics in Empirical Study 2

Model	Suite		Metrics
ChD-Eclipse (Training using 2.1 and Testing using 3.0)	**All**		ARFC, ADIT,TNDC and ANDC
	Martin		A
	C&K	**All**	ARFC, ADIT,TNDC and ANDC
		Total	TNDC ,TDIT and TRFC
		Average	ARFC & ADIT
	MOOD		MHFP
ChD GanttProject (Training using 2.0.7 and Testing using 2.0.8)	**All**		AHFP, Ce, NO. Class. NOA
	Martin		No. Classes, I,D, Ca, Ce and NOA
	C&K	**All**	ALCOM, ACBO, ADIT and ANDC
		Total	TLCOM, TRFC,TWMP, TDIT, TNDC and TCBO
		Average	ALCOM, ACBO, ADIT and ANDC
	MOOD		MHFP, AHFP

Table 37: Best subsets of package-level metrics in Empirical Study 3

Eclipse Model	Suite		Metrics
Eclipse (2.0) Effort	All		No. Classes, A, TLCOM, ALCOM,TRFC,TWMP, TDIT, ADIT ,PFP, AHFP and NOA
	Martin		No. Classes, D, Ca, Ce and NOA
	C&K	All	TLCOM,ALCOM,TRFC,ARFC,TWMP,TNDC and ANDC
		Total	TLCOM, TRFC,TWMP, TNDC and TCBO
		Average	ARFC, AWMP, ADIT and ANDC
	MOOD		MIFP, CFP and AHFP
GanttProject(2.0.6) Effort	All		No. Classes, TCBO,TRFC, MHFP and AHFP
	Martin		No. Classes, Ca, Ce and NOA
	C&K	All	TRFC,TCBO,TNDC and ANDC
		Total	TRFC, TNDC and TCBO
		Average	ARFC and AWMP
	MOOD		PFP

Appendix B – Regression models

Table 38: PPreF model for Eclipse (Train 2.0 & 2.1 and Test 3.0)

Eclipse		Pre-Release Faulty / Not Faulty Package						Post-Release Faulty / Not Faulty Packages					
Suites	**Metrics**	**All-Suites**	**Martin**	**MOOD**	**C&K All**	**C&K Total**	**C&K Avg**	**All-Suites**	**Martin**	**MOOD**	**C&K All**	**C&K Total**	**C&K Avg**
Martin	**No. Classes**	0	0.0234	0	0	0	0	-0.0229	-0.0283	0	0	0	0
	A	0	-0.9986	0	0	0	0	1.345	1.3361	0	0	0	0
	I	0	0	0	0	0	0	0	0	0	0	0	0
	D	0	0	0	0	0	0	0	0	0	0	0	0
	Ca	0.0126	0.0139	0	0	0	0	-0.0056	-0.0068	0	0	0	0
	Ce	0.0421	0.0343	0	0	0	0	-0.0307	-0.0274	0	0	0	0
	NOA	0	0	0	0	0	0	0	0	0	0	0	0
C&K	**TLCOM**	0.0037	0	0	0.002	0.0014	0	0	0	0	-0.0045	-0.0037	0
	ALCOM	0	0	0	0	0	0.0061	-0.0144	0	0	0	0	0
	TRFC	0	0	0	0.0018	0.0012	0	0	0	0	-0.0021	-0.0023	0
	ARFC	0	0	0	0	0	0.0435	0	0	0	-0.0202	0	-0.0344
	TWMP	0	0	0	-0.0016	0.0019	0	0	0	0	0.0027	0.002	0
	AWMP	0.1181	0	0	0	0	0	-0.0583	0	0	-0.013	0	0
	TDIT	0	0	0	0	0	0	0	0	0	0	0	0
	ADIT	0	0	0	0	0	0	0	0	0	0	0	0
	TNDC	0	0	0	0	0	0	-0.0103	0	0	-0.013	0	0
	ANDC	0.4573	0	0	0.3412	0	0.6906	-0.0876	0	0	-0.053	0	-0.2827
	TCBO	0	0	0	0.0011	0.0007	0	0	0	0	-0.0003	-0.0003	0
	ACBO	0	0	0	0	0	0	0.0038	0	0	0.0018	0	0
MOOD	**MHFP**	0	0	0	0	0	0	0	0	-0.9455	0	0	0
	MIFP	0	0	0.1625	0	0	0	0	0	-0.5761	0	0	0
	PFP	0	0	0	0	0	0	0	0	-0.0515	0	0	0
	CFP	-1.9205	0	-2.5037	0	0	0	0.112	0	1.9494	0	0	0
	AIFP	0	0	0.1544	0	0	0	0	0	0	0	0	0
	AHFP	0.7602	0	0.8229	0	0	0	0	0	0	0	0	0
Intercept		-1.2269	0.2814	0.4223	-0.6974	0.045	-0.6929	1.2908	0.705	0.4291	1.389	0.9637	1.0491

