A COMPARISON OF CK AND MARTIN'S PACKAGE METRIC SUITES IN PREDICTING PACKAGE REUSABILITY IN OPEN SOURCE OBJECT-ORIENTED SOFTWARE

KHALED ALHADI MEFTAH

A dissertation submitted in partial
fulfillment of the requirement for the award of the
Degree of Master of Computer Science (Software Engineering)

Faculty of Computer Science and Information Technology
Universiti Tun Hussein Onn Malaysia

MARCH 2016

ABSTRACT

Packages are units that organize source code in large object-oriented systems. Metrics used at the package granularity level mostly characterize attributes such as complexity, size, cohesion and coupling. Many of these characterized attributes have direct relationships with the quality of the software system being produced. Empirical evidence is required to support the collection of measures for such metrics; hence these metrics are used as early indicators of such important external quality attributes. This research compared the CK and Martin's package metric suites in order to characterize the package reusability level in object-oriented software. Comparing the package level of metrics suites as they measure an external software quality attribute is supposed to help a developer knows which metric suite can be used to effectively predict the software quality attribute at package level. In this research two open source Java applications, namely; jEdit and BlueJ were used in the evaluation of two package metrics suites, and were compared empirically to predict the package reusability level. The metric measures were also used to compare the effectiveness of the metrics in these package metrics suites in evaluating the reusability at the package granularity level. Thereafter metric measures of each package were normalized to allow for the comparison of the package reusability level among packages within an application. The Bansiya reusability model equation was adapted as a reusability reference quality model in this research work. Correlation analysis was performed to help compare the metrics within package metrics suites. Through the ranking of the package reusability level, results show that the jEdit application has 30% of its packages ranked with a very high reusability level, thus conformed to the Pareto rule (80:20). This means that the jEdit application has packages that are more reusable than packages in the BlueJ application. Empirically, the Martin's package coupling metric `Ce` with an `r` value of 0.68, is ranked as having a positive strong correlation with `RL`, and this has distinguished the Martin's package metrics suite as an effective predictor of package reusability level from the CK package metrics suite.

ABSTRAK

Pakej adalah unit yang menguruskan kod sumber dalam sistem berorientasikan objek yang besar. Metrik yang digunakan di peringkat butiran pakej kebanyakannya mempunyai ciri-ciri seperti kekompleksan, saiz, perpaduan dan gandingan. Kebanyakan karakter atribut ini mempunyai hubungan secara langsung dengan kualiti sistem perisian yang dihasilkan. Bukti empirikal diperlukan untuk menyokong koleksi pengukuran metrik tersebut, dan oleh sebab itu metrik ini digunakan sebagai penunjuk awal sifat-sifat penting kualiti luaran. Kajian ini membandingkan metrik suite pakej CK dan metrik suite pakej Martin untuk menilai tahap guna semula pakej di dalam perisian berorientasikan objek. Melalui perbandingan tahap pakej metrik suite yang mengukur sifat luar kualiti perisian sepatutnya dapat membantu pembangun untuk mengetahui metrik suite yang mana yang boleh digunakan dengan berkesan dalam meramal sifat kualiti perisian di peringkat pakej. Dalam penyelidikan ini, dua sumber terbuka aplikasi Java, iaitu jEdit dan BlueJ telah digunakan dalam penilaian dua pakej metrik suite, dan dibandingkan secara empirikal untuk meramal tahap boleh gunasemula pakej tersebut. Selain itu, pengukuran metrik telah digunakan untuk membandingkan keberkesanan metrik ini di dalam pakej metrik suite, bagi menilai tahap boleh gunasemula di peringkat pakej butiran. Selepas itu, langkah-langkah pengukuran metrik setiap pakej telah dinormalkan bagi membolehkan perbandingan tahap boleh gunasemula pakej antara setiap pakej dalam setiap aplikasi. Persamaan model boleh gunasemula oleh Bansiya telah disesuaikan sebagai model kualiti rujukan boleh gunasemula dalam kerja-kerja penyelidikan ini. Analisis korelasi telah dijalankan bagi membantu membandingkan metrik dalam metrik pakej suite. Melalui kedudukan tahap boleh gunasemula pakej, keputusan menunjukkan bahawa aplikasi jEdit mempunyai 30% daripada pakejnya yang memiliki tahap boleh gunasemula yang sangat tinggi, mematuhi apa yang telah ditentukan dalam peraturan Pareto (80:20). Ini bermaksud bahawa aplikasi jEdit mempunyai pakej boleh gunasemula yang lebih tinggi berbanding pakej dalam aplikasi BlueJ. Secara empirikal, gandingan metrik `Ce` dengan nilai `r` dari 0.68 dalam pakej Martin telah dikelaskan sebagai mempunyai korelasi positif yang kukuh dengan `RL`, dan ini membezakan metrik suite pakej Martin sebagai peramal tahap boleh gunasemula pakej yang berkesan dari metrik suite pakej CK.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 Research Overview

Software reuse is a process in which a new software application is built from existing software (Shiva and Shala, 2007). The concept is not new in programming and can be traced back to the advent of the concept of subroutines and functions. However, it came to be widely known during the NATO Software Engineering Conference in 1968 when it was formally proposed by Douglas McIlroy of Bell laboratories (Frakes and Kyo, 2005). Since then the concept has been bolstered by the tremendous growth in both software development and programming paradigms that support software reuse in principle.

Shiva and Shala (2007), reported that in software industry, corporations such AT&T, HP, Tata Consultancy Services (TCS) and many others have greatly embraced the concept of reusability in software development, due to the promises that the concept offers. The promises that software reuse offer include but not limited to; reduced software development cost, low maintenance and more so increased productivity on the part of the developer (Meena and O'Brien, 2011). Software reuse increases the productivity of the developer and can be of great help especially in an environment that software features become obsolete so fast like in the open source community.

Since consumers of such communities require new features very rapidly and bearing in mind that the consumer base is large, then features requested might tend to be enormous. Thus, when such features are effected on the current releases of a software product. Thus, a new product release is in inevitable (Wu, 2006). This is no surprise to the current state in software development, where it has becomes a known fact that software constantly changes; since the business environment in which software operates in is also very dynamic in nature (Stephens and Rosenberg, 2007). As was foreseen by Lehman (1979), software should change to meet the requirements of a changing environment and hence realizing its potential too. Hence, software is said to evolve with changing needs in the operating business environment.

As software changes, its design structure also changes, this can happen either during the software development phase or at the maintenance phase. Structural changes may increase software complexity, which consequently hamper its external product properties such as understandability, maintainability and internally may also affect design properties such as reusability, modifiability and modularity (Vasa, 2010).

