A METRICS-BASED FRAMEWORK FOR MEASURING THE REUSABILITY OF OBJECT-ORIENTED SOFTWARE COMPONENTS

SAMMY OLIVE NYASENTE

MASTER OF SCIENCE

(Computer Systems)

JOMO KENYATTA UNIVERSITY OF AGRICULTURE AND TECHNOLOGY

2015

A Metrics-based Framework for Measuring the Reusability of Object-oriented Software Components

Sammy Olive Nyasente

A thesis submitted in partial fulfillment for the degree of Master of Science in Computer Systems in the Jomo Kenyatta University of Agriculture and Technology

2015

DECLARATION

This thesis is my original work and has not been presented for a degree in any other university.

Signature _____   Date   _____

   Sammy Olive Nyasente

This thesis has been submitted for examination with my approval as a university supervisor.

Supervisors

Signature _____   Date   _____

   Prof. Waweru Mwangi
   JKUAT, Kenya

Signature _____   Date   _____

   Dr. Stephen Kimani
   JKUAT, Kenya

DEDICATION

To my family—with love and gratitude.

ACKNOWLEDGEMENTS

Table of Contents

LIST OF TABLES

LIST OF FIGURES

LIST OF APPENDICES

ABSTRACT

Software reuse is a popular way of addressing software development issues—such as high cost of development, low productivity, and poor quality. However, organizations are yet to realize maximum payoff from reuse due to reusability related issues. Reusability can only be improved through measurement, because measurement is the only sure way of monitoring and improving software quality. The goal of this study was to present a reliable metrics-based framework for measuring the reusability of object-oriented (OO) software components. A survey involving OO developers was conducted, where methods that they currently use in assessing reusability were examined. The methods in question include checking of source code, reading documentation, intuition, and checking of comments. These methods were found to be largely subjective, hence not reliable. In addition, four reusability assessment frameworks—found in literature were examined, and found to have various challenges: all of the frameworks lack predictive power, hence they cannot be applied at early stages of software development; three of the frameworks include traditional metrics, which cannot be used to measure OO software components; and lastly, one of the frameworks was platform dependent. The conclusions about the inefficiencies of the reviewed reusability assessment methods and frameworks were arrived at, following a comprehensive literature analysis, through which a criterion for evaluating the efficiency of reusability assessment frameworks for OO software was defined. Literature analysis revealed that an effective reusability assessment framework for OO software should include the following key elements: major reusability attributes, factors that influence the reusability attributes (reusability factors), measurable OO design constructs that influence the reusability factors, and OO metrics for measuring the OO constructs. This research culminated into the development and implementation of a framework that conforms to this criterion. The developed framework was validated for superiority over the existing reusability assessment methods and frameworks through a comparative analysis.

CHAPTER ONE
INTRODUCTION

1.1. Background Information

Software is increasingly becoming an important factor in the advancement of the modern economy. It is not only a basic infrastructure to economic advancement, but also delivers information, which is the most important product of our time (Pressman, 2010). According to Budhija, Singh, and Ahuja (2013) software has become critical to advancement in almost all areas of human endeavors. In fact, it is impossible for the modern world to run without software (Sommerville, 2011).

The criticality of the role played by software in socioeconomic advancement has seen a rapid growth in demand for software on one hand, with software developers being unable to meet this demand on the other. According to Sommerville (2011), this is due to the increasing demand for large and more complex systems that need to be delivered more quickly, and the failure of software development companies to use software engineering methods in their everyday work.

According to Budhija et al. (2013), the art of programming alone is not sufficient to construct complex software that are of good quality, maintainable, and that are delivered on time and within budget. One of the ways often pursued to achieve this purpose is software reuse (Frakes & Kang, 2005). Babu and Srivatsa (2009) define reuse as the process of creating software systems from existing software assets rather than building them from scratch.

In Object technology, it is possible for developers to build much of the software by combining existing classes; therefore, each time a new class is created it has the potential of becoming a reusable software asset (Deitel P. J. & Deitel H. M., 2006).

According to Deitel P. and Deitel H., (2011), reusable classes are crucial to the software revolution that has been spurred by object technology—just as the notion of interchangeable parts was crucial to the industrial revolution.