The model at Suite S is given by the followings equation:

M = No. Classes × S + A × S + I × S + D × S + Ca × S + Ce × S + NOA × S + TLCOM × S +ALCOM × S + TRFC × S + ARFC × S + TWMP × S + AWMP × S + TDIT × S + ADIT × S + TNDC × S + ANDC × S + TCBO × S + ACBO × S + MHFP × S + MIFP × S + PFP × S + CFP × S + AIFP × S + AHFP + Intercept$_S$

Table 39: Fault density model for Eclipse (Train 2.0 and Test 2.1)

		Pre-Release Fault Density						Post-Release Fault Density					
Suites	Metrics	All-Suites	Martin	MOOD	C&K All	C&K Total	C&K Avg	All-Suites	Martin	MOOD	C&K All	C&K Total	C&K Avg
Martin	No. Classes	0	0	0	0	0	0	0	0	0	0	0	0
	A	0	0	0	0	0	0	0	0	0	0	0	0
	I	-0.0142	-0.0175	0	0	0	0	0	0	0	0	0	0
	D	0	0	0	0	0	0	0	0	0	0	0	0
	Ca	-0.0001	-0.0002	0	0	0	0	0	0	0	0	0	0
	Ce	0	0	0	0	0	0	0	-0.0002	0	0	0	0
	NOA	0	0	0	0	0	0	0	0	0	0	0	0
C&K	TLCOM	0	0	0	0	0	0	0	0	0	0	0	0
	ALCOM	0	0	0	0	0	0	0	0	0	0	0	0
	TRFC	0	0	0	0	0	0	0	0	0	0	0	0
	ARFC	0	0	0	-0.0003	-0.0003	-0.0003	0	0	0	-0.0002	0	0
	TWMP	0	0	0	0	0	0	0	0	0	0	0	0
	AWMP	-0.0007	0	0	0	0	0	0	0	0	0	0	0
	TDIT	0	0	0	0	0	0	0	0	0	0	0	0
	ADIT	0	0	0	0	0	0	0	0	0	-0.0106	0	-0.0072
	TNDC	0	0	0	0	0	0	0	0	0	0	0	0
	ANDC	0	0	0	0	0	0	0	0	0	0	0	0
	TCBO	0	0	0	0	0	0	0	0	0	0	0	0
	ACBO	0	0	0	0	0	0	0	0	0	0	0	0
MOOD	MHFP	0	0	0	0	0	0	-0.01	0	0	0	0	0
	MIFP	0	0	-0.0084	0	0	0	0	0	0	0	0	0
	PFP	-0.0191	0	-0.0191	0	0	0	-0.0094	0	0	0	0	0
	CFP	0	0	0	0	0	0	0	0	0.1587	0	0	0
	AIFP	0	0	0	0	0	0	0	0	0	0	0	0
	AHFP	-0.0085	0	-0.0084	0	0	0	0	0	-0.0053	0	0	0
Intercept		0.0417	0.0287	0.025	0.0243	0.0243	0.0243	0.0081	0.0076	0.0034	0.0126	0.0057	0.006

The model at Suite S is given by the followings equation:

M = No. Classes × S + A × S + I × S + D × S + Ca × S + Ce × S + NOA × S + TLCOM × S +ALCOM × S + TRFC × S + ARFC × S + TWMP × S + AWMP × S + TDIT × S + ADIT × S + TNDC × S + ANDC × S + TCBO × S + ACBO × S + MHFP × S + MIFP × S + PFP × S + CFP × S + AIFP × S + AHFP + Intercept$_S$

Table 40: Fault density model for Eclipse (Train 2.0 &2.1 and Test 3.0)

		Pre-Release Fault Density						Post-Release Fault Density					
Suites	Metrics	All-Suites	Martin	MOOD	C&K All	C&K Total	C&K Avg	All-Suites	Martin	MOOD	C&K All	C&K Total	C&K Avg
Martin	No. Classes	0	-0.0001	0	0	0	0	0.0001	0	0	0	0	0
	A	0.0483	0	0	0	0	0	0	0	0	0	0	0
	I	0	0	0	0	0	0	0	0	0	0	0	0
	D	0	0	0	0	0	0	0	0	0	0	0	0
	Ca	0	0	0	0	0	0	0	0	0	0	0	0
	Ce	0	-0.0005	0	0	0	0	0	-0.0002	0	0	0	0
	NOA	-0.0011	0	0	0	0	0	0	0	0	0	0	0
C&K	TLCOM	0	0	0	0	0	0	0	0	0	0	0	0
	ALCOM	0	0	0	0	0	0	0	0	0	0	0	0
	TRFC	0	0	0	0	0	0	0	0	0	0	0	0
	ARFC	0	0	0	0	0	0	-0.0002	0	0	-0.0001	0	-0.0001
	TWMP	0	0	0	0	0	0	0	0	0	0	0	0
	AWMP	-0.0012	0	0	0	0	-0.0013	0	0	0	0	0	0
	TDIT	0	0	0	0	0	0	0	0	0	0	0	0
	ADIT	0	0	0	0	0	-0.0241	0	0	0	0	0	0
	TNDC	0	0	0	0.0001	0	0	0	0	0	0	0	0
	ANDC	0	0	0	0.0051	0	0	0	0	0	0	0	0
	TCBO	0	0	0	0	0	0	0	0	0	0	0	0
	ACBO	0	0	0	0	0	0	0	0	0	0	0	0
MOOD	MHFP	0	0	-0.0481	0	0	0	0	0	-0.0126	0	0	0
	MIFP	0	0	0	0	0	0	-0.0037	0	-0.0053	0	0	0
	PFP	0	0	-0.024	0	0	0	0	0	-0.0089	0	0	0
	CFP	0	0	0	0	0	0	0	0	0	0	0	0
	AIFP	0	0	-0.0198	0	0	0	0	0	0	0	0	0
	AHFP	0	0	0	0	0	0	0	0	0	0	0	0
Intercept		0.0244	0.0254	0.0339	0.0223	0.0187	0.0322	0.014	0.0086	0.0104	0.0087	0.0057	0.0084

The model at Suite S is given by the followings equation:

M = No. Classes × S + A × S + I × S + D × S + Ca × S + Ce × S + NOA × S + TLCOM × S +ALCOM × S + TRFC × S + ARFC × S + TWMP × S + AWMP × S + TDIT × S + ADIT × S + TNDC × S + ANDC × S + TCBO × S + ACBO × S + MHFP × S + MIFP × S + PFP × S + CFP × S + AIFP × S + AHFP+ Intercept$_S$

Table 41: Change density model for Eclipse

Change density & package-level metrics							
Suites	Metrics	All-Suites	Martin	MOOD	C&K All	C&K Total	C&K Average
Martin	No. Classes	0	0	0	0	0	0
	A	0	0	0	0	0	0
	I	0	0	0	0	0	0
	D	0	0	0	0	0	0
	Ca	0	0	0	0	0	0
	Ce	0	0	0	0	0	0
	NOA	0	0	0	0	0	0
C&K	TLCOM	0	0	0	0	0	0
	ALCOM	0	0	0	0	0	0
	TRFC	0	0	0	0	0	0
	ARFC	0	0	0	0	0	0
	TWMP	0	0	0	0	0	0
	AWMP	0	0	0	0	0	0
	TDIT	0	0	0	0	0	0
	ADIT	0	0	0	0	0	0
	TNDC	0	0	0	0	0	0
	ANDC	0	0	0	0	0	0
	TCBO	0	0	0	0	0	0
	ACBO	0	0	0	0	0	0
MOOD	MHFP	0	0	2.0308	0	0	0
	MIFP	0	0	0	0	0	0
	PFP	0	0	0	0	0	0
	CFP	0	0	0	0	0	0
	AIFP	0	0	0	0	0	0
	AHFP	0	0	0	0	0	0
Intercept		1.2397	1.2397	0.9597	1.2397	1.2397	1.2397