In addition, developers productivity need to be enhanced and software reusability is essential in this direction. This require a purposeful embracement of developing software that is reusable as its being developed. To accomplish that objective, it is important to have control of reusability as a design property and as software is being developed.

1.2 Research Background

Due to the way developers in open source environment handle software changes as they maintain and enhance features of current releases, it might not be easy to maintain a very sound structured design that can support some of the mentioned software quality attributes especially reusability (Brown and Booch, 2002). Reusability is said to have a very considerable effect on general software quality (Goel and Bhatia, 2013). Though software changes can be a bit challenging especially when there is a requirement such as maintaining good software quality on reusable software components. So too be able to achieve this, time and effort is required to always assess the quality of reusable components (Goel and Bhatia, 2013).

Assessing a software product quality attribute, require the reference to a quality model, which defines the properties that are to be measured in order to make a decision on the quality level of such an attribute. It is important to note that software reusability as a software quality attribute was missing in ISO 9126 and as such was recently introduced into the ISO 25010 software quality model (ISO/IEC 25010, 2011). In ISO 25010, software reusability is one of the sub-characteristics of the external quality attribute maintainability, and therefore can only be measured using internal properties of the software product (ISO/IEC 25010, 2011). Since, it was missing even in the earlier ISO 9126 standard, researchers specified their own reusability assessment models including the one proposed by Poulin (1994), which are oftenly referred to as extended quality models (Thapar et al., 2014).

Using the extended quality models, various quality factors related to software reusability have been measured, which include: portability, flexibility, understandability, independence, stability and many others (Poulin, 1994), mostly assessing the reusability level of software products. In this section, a review of some of the related research in terms of assessment of software reusability is done and a summary of these related research studies are given in Table 1.1. This research work is concern with recent studies that have assessed software reusability in open source community.

Fazal et al. (2012), illustrate an evolutionary case study to evaluate a proposed conceptual reusability model which was used to study the reusability of software during evolution. An evaluation of the model was conducted using a case study, where two open source projects were used, to evaluate the metrics in the said reusability attribute model. Various releases of the said open source projects were evaluated, both at the class and method granularity levels using an assorted set of reusability metrics as per the proposed reusability attribute model. The model of proposed attribute reusability proposed various new metrics that were used in the assessment of software reusability. Except for the consideration of scale for the various metrics that were used in coming up with the new metrics, there is no mathematical rigour used to justify the coefficients of the various metrics in the equations of the new metrics.

Another research work done by Goel and Bhatia (2013), evaluated metrics in object-oriented software written in C++ based on three inheritance features with an objective of finding out which of these features have more impact on the software reusability. Inheritance features investigated include: multilevel, multiple and hierarchical inheritance; three C++ programs which had these features were used in this study. This study used Chidamber and Kemerer (CK) metrics suite in measuring the inheritance features, later proposed new metrics for measuring the said inheritance features. However, the metrics were not validated, thus the researchers used their own intuition to suggest the aggregation of some of their metrics. From the original CK metrics and their derived metrics, it was found out that multi-level inheritance has more impact on reusability among the three features corroborating the principle of multi-level inheritance as a good indicator for reusability.

Ampatzoglou et al. (2011), did an empirical investigation on the reusability of design patterns and packages in open source projects, in order to help developers have a starting point in white box software reuse. The main concern in that research work was to characterize the reusability of these projects, identify reusable design patterns and packages that can be of help to developers that require to use them in other projects. The research work only investigated one release of each software project that was studied, and the granularity level was at package and design pattern level. Design level metrics from the Quality Model for Object Oriented Design (QMOOD) were used in characterising the reusability level of the software investigated in the study.

Another related research study is by Makkar et al. (2012), specified an inheritance metric that is better than Depth of Inheritance (DIT) or Depth of Inheritance of Class (DITC), with a reusability perspective, with the claim that current inheritance metrics are primitive and only give rough estimate of the inheritance of a class or lack validation support. The metric is theoretically validated through Weyuker’s nine axioms, showing a good coverage of the inheritance concept in a class. Though, the metric might be good in terms of its rigour in validation and its measurement, it only covers one aspect of reusability, therefore other metrics need to be identified and used to accomplish the assessment of reusability as a whole.

A few of the research studies discussed this section considered investigating software reusability as software evolves and more, so all are not identify the corresponding software components which can be reused based on the software projects that were studied as shown in Table 1.1. Moreover, they are lacking in decision to advice the developers mentioned at their projects ways of how to reuse the software components present in their projects, so that they can increase productivity as they come up with new releases

Table 1. 1: Assessment of Software Reusability

Authors	Case Study	No. of Projects	Granularity Level	Quality Model	Evolution
(Fazal et al., 2012)	Java Open Source Projects	2 (Jasmin, pBeans) 6 versions and 10 versions	Class, method	Proposed a quality model with new metrics proposed	Yes
(Goel and Bhatia, 2013)	C++ programs	3 programs (3 classes)	Class	CK metrics used, new metrics were also proposed	No
(Ampatzoglou et al., 2011)	Java Open Source Projects	29 projects	Design Patterns, packages	QMOOD (Design level metrics)	No
(Makkar et al., 2012)	Reusability Metric specification	No project	Class	DIT, DITC	No

Moreover, from the summary of the research studies as presented in Table 1.1, it is evident that different studies have touched a different granularity levels when it comes to reuse component. The class has been the most consistent granularity level used in most studies discussed, so this can largely be attributed to the metrics used. One other hand, aspect that was observed in many studies has appeared with a proposal of new reusability metrics, through the mathematical rigour required to validate them if they were lacking in serious way.

1.3 Problem Statement

Comparing package level metrics suites "as they measure an external software quality attribute" is supposed to help a developers to know which metric suite can be used to effectively predict the software quality attribute at package level. Nonetheless, software changes that are implemented due to addition feature and maintenance, that affect at the internal structures of the software, hence directly affecting design properties such as reusability, modularity and modifiability.

This research work, would like to compare two package level metrics suites on the design property, reusability. The purpose is to understand which of the two suites can effectively predict the reusability level of packages within an object-oriented software. The two package level metrics suites that were used to compare package reusability level, were; the Martin’s metrics suite and the CK metrics suite. The reusability computation index equation by (Bansiya, 2002) was adapted as the reusability quality model in this research work.

1.4 Research Aim and Objectives

The aim of this study is therefore to compare two package level metrics suites on the design property and reusability; in order to understand which of the two suites can effectively predict the reusability level of packages within object-oriented software. The aim of this study was achieved through the following objectives:

i. Measure package reusability properties in two open source Java software using CK package metrics suite.
ii. Measure the package reusability properties in two open source Java software using Martins package metrics suite.
iii. Evaluate and compare package reusability level within the two open source Java software.
iv. Evaluate the effectiveness of the two package metric suites in measuring package reusability level.