Even though the software industry has developed massively in last decades, component reuse is still facing numerous issues, and lacking adoption from software developers (Hristov, Hummel, Huq, & Janjic, 2012). Hristov et al. point out the difficulty of determining which artifacts are best suited to solve a particular problem in a certain context, and the ease with which they can be reused—as one of the impediments that prevent efficient and effective reuse. The authors claim that, this is due to lack of a comprehensive framework describing reusability of software and structuring appropriate metrics in a way that is easy to use. Such a framework—according to them is crucial in facilitating the adoption of reuse in software development.

According to Frakes and Kang (2005), reusability is a quality factor that indicates the probability of reuse of any software artifact. Reusability of components is important, and should be measured, in order to realize effective reuse (Washizaki, Yamamoto, & Fukazawa, 2003). In object oriented software development (OOSD), reusability metrics can be used to predict the extent to which classes can be reused. That is, reusability metrics try to find out the ease with which classes can be reused (Gill & Sikka, 2011).

The term metric is defined by the IEEE *standard glossary of software engineering terminology* (IEEE, 1990) as a quantitative measure of the degree to which a system, component, or process possesses a given attribute. According to Chawla and Nath (2013), metrics are helpful in evaluating the status of an attribute of software, and finding opportunities of improvement.

There have been tremendous efforts towards achieving effective reuse in OOSD in recent decades, and several metrics that can be used in measuring the reusability of object oriented (OO) software have been presented in various literatures(e.g. (Chidamber & Kemerer, 1991, 1994; Cho, M. S. Kim, & S. D. Kim, 2001; Gill & Sikka, 2011)). According to Cho et al. (2001), OO metrics measure principal structures that if improperly designed, negatively affects design and code quality attributes. This implies that, to measure reusability of OO components, there is need for a good understanding of the attributes that influence reusability, how OO principal structures are related with the reusability attributes—as well as OO metrics that can be used to measure the OO structures.

According to Hristov et al. (2012),the major challenge in reusability measurement is, determining the attributes that should be used to assess reusability because a common agreement is yet to be reached in the research community as to which software characteristics provide a sufficient basis for determining software reusability, and which metrics should be used to measure these characteristics. This challenge is compounded by a general disagreement within the research community regarding software measurement (Pressman, 2010).Pressman observes that, there is no agreement as to which attributes should be used in assessing software quality. Therefore, the software community still has a long way to go in achieving effective reuse through OO technology, since there is no effective framework for describing and measuring reusability of classes.

1.2. Statement of the Problem

The demand for new software is currently increasing at an exponential rate, as well as the cost and effort to develop them (Sandhu, Kaur, & Singh, 2009).This has led to a large backlog of software that need to be written—a situation described as the software

crisis (Sommerville, 2011). Software professionals have over the years recognized reuse as a powerful means of potentially overcoming this crisis (Frakes & Kang, 2005).

To achieve the objectives of reuse, there is need to focus on the concept of reusability in a disciplined manner (Budhija et al., 2013). Thus, in addition to adopting OO design and development—which is a popular method for improving software reusability, productivity and flexibility (Dubey & Rana, 2010), there is also need to ensure that the classes being developed or reused have the required degree of reusability(Nyasente, Mwangi, & Kimani, 2014a). Research has shown that this objective can be achieved by use of OO software metrics (Chawla & Nath, 2013).

Measurement has been an active area of reuse research for more than two decades (Frakes & Kang, 2005), and a number of metrics and frameworks for reusability assessment have been presented in literature. However, the software community is yet to agree on characteristics that should be used in assessing reusability, and which metrics are sufficient in measuring these characteristics (Hristov et al., 2012). Therefore, there is lack of clear and efficient methods of measuring reusability—posing a major setback in achieving successful reuse. To achieve the maximum benefits from reuse in OO software development, a clear framework that describes reusability and structures OO metrics in a way that is easy to use is required. Such a framework could not be found in literature.