The model at Suite S is given by the followings equation:

M = No. Classes × S + A × S + I × S + D × S + Ca × S + Ce × S + NOA × S + TLCOM × S +ALCOM × S + TRFC × S + ARFC × S + TWMP × S + AWMP × S + TDIT × S + ADIT × S + TNDC × S + ANDC × S + TCBO × S + ACBO × S + MHFP × S + MIFP × S + PFP × S + CFP × S + AIFP × S + AHFP+ Intercept$_S$

Table 42: Change density model for GanttProject

Change density & packages-level metrics							
Suites	**Metrics**	**All-Suites**	**Martin**	**MOOD**	**C&K-All**	**C&K-Total**	**C&K-Average**
Martin	**No. Classes**	0	0	0	0	0	0
	A	0	0	0	0	0	0
	I	0	0	0	0	0	0
	D	0	0	0	0	0	0
	Ca	0	0	0	0	0	0
	Ce	-0.004	0	0	0	0	0
	NOA	0.0041	0	0	0	0	0
C&K	**TLCOM**	0	0	0	0	0	0
	ALCOM	0	0	0	1.6355	0	1.6355
	TRFC	0	0	0	0	0	0
	ARFC	0	0	0	0	0	0
	TWMP	0	0	0	0	0	0
	AWMP	0	0	0	0	0	0
	TDIT	0	0	0	0	0	0
	ADIT	0	0	0	2.6437	0	2.6437
	TNDC	0	0	0	0	0	0
	ANDC	0	0	0	-0.5204	0	-0.5204
	TCBO	0	0	0	0	0	0
	ACBO	0	0	0	-0.0736	0	-0.0736
MOOD	**MHFP**	0	0	2.9523	0	0	0
	MIFP	0	0		0	0	0
	PFP	0	0	0	0	0	0
	CFP	0	0	0	0	0	0
	AIFP	0	0	0	0	0	0
	AHFP	0	0	0.9696	0	0	0
Intercept		0.0521	0.2115	-0.7617	-0.5573	0.2115	-0.5573

The model at Suite S is given by the followings equation.

M = No. Classes × S + A × S + I × S + D × S + Ca × S + Ce × S + NOA × S + TLCOM × S +ALCOM × S + TRFC × S + ARFC × S + TWMP × S + AWMP × S + TDIT × S + ADIT × S + TNDC × S + ANDC × S + TCBO × S + ACBO × S + MHFP × S + MIFP × S + PFP × S + CFP × S + AIFP × S + AHFP+ Intercept$_S$

Eclipse 2.0		**Table 43: Implementation effort model for Eclipse**					
Suites	**Metrics**	**All-Suites**	**Martin**	**MOOD**	**C&K-All**	**C&K-Total**	**C&K-Average**
Martin	**No. Classes**	-19.6197	146.0989	0	0	0	0
	A	-265.5915	0	0	0	0	0
	I	0	0	0	0	0	0
	D	0	-1175.454	0	0	0	0
	Ca	0	21.5698	0	0	0	0
	Ce	0	-79.4332	0	0	0	0
	NOA	-11.6841	-180.6308	0	0	0	0
C&K	**TLCOM**	-2.1542	0	0	-2.9586	-2.7051	0
	ALCOM	0	0	0	13.2221	0	0
	TRFC	1.3433	0	0	0.297	0.6523	0
	ARFC	0	0	0	16.9447	0	065.312
	TWMP	5.4541	0	0	6.9676	6.4178	0
	AWMP	0	0	0	-32.0398	0	0
	TDIT	11.6729	0	0	13.6052	0	0
	ADIT	0	0	0	-484.1904	0	5169.7273
	TNDC	0	0	0	0	0	0
	ANDC	0	0	0	0	0	544.1254
	TCBO	0	0	0	0	0	0
	ACBO	0	0	0	0	0	0
MOOD	**MHFP**	0	0	0	0	0	0
	MIFP	0	0	2006.2502	0	0	0
	PFP	224.1131	0	0	0	0	0
	CFP	0	0	-5254.758	0	0	0
	AIFP	0	0	0	0	0	0
	AHFP	-491.5567	0	-888.2309	0	0	0
Intercept		338.7079	73.9807	1084.296	-225.5899	-169.5533	0-1048.469