1.5 Scope of the Study

Some open-source software undergo a tremendous changes within a very short time, because of the environment they operate under; this puts a lot of constraint to the community developers, especially when coming up with new releases. Therefore, if the developers can be helped in terms of knowing which parts of their software can easily be reused; it can go a great length in reducing the development time of new features as requested by users.

This research would like to fill-in such a gap by first being able to characterize the reusability levels of packages in a mature open-source object-oriented software using two well-known package metrics suites. Two open source object-oriented software namely jEdit and BlueJ written in Java were used; the two open source projects were considered in this research since the two are from the same problem domain as Java programming editors. Three measurement tools, namely; Metrics 1.3.8, JHawk and JDepend were used together with one reverse engineering tool called ObjectAid; all these tools were used as plug-ins in the Eclipse Java IDE. This measurement activity will help the developers to know the reusability of the packages in question, to help control deterioration of its structure in future changes. The object-oriented granularity level of concern in this research is the pegged at the package level, this is because the class is a very low level element for consideration in terms of reuse. The two metrics suite that were used are; Martin's (2006) package metrics suite and CK (1991; 1994) package metrics suite.

A comparison of the measurements from the reusability properties was done to know which of the two metric suites could effectively predict package reusability in an object-oriented software. These measurements can be used to allow the developers to know which package are more reusable than the others in terms of the metrics values. As an indicator, this will go a long way to control the design properties, because each package was evaluated on at least a minimum of three measures, letting the developer understands the packages structural elements, such as coupling, cohesion and its interface size. The comparison of the metrics from the two metrics suites was done using correlation analysis which using the measures of reusability properties obtained from two open Java source software.

1.6 Significance of the Study

Measuring a software product is essential in software engineering for commonly two purposes as described by Fenton (Fenton and Bieman, 2015); first, to understand the level of the quality attribute in the product, secondly to be able to control it as its being developed. This study will first help the developer to address basic issues that is essential to reusability, such as: understanding the package reusability level in the software product as it is being developed; ultimately then know which of the packages are likely to be more reusable than the others. It will also help the developer to know which of the two package metric suites is more effective in predicting package reusability in object-oriented software. These are some of the concerns that this research tried to handle through the results obtained from measuring the reusability level of two open source case study, namely; jEdit and BlueJ using the two package metrics suites.

Measurement in a software product is an indicator of an attribute quality level within a software product, which sends a signal to the developer that the software product is moving towards the right direction in terms of the specified design goals or not. In this research the package reusability level was measured to ascertain its level in the software product, hence go a long way to help the developer to understand the reusability level of each package in two open source Java software. This will go a long way in guiding the design of the software product being developed and let the developer reuse such packages in developing other software products in the same domain. The two package reusability measures will also help to characterize the structural design properties of the software by measuring the cohesion and coupling among packages, before computing the reusability level of the package. Through the consistency of the metric values from each metric suite the developer will be able to know which of the two metrics suites can effectively predict the reusability level.

1.7 Chapter Summary

This chapter was able to introduce the concept of reusability from a measurement point of view and its relationship to software evolution, this was covered in Section 1.1. In Section 1.2, a brief overview of the related work that of concern, that show the missing link in literature was discussed to help the researcher map the scope of this research work. That was followed immediately by a description of the problem statement in Section 1.3, whose objectives were specified in Section 1.4. After which, the scope of the research work was specified in the Section 1.5. In Section 1.6, a justification of the study was fully discussed.

CHAPTER 2

LITERATURE REVIEW

2.1 Introduction

A common phenomenon among software development houses is that, the process of software development is always faced with issues in terms of cost, not meeting delivery time, delivery of software that does not fully meet user's functionality or has runtime errors among other issues (Nazareth and Rothenberger, 2004). Software reuse has been recognized by many researchers (Tripathy and Naik, 2014; Keswani et al., 2014; Spoelstra et al., 2011; Agresti, 2011) as a way that can help to solve some of the software crisis issues mentioned; especially when the software's evolution rate is a bit faster.

User's needs always increase continuously, and to accommodate those needs the softwares should also change. It is therefore necessary for developers to craft software that is flexible enough, which can accommodate change when required, afford extendibility with or without rework. Such a description fits so well to the characteristics of software product property called software reusability. Most software systems that have exhibited some success in the market, should be able to add new system capabilities when required - that is as it evolves; this can be possible if software modules written earlier can be reused (Stierna and Rowe, 2003). Hence, software evolution and software reusability are related in one way or another; software that is reusable can easily evolve as user needs or change in requirements.

As software evolves then the developer needs to also be careful on how its internal properties also change to be able to facilitate later reuse of some software artefacts. Thus, it is essential for the developer to understand and control the level of some internal product attributes such as reusability, modularity, modifiability among others; which have a direct effect on external product properties. To be able to characterize and hence control the reusability level as software evolves, reusability level measurement is required for each software release as it evolves.

The need for the software measurement in order to characterize and control a software product attribute, specifically for reusability was briefly discussed in sub section 1.2. That section also gave an overview of the existing research that has been conducted in the field of software reusability, which was then be further expounded in this sub section 2.3 and 2.4. To achieve the objectives stated in sub section 1.4, a package reusability metrics are required; these are discussed in details sub section 2.3.1. Before that discussion, a detailed coverage of what software reusability entails is first done in the Section 2.2 that follow.

2.2 Software Reusability

Software reusability is a software product property that specifies the ability of a software artefact to be reused in developing new software (ISO/IEC 25010, 2011). This property hangs on the process of software reuse, which involves crafting new software systems by using already existing software artifacts instead of developing them from scratch (De’ and Rao, 2013; Heinemann et al., 2011; Lucrédio et al., 2008). In most large software systems, it is always possible to find reuse opportunities, this can be confirmed by the facts given by Nazareth and Rothenberger (2004), that suggested about 75% of functions within a program can be reused in other programs. That fact is also confirmed what was suggested by Joyce (1990), that roughly about 15% of all source code is unique; meaning the other code can be easily reused in other contexts. Thus, in most software systems, that are large enough, there should be software artifacts that can be reused either within that system or externally.