1.3.   Justification of the Study

Although several OO metrics—such as the "CK" metrics (Chidamber & Kemerer, 1991, 1994) exist in literature; a framework that describes, and that can be used to measure the reusability of OO components (i.e. a framework that relates major OO design constructs, with key reusability attributes and groups metrics for quantifying reusability on one system), could not be found. This research work presents a novel metrics-based

framework that fulfills this purpose. The framework is critical in facilitating reusability measurement, since OO metrics—according to Dubey and Rana (2010) require a thorough understanding of OO concepts. In addition, no single metric can be used to measure all characteristics of OO software.

This research does not only culminates to an easy to use framework that developers can use to measure the reusability of OO components, but also seeks to foster the understanding of software developers on what OO reusability is; how it can be measured, and how it can be improved. This research will also form a basis for further research in the area of OO reusability assessment. This will go a long way in accelerating successful reuse of OO component—with an objective of resolving the software crisis in the end.

1.4. Research Questions
1. What are the methodologies employed by OO developers in assessing the reusability of components?
2. What are the shortcomings of the reusability assessment strategies that are in place?
3. What are the attributes that influence the reusability of OO components?
4. How can we use the reusability attributes to objectively determine the reusability of OO components?

1.5. Objectives of the Study
1.5.1. Broad Objective
The broad objective of the study is to establish an easy to use metrics-based framework for measuring the reusability of OO components. The framework will contain key reusability attributes that will be measured using software metrics that exist in literature.

1.5.2. Specific Objectives

1. Identify and examine the strategies and methods used by OO developers in assessing the reusability of classes.
2. Identify the shortcomings of the reusability assessment strategies that are currently used.
3. Determine the major attributes that influence reusability, hence design and implement a framework based on metrics—that can be used to measure the reusability of OO components.
4. Test the working of the framework and validate its superiority over the existing methods of measuring reusability.

1.6. Scope of the Study

According to Hristov et al. (2012), there are several factors, which influence component reuse and reusability is just one of them. This research only focuses on reusability measurement as a way of improving the reusability of OO components—which will improve reuse in return. The research does not come up with new reusability metrics: it presents a framework that structures OO metrics that exist in literature, in a way that is easy to use. A survey is also conducted to find out methods that OO developers use in measuring reusability; shortcomings of these methods, as well as challenges encountered in measuring reusability. This information is crucial in establishing the reusability assessment framework, and provides empirical evidence on the theoretical literature that this research is based on.

1.7. Thesis Outline

This thesis is organized into six chapters. Chapters one presents the background information for the study, the research problem, as well as the significance of the study. In addition, the research questions, objectives, and the scope of the study are presented. Chapter two presents a detailed literature review—where both theoretical and empirical

literature is reviewed and some unresolved issues (research gaps) with regards to OO reusability assessment are highlighted. Finally, a concept that guides the study is presented. Chapter three analyzes the research methodology used in this study. It describes the methodologies used for data collection and analysis, framework design, framework validation, and framework implementation (system design).Chapter four on the other hand, discusses the findings of data analysis. Chapter five describes the framework developed in this thesis, and ends with its implementation (system development). Finally, chapter six gives conclusions and recommendations.

CHAPTER TWO
LITERATURE REVIEW

2.1. Introduction

Measurement is a key element in any engineering process, as it enables engineers to better understand attributes of models that they create, and to assess quality of engineered products (Pressman, 2010). Measurement is required in software engineering to assess quality and improvements in performance of software products, in order to meet the ever-increasing demands of users (Chawla & Nath, 2013). Pressman (2010) underscores the importance of measuring the process of software engineering and software products by stating that; "if you do not measure, there is no real way of determining whether you are improving. And if you are not improving, you are lost".