The model at Suite S is given by the followings equation:

M = No. Classes × S + A × S + I × S + D × S + Ca × S + Ce × S + NOA × S + TLCOM × S +ALCOM × S + TRFC × S + ARFC × S + TWMP × S + AWMP × S + TDIT × S + ADIT × S + TNDC × S + ANDC × S + TCBO × S + ACBO × S + MHFP × S + MIFP × S + PFP × S + CFP × S + AIFP × S + AHFP+ Intercept$_S$

GanttProject 2.0	Table 44: Implementation effort model for GanttProject						
Suites	**Metrics**	**All-Suites**	**Martin**	**MOOD**	**C&K-All**	**C&K-Total**	**C&K-Average**
Martin	**No. Classes**	3.8552	0.0087	0	0	0	0
	A	0	0	0	0	0	0
	I	0	0	0	0	0	0
	D	0	0	0	0	0	0
	Ca	0	0	0	0	0	0
	Ce	0	0.5015	0	0	0	0
	NOA	0	1.5374	0	0	0	0
C&K	**TLCOM**	0	0	0	0	0	0
	ALCOM	0	0	0	0	0	0
	TRFC	1.9263	0	0	2.0165	2.0242	0
	ARFC	0	0	0	0	0	39.5952
	TWMP	0	0	0	0	0	0
	AWMP	0	0	0	0	0	141.3918
	TDIT	0	0	0	0	0	0
	ADIT	0	0	0	0	0	0
	TNDC	0	0	0	7.1523	4.0553	0
	ANDC	0	0	0	-49.6819	0	0
	TCBO	-1.0413	0	0	-0.999	0.998	0
	ACBO	0	0	0	0	0	0
MOOD	**MHFP**	241.1874	0	0	0	0	0
	MIFP	0	0	0	0	0	0
	PFP	0	0	1597.3074	0	0	0
	CFP	0	0	0	0	0	0
	AIFP	0	0	0	0	0	0
	AHFP	-54.6178	0	0	0	0	0
Intercept		18.1359	2.1667	422.3599	15.0732	0.998	-847.1114

The model at Suite S is given by the followings equation:

M = No. Classes × S + A × S + I × S + D × S + Ca × S + Ce × S + NOA × S + TLCOM × S +ALCOM × S + TRFC × S + ARFC × S + TWMP × S + AWMP × S + TDIT × S + ADIT × S + TNDC × S + ANDC × S + TCBO × S + ACBO × S + MHFP × S + MIFP × S + PFP × S + CFP × S + AIFP × S + AHFP+ Intercept$_S$

VITA

Ali Hashem Ali was born on 1981. He obtained his Bachelor of Science (B.S.) degree with a second honor in Computer Science from King Fahd University of Petroleum and Minerals (KFUPM), Dhahran, Saudi Arabia in January 2005. His COOP project "Housing Units portal" was awarded as one of the best university projects in the 2004 annual exhibition.

Since 2005, He is working in Arabian Advanced Systems, which is the leading IT regional company in providing Knowledge solutions and services. It gave him the exposure to real world information technology deployments, inside and outside Saudi Arabia. While working in the company, He was pursuing the master's degree in computer science as a part time student.

Ali received his Master degree in Computer Science (with a second honor) from KFUPM in June 2010. His research areas are in software engineering and in artificial intelligent.

Email: ali.alsaadi7@gmail.com