The estimates suggested here are mostly based on software systems that are similar, which in most cases will be from similar application domains, also may be from a specific software systems domain or are using similar programming languages (Nazareth and Rothenberger, 2004; Sommerville and Kotonya, 1998). There are many examples of software reuse in practice, stemming from as easy as functions as well as subroutines, where a repetitive task is put together in a sub program, then called when required; though not a common example. Other examples include libraries or packages, application programming interfaces (API), frameworks, design patterns among other examples (Sojer and Henkel, 2011; Sommerville, 2010; Postmus and Meijler, 2008; Brown and Booch, 2002). In Section 2.2.3, a detailed classification of the software reuse is done, and in each case examples are provided.

The open source community, has provided an avenue for software reuse as acknowledged by Brown and Booch (2002), libraries, APIs, frameworks and many others have been provided with access to the source code also; making it easier for developers to directly reuse or rework some solutions to suit their needs. In the open source community, frameworks and libraries for many recurring tasks are provided for in software repositories such as Source Forge, Linux, Apache and many others (Sojer, 2011; Brown and Booch, 2002). Examples of popular frameworks that have been used in market with tremendous impact include web application frameworks like Apache, Ant, Hibernate, Struts; programming platforms like Eclipse, Netscape among others (Heinemann et al., 2011). The contribution of these frameworks and libraries have touched many software development and some may not be able to survive without such software in use. Open source software is driven towards reuse, since they have to keep up with the rapid pace of dynamic change in user requirements and effecting new user requirements (Brown and Booch, 2002). In as much as these repositories are good, they tend to be abused in terms of software reuse.

With the advent of the many online code repositories offering access to source code, some software developers who copy and paste such code into their software systems may perceive that as software reuse (Barzilay and Urquhart, 2014). While this process make the developer not reinvent the wheel as it were, it is a process of copying and pasting, which culminates to code duplication, leading to bad code seems as described by Fowler et al. (1999). As good as it is to the developer at the time of doing it, it should not be confused with software reuse; it is code duplication. As the access to such repositories is there, source code reuse should be in form of the impacts that are modular in form such as libraries, packages among others. Such modular impacts when required in development, can then be used in that form or reworked but not copied. This is one of many issues which faced in case of software reuse, many examples are given in sub section 2.2.2; before then a discussion of the need for reuse and its advantages is done in sub section 2.2.1.

2.2.1 Software Reuse

Software reuse has been widely accepted, has seen a lot of progress made in the software industry, in relation to reducing the issues that are always associated with the phenomena of software crisis. In literature, reusing already existing code has a lot of benefits that include:

i. Increased Productivity

   The use of already existing software artifacts increase the productivity of software developers, since new products are not written from scratch and can therefore use their time to write that part of the software that really is unique (Rothenberger et al., 2003; Lim, 1994). This in a way expedites the creation of a new software product, hence reduces the time to develop and considerable reduces the cost of the software produced (Meena and O'Brien, 2011). This is a very important reuse benefit since one of the issues that software development is battling out in practice is increasing production of software and ultimately reducing the cost of the product. This was one of the concerns of McIlroy and his colleagues, in their seminal work (McIlroy et al., 1968) on software reuse coined the concept software factory in 1968.

ii. Improved Software Quality

   An existing software impacts that can be reused presents a tested piece of software, hence when reused assures that the product realized will also conform to the quality level of the impacts (Barzilay and Urquhart, 2014; Lim, 1994). As said earlier, the part of software that is unique is always relatively small compared to the parts that are reused, hence if a software product is built from existing software artifacts the end product will be of high quality (Meena and O'Brien, 2011). Again this property considerably reduces the quality issues that of concern in software development, since reusable artifacts have a proven quality level.

iii. Enhanced Code and Design Standards

Using existing frameworks, present a proven system architecture that the developer should conform to, hence in a way will push the developer to conform to a coding and design standard that is with reference to the framework being used (Heinemann et al., 2011; Brown and Booch, 2002). Making it clear that developers that are using the similar frameworks will follow a certain way of reasoning in terms of coding and design standards. Hence, developers within such domain of frameworks can easily share reusable software artifacts and to some extent follow all design principles within that domain.

iv. Facilitates Knowledge Reuse

Software reuse is one way of sharing software development knowledge amongst developers (Sojer, 2011). This was further enhanced by introduction of design patterns (Gamma et al., 1994) which involves expert developers sharing their expert experience with novice designers through patterns. Through frameworks some design patterns are now part of developer’s toolkit and it is a must in one way or the other to follow these design patterns to fully utilize the framework.

The four purposes which also be viewed as benefits of software reuse are not an exhaustive list but form a basis of almost all other objectives or reasons why software reuse is important to the software developer and software development house. It is obvious it is not all roses to software reuse, challenges of software reuse are presented in the next sub section.

2.2.2 Challenges of Software Reuse

Despite the benefits that can be accrued by reusing software artifacts, software reuse has still not reached its full potential due to many challenges it faces (Spoelstra et al., 2011). This is evident in any field that has a lot to accomplish in varying contexts of use, again the issue of it being a young concept and the complexity of the artifact that it is dealing with; brings all sorts of challenges. This is attributed to the fact that earlier on the software reuse community concentrated on solving technical issues related to software reuse and leaving the other concerns (Meena and O'Brien, 2011). However, current research in software reuse (Keswani et al., 2014; Holmes and Walker, 2013; Agresti, 2011; Spoelstra et al., 2011) is geared towards finding solutions to the non-technical issues due to their importance in making sure that software reuse delivers its promises. Some of the challenges faced in enabling software reuse include:

i. Organizational Challenges

A software development house that intends to adopt software reuse as part of its software development process has first to think about how it is going to change its software development process to accommodate software use either in developing software artifacts to be reused or developing with software reusable artifacts (Morisio et al., 2002). This can be a very big drift from the normal way the organization is used to developing software, hence care must be taken to embrace the concept of software reuse in the organization. In an organization, managing the different software development units so as to embrace software reuse across board needs a lot of input from the organization in the form of infrastructure required for software reuse (Frakes and Kyo, 2005; Schmidt, 1999). There will need for the organization to manage the various reusable software artifacts that either produced within or are purchased from external sources in order to reap the benefits of software reuse.

ii. Economic Challenges

There are two very popular strategies in implementing software reuse, its either one is developing with reuse or developing for reuse (Sojer, 2011; Mili et al., 1995); in either case the software organization must put the right infrastructure for either or both. For one to achieve the right infrastructure in terms of tools, developers and software development process, the organization must commit a lot of investments into it (Jalender et al., 2010; Schmidt, 1999). This requires an organization that values software reuse, knows that it is an asset to the organization rather than an expenditure without return of investment. Management should look at it as a long term investment not short term as such to reap the promised benefits of software reuse (Postmus and Meijler, 2008; Sherif and Vinze, 2003). Therefore require a lot of commitment towards embracing the concept of software reuse. One challenge facing most organization is the lack of an economic model to implement the software reuse concept within organizations (Jalender et al., 2010).

iii. Human Challenges

The introduction of software reuse in an organization can have adverse effect on developers as human beings. It first comes in as a shock to them, that they are not good enough for the task, it can very easily lower the morale of developers (Sherif and Vinze, 2003). Organizations therefore should introduce development with reuse in their institutions after having thoroughly trained their man power and sensitize them on use of reusable software artifacts so that developers do not see them as objects that have to replace their jobs. The other concern is about trust, which can bring about the not-invented-here syndrome. Some developers may not just believe and trust that someone else can do a software impacts that can be good enough to fit their specification well, and they would always feel better when they develop from scratch (Meena and O'Brien, 2011; Sojer, 2011). Developers like any human beings have ego issues and some have the know-it-all syndrome which can be a great challenge to software reuse in an organization.