According to Sommerville (2011), software measurement is concerned with deriving a numeric value or profile for an attribute of a software component, system, or process. The quality of software products, effectiveness of processes, tools, and methods, can be assessed by comparing the derived values for the attributes (Sommerville, 2011). Measurement of software products as well as the process of software production is achieved through metrics (Rawat, Mittal, & Dubey, 2012; Sharma &Dubey**,** 2012)

Nirpal and Kale (2011) describe software metrics, as measurement based techniques that are applied to processes, products and services, to supply engineering and management information—which they can work on to improve processes, products and services, if required. This description is depicted in figure 2.1.
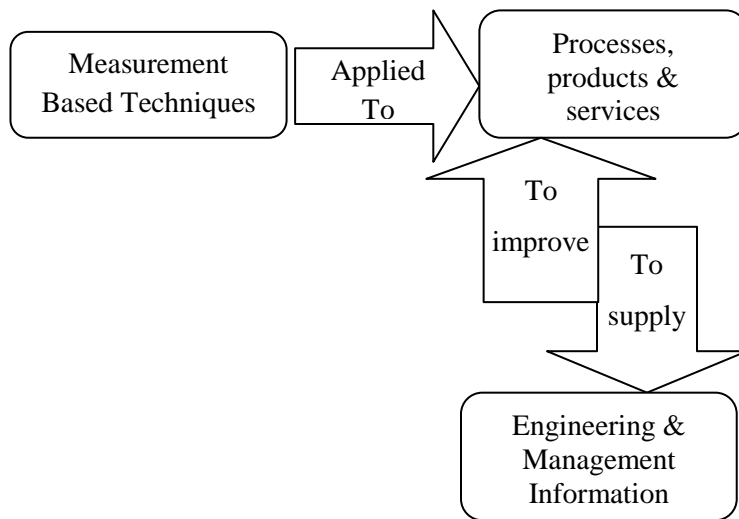
Figure 2.1: Pictorial description of software metrics (Nirpal & Kale, 2011)

According to Chawla and Nath (2013), metrics help software engineers and developers to find opportunities of improving software products and software production processes, by providing information regarding the status of certain attributes of the software product or process. In other words, metrics can be used to assess quality of software (Sandhu et al., 2009).

2.2.   Classification of Software Metrics

Software metrics can be classified into different categories; however, they are often categorized in a much broader sense as: (i) process metrics, and (ii) product metrics (Farooq, Quadri, & Ahmad, 2011). According to Pressman (2010), process metrics are used to measure the efficacy of software development processes. They provide a set of process indicators by measuring specific attributes of the software development process.

Product metrics on the other hand are used to assess the quality of software products during development (Pressman, 2010). Product metrics are also known as quality metrics and they measure properties of the software (Singh G., Singh D., & Singh V., 2011). Product metrics provide indicators of the efficacy of the requirements, design and

code models; the effectiveness of test cases; and the overall quality of software to be build. These indicators provide insights that enable software developers to adjust the product to make things better (Pressman, 2010).

2.3.   Measuring Software Size and Size Metrics

Software size is one of the most elementary attributes which can be used to estimate the complexity of a software system, because complexity is basically the quality of "interconnectedness" of parts, and size can be thought as the sheer numbers of basic "parts" (Laird & Brennan, 2006).According to Hristov et al. (2012) excessive complexity limits the chances of a software component of being reused. Therefore, it should be managed through measurement (Nyasente et al., 2014b).

2.3.1. The LOC Metric

The number of lines of code (LOC) is the simplest measure for software size, which measures the physical length of software (Laird & Brennan, 2006). The LOC measure can be used to normalize metrics and compare different projects; however, its reliability is dependent on rules that apply when counting source lines of code. Different organizations and studies use different rules, with the most popular one being NKLOC (non-commented thousands lines of code) and LLOC (logical lines of code) (Laird & Brennan, 2006).

The LOC metric can be easily counted for any software, and other simple size-oriented metrics such as errors per thousand lines of code (Errors per KLOC), defects per thousand lines of code (defects per KLOC) , cost per thousand lines of code ($ per KLOC), etc; can be derived from this metric. Other important measures that can be computed from LOC include; Errors per person-month, KLOC per person-month, and, $ per page of documentation (Pressman, 2010). However, the LOC metric is programming

language dependent and its usefulness fizzles out for non-procedural languages (Laird & Brennan, 2006; Pressman, 2010).

2.3.2. Function Points (FP) Metrics

Physical software size measurement such LOC was largely relevant until early 1990s, since most code was text. However, with the advent of visual languages such as visual basic, the size of text became irrelevant, as "selections" and "clicks" become the "code". For these types of languages and for generated languages, functional size measurements are typically used (Laird & Brennan, 2006).