The other issue would be managing developers to accept change in software development process; it is obvious that software reuse would fit it very well with a top-down approach, changing developers thinking to fit into this approach while already used with other approach can be monumental in nature (Sojer, 2011; Sherif and Vinze, 2003). Issue of losing developers to competitors is always looming when changes are effected into an organization in a haste; it is always important to always consult developers on a way forward when it comes to the introduction of drastic changes in either management or development process especially if there is a major drift from the norm such as developing with reuse.

iv. Technical Challenges

Among the technical challenges include selection of reusable software artifacts from within or external. For a software impacts to be reusable, it should meet a bare minimum quality properties that include portability, reliability, adaptability, platform independence, among others (Tripathy and Naik, 2014). Therefore, the challenge is usually some of these properties can be measured amongst many artifacts after they have met the basic functionality property (Jalender et al., 2010). It is usually a big technical challenge to developer. Apart from that the software development process should also change or be tweaked to conform to development with software reuse; that has operationalization concerns to be dealt with, change management to handle amongst developers, tools to support the same must be sourced among other concerns (Sojer, 2011; Jalender et al., 2010; Sherif and Vinze, 2003).

A new software architecture that can be used in software development has to be adopted to suit the reusable software artifacts as such. This can cause some rifts amongst developers with the issue of learning curve, hence sometime losing some developers to other competitors in the industry (Sherif and Vinze, 2003). Tools to support is another concern to either developing reusable software artifacts or develop with reuse is essential for software reuse to take effect in an organization (Sojer, 2011). Right development tools must be sourced to support software reuse. Software artifacts are supposed to be portable to any operating environment theoretically but some artifacts may require some rework before they can fit into a new operating platform and therefore increase the maintenance cost in terms of time used to rework it (Sherif and Vinze, 2003). As the software system evolve the reusable software artifact should be able to evolve with, however most of artifacts are either completely redone or have to be rework extensively to fit in the evolved software system.

2.2.3 Classification of Software Reuse

Software reuse comes in all forms and perspectives; such that in literature the different forms or perspectives converge in one way or the other. In most cases, the forms or perspectives are just naming perspectives, however they mean the same thing from another perspective. Prieto-Diaz (1993), tried to classify these different forms and perspectives into at least six perspectives or facets as shown in Table 2.1, namely; substance, scope, mode, technique, intention, and product.

Table 2. 1: Software Reuse Facets (Prieto-Diaz, 1993)

Substance	**Scope**	**Mode**	**Technique**	**Intention**	**Product**
Ideas, concepts	Vertical	Planned, systematic	Compositional	Black-box, as-is	Source code
Artifacts, components		Ad-hoc, opportunistic	Generative	White-box, modified	Design
Procedures, Skills	Horizontal				Specifications
					Objects Text Architectures

The facet substance, defines the nature of reusable items; which can be in the form of ideas or concepts, artifacts or components, and procedures or skills. While reuse can also be viewed from the scope facet that refers to the extent in which a reusable artifact can utilized. If an artifact can only be reused within the same application domain then it is said to exhibit vertical reuse; whereas if it can be reused across dissimilar application domains then the artifact is said to exhibit horizontal scope reuse (Prieto-Diaz, 1993). Vertical scope reuse is the common form of software reuse, such that in general software reuse is attributed to this form or facet, in which case a reuse artifact is mostly reused within the same application domain (Shiva and Shala, 2007; Nazareth and Rothenberger, 2004; Stierna and Rowe, 2003).

The other way reuse can be viewed from is by the way it is conducted by an organization or an individual and it is classified to be of the mode facet. Software reuse can be implemented either using ad hoc or opportunistic reuse which boils down to unplanned implementation of reuse; whereas systematic reuse is a planned and managed software reuse, where software artifacts as well as processes are reused in a predefined manner not haphazardly (De' and Rao, 2013; Spoelstra et al., 2011). Another form of software reuse is based on the approach used to implement reuse, it commonly known as by-technology.

Software reuse can also be view from the intention of reuse, how are the reuse artifacts going to be reused is another perspective; can be viewed into two either white-box or black-box reuse. In white-box reuse, the source code of the reuse impacts is included in the project files of the software being developed; whereas black-box reuse, is reuse in which the software to be reused is included in the software system in binary form, in which case the code is used as is (Heinemann et al., 2011; Postmus and Meijler, 2008). The concern of this research is in the intention of reuse; where the assessment will be concentrated on the reusability level of the versions which can facilitate white-box reuse. The other form of reuse defines what work products are reused.

2.3 Measuring Software Reusability

The importance of measurement in general is always summarized by the statement; it is not easy to control what cannot be measured, which is often quoted in many measurement research work (Abran, 2010). In software engineering there it is generally agreed that it is essential for software processes and products to be measured so that stakeholders can be able to understand objectively the software system state as its being developed or during maintenance (Suresh et al., 2012; Malhotra and Khanna, 2013). In doing that, it can establish proper control over software development, where software metrics can provide the necessary feedback that is required to facilitate corrective actions as well as track the outcome of the same. A software metric here is meant to refer to a quantitative measure of the degree to which a software abstraction has a given attribute.

In this research work, an assessment of software reusability on an open source software was carried out, evaluating the reusability at the package granularity level. A package is defined as a group of classes that has a related set of facilities, they are usually related by a purpose or by the application domain (Horton, 2011). Classes in the same package have special access privileges with respect to one another and may be designed to work together. Several studies related to analysing packages in object-oriented systems that evaluate various quality attributes have been carried showing the importance the package as module within these systems as they grow in size (Ampatzoglou et al., 2011; Elish et al., 2011; Zhao et al., 2015). The concern in this research was to measure the reusability of packages level using two metric suites as discussed earlier and compare the results obtained. The two package metrics suites used in this research work are discussed in the two sub sections that follow:

2.3.1 Package Metrics Suites

As reiterated by Elish et al. (2011), there are three very popular package metrics suites that exists, namely Martin's package metrics suite (Martin and Martin, 2006), MOOD (Metrics for Object-Oriented Design) package metric suite (Harrison et al., 1998) and the CK package metrics suite (Chidamber and Kemerer, 1994). In this research work only two were considered for the comparison, namely; Martin's package metrics suite and the CK package metrics suite, these two are described in this section.

A. Martin’s Package Metrics Suite

Martin’s package metrics suite (2006) has specified six popular and widely used package metrics. The metrics in this suite measure various package structural elements which include its package size, cohesion, coupling and stability. The specification of the six metrics are as follows:

i. Number of Classes (NC)

The NC metric for a package is defined as the number of concrete and abstract classes (and interfaces) in the package. It is a measure of package size.

ii. Relational Cohesion (H)

The H can be represented as the average number of internal relationships per class in a component. Let R be the number of class relationships that are internal to the component (i.e., that do not connect to classes outside the component. Let NC be the number of classes within the component). The extra 1 in the formula prevents $H = 0$ when $NC = 1$ and represents the relationship that the package has to all its classes: $= \frac{R+1}{NC}$.

iii. Afferent Couplings (Ca)

The Ca metric for a package is defined as the number of other packages that depend on classes within the package. It measures the incoming dependencies (fan-in).

iv. Efferent Couplings (Ce)

The Ce metric for a package is defined as the number of other packages that the classes in the package depend on. It measures the outgoing dependencies (fan-out).

v. Instability (I)

The I metric for a package is defined as the ratio of efferent coupling (Ce) to total coupling ($Ce + Ca$) for the package; such that $I = Ce/(Ce + Ca)$. This metric is an indicator of the package's resilience to change. The range for this metric is zero to one, with zero indicating a completely stable package and one indicating a completely unstable package.

vi. Distance (D)

The D metric for a package is defined as the perpendicular distance of the package from the idealized line ($A + I = 1$). This metric is an indicator of the package's balance between abstractness and stability. A package squarely on the main sequence is optimally balanced with respect to its abstractness and stability. The range for this metric is zero to one, with zero indicating a package that is coincident with the main sequence and one indicating a package that is as far from the main sequence as possible.

B. Chidamber and Kemerer Package Metrics Suite

The second package metrics suite considered in this research work is by Chidamber and Kemerer (CK) package metrics suite (1991; 1994). It consists of six class-level metrics that were aggregated to package level using average. They measure several structural package properties that include size or complexity, coupling, cohesion and inheritance. These metrics are defined as follows:

i. Average Weighted Methods per Class ($AWMC$)

Weighted Methods per Class (WMC) is defined as the sum of the cyclomatic complexities of all methods defined in a class. The AWMC metric for a package is the average of the WMC values of the classes in the package.

ii. Average Coupling Between Object Classes ($ACBO$)

Coupling between Object classes (CBO) is defined as the number of classes to which a class is coupled. The $ACBO$ metric for a package is the average of the CBO values of the classes in the package.

iii. Average Response for a Class ($ARFC$)

Response for a Class (RFC) is defined as number of methods in the set of all methods that can be invoked in response to a message sent to an object of a class. The $ARFC$ metric for a package is the average of the RFC values of the classes in the package.

iv. Average Lack of Cohesion in Methods ($ALCOM$)

Lack of Cohesion in Methods ($LCOM$) is defined as the number of pairs of methods in a class using no attributes in common, minus the number of pairs of methods that do. If this difference is negative, $LCOM$ is set to zero. The $ALCOM$ metric for a package is the average of the $LCOM$ values of the classes in the package.

v. Average Depth of Inheritance ($ADIT$)

Depth of Inheritance (DIT) is defined as the maximum length from a class to the root class in the inheritance tree. The $ADIT$ metric for a package is the average of the DIT values of the classes in the package.

vi. Average Number of Children ($ANOC$)

Number of Children (NOC) is defined as the number of immediate subclasses of a class. The $ANOC$ metric for a package is the average of the NOC values of the classes in the package.

2.3.2 Reusability Measurement Reference Model

Each package metric suite provides a set of package metrics that can be measure individual package structural properties like size, which is the number of classes in the package, which may not have a direct meaning to the proposed general aim of this research. To be able to accomplish this aim and hence be able to achieve the corresponding objectives of this research work, the reusability index computation equation from the Hierarchical Model for Object-Oriented Design Quality Assessment (Bansiya and Davis, 2002) will be used in this research work. The reusability index computation equation (Bansiya and Davis, 2002) is defined as follows:

$$\text{Reusability} = -0.25*\text{Coupling}+0.25*\text{Cohesion}+0.5*\text{Messaging}+0.5*\text{Design Size} \qquad (1)$$

The object-oriented design quality model defined by Bansiya and Davis mainly considered the reusability at class level, requiring three structural properties at class level namely coupling, cohesion and messaging. The same equation is adapted in this research to compute the reusability at package level using the corresponding package level metrics provided in the suites above.

Software measurement is usually undertaken with reference to a corresponding software quality model (Thapar et al., 2014), which defines the measures to be undertaken for each software attribute and also give meaning to each corresponding metric attached to a measure. It with knowledge in mind that this research adapted equation (1) as its reusability measurement reference model to measure and interpret the results obtained in this research work. From equation (1), the structural elements that of importance are coupling, cohesion, messaging and design size. These structural elements are defined at class level in the Hierarchical Model for Object-oriented Quality Assessment, the reason being that the class was the module in that model. In this research work, the module of interest is the package, hence a specification of the corresponding design properties and metrics that fit the package as a module is required. Table 2.1 specify the corresponding design properties.

Table 2.2: Design Property Definitions

Design Property	**Definition**
Design Size (DS)	A count of the number of classes and interfaces in a package.
Coupling (CG)	Assesses the degree of interdependency between packages. The number of packages a package depends on.
Cohesion (CN)	Assesses the degree of relatedness of classes within a package.
Interface Size (IS)	A count of the number of public classes that are available as services to other packages. This is a measure of the services that a package provides.

From these definitions the reusability reference model can be rewritten to fit the design properties specified in Table 2.2 as follows:

$$\text{Reusability} = -0.25*CG+0.25*CN+0.5*IS+0.5*DS \qquad (2)$$

2.4 Comparing Package Metrics Suites

There are a limited number of studies that have explored the relationships between package-level metrics and external software quality attributes. In a previous research work by Elish (2010), explored the relationships between Martin metrics and package understandability. He found correlation between all Martin metrics, except the Ce metric, and the effort required to understand a package. In addition, it was observed that considering only the package size as input to prediction models for package understandability is not enough. Improved prediction could be achieved by considering other structural properties in addition to size.