Function point is an approach for sizing a system, based on the functions delivered by the system rather than how it does it internally (Laird & Brennan, 2006). The FP measure was presented by Albrecht (1983) and can be determined early in the software development life cycle (as cited in Sharma & Dubey, 2012). It measures project size by functionality indicated in the customer's or tender requirement specification (Galin, 2004). Unlike the LOC measure, FP is independent of the programming language used to develop software (Sommerville, 2011).

The number of function points in a system is computed by measuring or estimating external inputs and outputs, user interactions, external interfaces, and files or database tables used by the system (Sommerville, 2011). The FP of a system is the weighted total of the number of external inputs (e.g., transaction types); number of external outputs (e.g., report types); number of logical internal files (files as the user might conceive them, not physical files); number of external interface files (files accessed by the application but not maintained by it); and number of external inquiries (types of online inquiries supported) (Kan, 2002).

2.4. Measuring Software Complexity and Complexity Metrics

Any software system or module has some inherent complexity, based on the problem it needs to solve. However, unnecessary complexity introduces a number of problems—such as additional defects and lower productivity. The hypothesis in this case is that, the more complex the software, the more difficult it is to understand it—hence the more difficult it will be to debug and maintain it (Laird & Brennan, 2006). According to Ghezzi, Jazayeri, and Mandrioli (2003) and Nyasente et al. (2014a), reusable components should be easy to understand, debug and maintain. This means that, complex modules are hard to reuse as compared to simple ones.

The objective of measuring complexity is to identify factors that cause software complexity, so that it can be managed (Laird & Brennan, 2006). According to Laird and Brennan, complexity metrics can be used in identifying designs and code that should be considered for simplification, or modules that should be subjected to additional testing.

According to Laird and Brennan (2006), system complexity can be viewed and measured from three different aspects: structural, conceptual and computational. Structural complexity concerns the design and structure of the software itself. This type of complexity can be determined by structural complexity metrics such as LOC, function point, cyclomatic complexity etc. Conceptual complexity on the other hand refers to the difficulty in understanding a software system. There are no specific metrics that are known to measure this type of complexity. One possible explanation to this is that, conceptual complexity is more of psychological, and is dependent on the mental capacity of the programmer—making it difficult to quantify.

Lastly, computational complexity refers to the complexity of the computation being performed by a system. Computational complexity is measured by determining the amount of time and space required by the system for calculations. Computational

12

complexity is useful in evaluating and comparing implementations and designs for efficiency, and in ensuring that the complexity of the solution does not exceed the inherent complexity of the problem being solved.

2.4.1. McCabe's Cyclomatic Complexity (CC) Metrics

McCabe's Cyclomatic Complexity is the most famous complexity metric, which is a measure of the number of control flows within a module (Laird & Brennan, 2006). The metric's original goal was to measure the testability and understandability of the software module (Sharma & Dubey, 2012; Laird & Brennan, 2006). Cyclomatic complexity is based on graph theory, and is calculated according to the program characteristics as captured by its program flow graph (Galin, 2004). Cyclomatic complexity denoted by *V(G)* can be computed using any of the three equations (Galin, 2004):

$$V(G) = R \tag{2.1}$$

$$V(G) = e - n + 2 \tag{2.2}$$

$$V(G) = P + 1 \tag{2.3}$$

Where:

$R \equiv$ is the number of regions in the program flow graph (i.e. any enclosed area in the program flow graph. In addition, the area around the graph not enclosed by it is counted as one additional region.)

$N \equiv$ is the number of nodes in the program flow graph.

$P \equiv$ is the number of decisions contained in the graph, represented by nodes having more than one leaving edge.

High Cyclomatic complexity of a program module indicates higher complexity (Laird & Brennan, 2006), making the module difficult to reuse (Nyasente et al., 2014c).