Zimmermann et al. (2007), collected the faults data (i.e. the numbers of pre-release and post-release faults), as well as some complexity metrics for every package and file in the Eclipse releases 2.0, 2.1 and 3.0. They constructed logistic regression models and linear regression models to predict the post-release fault proneness and the number of post-release faults in packages respectively. The models were built as a function of some complexity metrics only. They concluded from their experiment that these prediction models are far from being perfect, and suggested the investigation of better indicators than complexity metrics (Zimmermann et al., 2007).

Later on Elish et al. (2011) empirically evaluated and compared the predictive power of three metrics suites (Martin, MOOD and CK) for pre-release and post-release fault prediction in packages of object-oriented systems through a case study of Eclipse. Seven different multivariate linear regression models were constructed, using different combinations of the three metric suites, and tested across releases of Eclipse. It was observed that the prediction models that include subset of Martin metrics achieved competitive accuracy, and that they outperformed the models that do not include them in predicting the number of pre-release faults in packages, and also in predicting the number of post-release faults across releases. In their research work, a recommendation for future work, elicited the thinking of how this research work could also compare at least two package metrics with another maintainability sub-characteristic reusability.

In Table 2.4 a summary of the discussed previous work is presented and compares it with what is done in this research work.

Table 2.3: Comparing Package Metrics Suites Related Research Work

Research work	**External software quality**	**Package metrics suites**	**Case studies**
This work	Package reusability	Martin and CK	jEdit and BlueJ
Elish et al. (2011)	Package faults	Martin, MOOD and CK	Eclipse
Zimmermann et al. (2007)	Package faults	Complexity metrics	Eclipse
Elish (2010)	Package understandability	Martin's suite	XGen and Jakrata ECS

This summary of related research work has been done using three distinguishing properties, namely external software quality, package metrics suites and case studies. It can be seen from Table 2.3, two of the related research work concentrated on package faults as the software quality attribute, while the third has dwelt on understandability which is different from what this research work is trying to achieve that is with reusability. One research work, has already compared the three

REFERENCES

Abran, A. (2010). *Software Metrics and Software Metrology*. John Wiley & Sons, Inc.

Agresti, W. W. (2011). Software Reuse: Developers' Experiences and Perceptions. *Journal of Software Engineering and Applications*. 4(01)**,** 48.

Ampatzoglou, A., Kritikos, A., Kakarontzas, G. and Stamelos, I. (2011). An empirical investigation on the reusability of design patterns and software packages. *Journal of Systems and Software*. 84(12)**,** 2265-2283.

Bansiya, J. and Davis, C. G. (2002). A hierarchical model for object-oriented design quality assessment. *Software Engineering, IEEE Transactions on*. 28(1)**,** 4-17.

Barzilay, O. and Urquhart, C. (2014). Understanding reuse of software examples: A case study of prejudice in a community of practice. *Information and Software Technology*. 56(12)**,** 1613-1628.

Brown, A. W. and Booch, G. (2002). Reusing Open-Source Software and Practices: The Impact of Open-Source on Commercial Vendors. In Gacek, C. (Ed.) *Software Reuse: Methods, Techniques, and Tools*. (pp. 123-136). Springer Berlin Heidelberg.

Chidamber, S. R. and Kemerer, C. F. (1991). *Towards a metrics suite for object oriented design*. ACM.

Chidamber, S. R. and Kemerer, C. F. (1994). A metrics suite for object oriented design. *Software Engineering, IEEE Transactions on*. 20(6)**,** 476-493.

De', R. and Rao, R. A. (2013). Open Source Reuse and Strategic Imperatives. In Sim, S. E. and Gallardo-Valencia, R. E. (Eds.) *Finding Source Code on the Web for Remix and Reuse*. (pp. 187-204). Springer New York.

Elish, M. O. (2010). Exploring the Relationships between Design Metrics and Package Understandability: A Case Study. Program Comprehension (ICPC), 2010 IEEE 18th International Conference on.June 30 2010-July 2 2010. 144-147.

Elish, M. O., Al-Yafei, A. H. and Al-Mulhem, M. (2011). Empirical comparison of three metrics suites for fault prediction in packages of object-oriented systems: A case study of Eclipse. *Advances in Engineering Software*. 42(10)**,** 852-859.

Fazal, E. A., Mahmood, A. K. and Oxley, A. (2012). An evolutionary study of reusability in Open Source Software. Computer & Information Science (ICCIS), 2012 International Conference on.12-14 June 2012. 967-972.

Fenton, N. and Bieman, J. (2015). *Software Metrics: A Rigorous and Practical Approach, Third Edition*. CRC Press.

Fowler, M., Beck, K., Brant, J. and Opdyke, W. (1999). *Refactoring: Improving the Design of Existing Code*. Westford, MA - USA: Addison Wesley Longman, Inc.

Frakes, W. B. and Kyo, K. (2005). Software reuse research: status and future. *Software Engineering, IEEE Transactions on*. 31(7)**,** 529-536.

Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1994). *Design patterns: elements of reusable object-oriented software*. Pearson Education.

Goel, B. M. and Bhatia, P. K. (2013). Analysis of reusability of object-oriented systems using object-oriented metrics. *SIGSOFT Softw. Eng. Notes*. 38(4)**,** 1-5.

Harrison, R., Counsell, S. J. and Nithi, R. V. (1998). An evaluation of the MOOD set of object-oriented software metrics. *Software Engineering, IEEE Transactions on*. 24(6)**,** 491-496.

Heinemann, L., Deissenboeck, F., Gleirscher, M., Hummel, B. and Irlbeck, M. (2011). On the Extent and Nature of Software Reuse in Open Source Java Projects. In Schmid, K. (Ed.) *Top Productivity through Software Reuse*. (pp. 207-222). Springer Berlin Heidelberg.

Holmes, R. and Walker, R. J. (2013). Systematizing pragmatic software reuse. *ACM Trans. Softw. Eng. Methodol.* 21(4)**,** 1-44.

Horton, I. (2011). *Ivor Horton's Beginning Java*. John Wiley & Sons.

ISO/IEC 25010 (2011). *Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models*. BS ISO/IEC 25010. BS ISO/IEC.

Jalender, B., Gowtham, N., Kumar, K. P., Murahari, K. and Sampath, K. (2010). Technical impediments to software reuse. *Int. J. Eng. Sci. Technol*. 2(11)**,** 6136-6139.

Joyce, E. J. (1990). Reusable software: passage to productivity? *Management information systems*. (pp. 203-207). Scott, Foresman \&amp; Co.

Keswani, R., Joshi, S. and Jatain, A. (2014). Software Reuse in Practice. Advanced Computing & Communication Technologies (ACCT), 2014 Fourth International Conference on.8-9 Feb. 2014. 159-162.

Lehman, M. M. (1979). On understanding laws, evolution, and conservation in the large-program life cycle. *Journal of Systems and Software*. 1(0)**,** 213-221.

Lim, W. C. (1994). Effects of reuse on quality, productivity, and economics. *Software, IEEE*. 11(5)**,** 23-30.

Lucrédio, D., dos Santos Brito, K., Alvaro, A., Garcia, V. C., de Almeida, E. S., de Mattos Fortes, R. P. and Meira, S. L. (2008). Software reuse: The Brazilian industry scenario. *Journal of Systems and Software*. 81(6)**,** 996-1013.

Makkar, G., Chhabra, J. K. and Challa, R. K. (2012). Object oriented inheritance metric-reusability perspective. Computing, Electronics and Electrical Technologies (ICCEET), 2012 International Conference on.21-22 March 2012. 852-859.

Malhotra, R. and Khanna, M. (2013). Investigation of relationship between object-oriented metrics and change proneness. *International Journal of Machine Learning and Cybernetics*. 4(4)**,** 273-286.

Martin, M. and Martin, R. C. (2006). *Agile principles, patterns, and practices in C#*. Pearson Education.

McIlroy, M. D., Buxton, J., Naur, P. and Randell, B. (1968). Mass-produced software components. Proceedings of the 1st International Conference on Software Engineering, Garmisch Pattenkirchen, Germany. sn, 88-98.

Meena, J. and O'Brien, L. (2011). A comparison of software reuse in software development communities. Software Engineering (MySEC), 2011 5th Malaysian Conference in.13-14 Dec. 2011. 313-318.

Mili, H., Mili, F. and Mili, A. (1995). Reusing software: issues and research directions. *Software Engineering, IEEE Transactions on*. 21(6)**,** 528-562.

Mordal, K., Anquetil, N., Laval, J., Serebrenik, A., Vasilescu, B. and Ducasse, S. (2013). Software quality metrics aggregation in industry. *Journal of Software: Evolution and Process*. 25(10)**,** 1117-1135.

Morisio, M., Ezran, M. and Tully, C. (2002). Success and failure factors in software reuse. *Software Engineering, IEEE Transactions on*. 28(4)**,** 340-357.

Mubarak, A., Counsell, S. and Hierons, R. M. (2009). Does an 80:20 rule apply to Java coupling? *Proceedings of the 13th international conference on Evaluation and Assessment in Software Engineering.* UK: British Computer Society.

Nazareth, D. L. and Rothenberger, M. A. (2004). Assessing the cost-effectiveness of software reuse: A model for planned reuse. *Journal of Systems and Software*. 73(2)**,** 245-255.

Postmus, D. and Meijler, T. D. (2008). Aligning the economic modeling of software reuse with reuse practices. *Information and Software Technology*. 50(7–8)**,** 753-762.

Poulin, J. S. (1994). Measuring software reusability. Software Reuse: Advances in Software Reusability, 1994. Proceedings., Third International Conference on.1-4 Nov 1994. 126-138.

Prieto-Diaz, R. (1993). Status report: software reusability. *Software, IEEE*. 10(3)**,** 61-66.

Rothenberger, M. A., Dooley, K. J., Kulkarni, U. R. and Nada, N. (2003). Strategies for software reuse: a principal component analysis of reuse practices. *Software Engineering, IEEE Transactions on*. 29(9)**,** 825-837.

Schmidt, D. C. (1999). Why software reuse has failed and how to make it work for you. *C++ Report*. 11(1)**,** 1999.

Sherif, K. and Vinze, A. (2003). Barriers to adoption of software reuse: A qualitative study. *Information & Management*. 41(2)**,** 159-175.

Shiva, S. G. and Shala, L. A. (2007). Software Reuse: Research and Practice. Information Technology, 2007. ITNG '07. Fourth International Conference on.2-4 April 2007. 603-609.

Sojer, M. (2011). Open source software developers' perspectives on code reuse. *Reusing Open Source Code*. (pp. 20-130). Gabler.

Sojer, M. and Henkel, J. (2011). Code Reuse in Open Source Software Development: Quantitative Evidence, Drivers, and Impediments. Rochester: Social Science Research Network.

Sommerville, I. (2010). *Software Engineering*. 10th. edition. Boston, MA: USA: Addison-Wesley.

Sommerville, I. and Kotonya, G. (1998). *Requirements Engineering: Processes and Techniques*. John Wiley & Sons, Inc.

Spoelstra, W., Iacob, M. and Sinderen, M. v. (2011). Software reuse in agile development organizations: a conceptual management tool. *Proceedings of the 2011 ACM Symposium on Applied Computing.* TaiChung, Taiwan: ACM.

Stephens, M. and Rosenberg, D. (2007). Use Case Driven Object Modeling with UML: Theory and Practice. Apress.

Stierna, E. J. and Rowe, N. C. (2003). Applying information-retrieval methods to software reuse: a case study. *Information Processing & Management*. 39(1)**,** 67-74.

Suresh, Y., Pati, J. and Rath, S. K. (2012). Effectiveness of Software Metrics for Object-oriented System. *Procedia Technology*. 6(0)**,** 420-427.

Thapar, S. S., Singh, P. and Rani, S. (2014). Reusability-based quality framework for software components. *SIGSOFT Softw. Eng. Notes*. 39(2)**,** 1-5.

Tripathy, P. and Naik, K. (2014). Reuse and Domain Engineering. *Software Evolution and Maintenance*. (pp. 325-357). John Wiley & Sons, Inc.

Utts, J. (2014). *Seeing through statistics*. Cengage Learning.

Vasa, R. (2010). *Growth and Change Dynamics in Open Source Software.* Ph.D Thesis, Swinburne University of Technology.

Wu, J. (2006). *Open Source Software Evolution and Its Dynamics.* Ph.D Thesis, University of Waterloo.

Zhao, Y., Yang, Y., Lu, H., Zhou, Y., Song, Q. and Xu, B. (2015). An empirical analysis of package-modularization metrics: Implications for software fault-proneness. *Information and Software Technology*. 57(0)**,** 186-203.

Zimmermann, T., Premraj, R. and Zeller, A. (2007). Predicting Defects for Eclipse. Predictor Models in Software Engineering, 2007. PROMISE'07: ICSE Workshops 2007. International Workshop on.20-26 May 2007. 9-9.