2.4.2. Halstead's Software Science

Halstead (1977) proposed the first "laws" for computer software (as cited in Pressman, 2010). He developed metrics based on the number of distinct operands (n_1), and the number of distinct operators (n_2) in a computer program (Laird & Brennan, 2006). Halstead considers a program to be constructed by manipulating unique operators (n_1) and unique operands (n_2); i.e. a computer program composed of N_1 operators and N_2 operands is constructed by selecting n_1 distinct operators and n_2 distinct operands (Singh et al., 2011). Based on this model, Halstead developed equations for, program length, volume, program level (which is a measure of program complexity), language level, and other features such as development effort, development time, and projected faults in a software (Pressman, 2010).Halstead's Metrics are defined as (Laird & Brennan, 2006):

Length:	$N = N_1 + N_2$	(2.4)
Vocabulary:	$n = n_1 + n_2$	(2.5)
Volume:	$V = N(log_2(n))$	(2.6)
Difficulty:	$D = (n_1/2) * (N_2/n_2)$	(2.7)
Effort:	$E = D * V$	(2.8)

Where:

n_1= number of distinct operators

n_2= number of distinct operands

N_1= total number of operators

N_2= total number of operands

Although Halstead metrics do not tend to be practical in usage, they were extremely useful in setting up the stage for discussions on code structure metrics. One of the limitations of, Halstead metrics is that, they have no predictive power for development effort, since they are calculated after code is written. Although Halstead metrics can be

used as a predictor of maintenance effort, they have not been shown to be better than LOC, which is simpler (Laird & Brennan, 2006).

2.4.3. Information Flow Metrics (Henry and Kafura's Metrics)

In 1981, Henry and Kafura proposed information flow metrics, which are sometimes referred to as Henry and Kafura's metrics (as cited in Singh et al., 2011). Information flow metrics can be used to determine the complexity of a system by measuring the flow of information among system modules (Laird & Brennan, 2006; Sharma & Dubey, 2012; Singh et al., 2011). The underlying principle behind this is that, high information flow among system modules indicates lack of cohesion (i.e. a low degree of relationship between methods of a module), which causes higher complexity (Laird & Brennan, 2006).

Information flow metrics use some combination of the number of local flows into a module (fan-in), the number of local flows out of a module (fan-out), and the length to compute a complexity number for a procedure. Fan-in, fan-out, and length are defined in a number of ways by different variations of the metric (Laird & Brennan, 2006).Initially Henry and Kafura (1981) defined the Information Flow Complexity (IFC) of a module as (as cited in Laird & Brennan, 2006):

$$IFC = Length * (fanin * fanout)^2 \qquad\qquad (2.9)$$

Where:

Fanin≡ is the number of local flows into a module plus the number of data structures that are used as input.

Fanout ≡ is the number of local flows out of a module plus the number of data structures that are used as output.

Length≡ is the length of a procedure in LOC.

Henry and Kafura's metrics later evolved into the IEEE Standard 982.2, and they are defined as follows (Laird & Brennan, 2006):

$$IFC = (fanin * fanout)^2 \qquad (2.10)$$

$$Weighted\ IFC = Length * (fanin * fanout)^2 \quad (2.11)$$

Where:

Fanin $\equiv$ local flows into a procedure plus number of data structures from which the procedure retrieves data

Fanout$\equiv$ local flows from a procedure plus number of data structures that the procedure updates

Length$\equiv$ number of source statements in a procedure (excluding comments in a procedure)

2.5.   Measuring Quality of Object-Oriented (OO) Software

The quality of OO software is dependent on various quality concepts like complexity, usability, reliability, testability, understandability etc. These concepts are closely related to OO features such as coupling, inheritance, and cohesion, which can be measured using various OO metrics (Dubey & Rana, 2010). Therefore, quality of OO software can be determined by measuring OO design features—using OO metrics. Examples of metrics that can be used in assessing different OO software quality aspects include, the Chidamber and Kemerer's metrics—commonly known as the CK metrics suite (Chidamber & Kemerer, 1991, 1994), and, the Gill and Sikka Metrics (Gill & Sikka, 2011).

2.6.   Software Reuse and Reusability

Software reuse is the use of existing software components to create new software, whilst reusability is the degree to which a given component can be reused (Gill & Sikka, 2011). This means that reusability is the property that determines a component's reuse.

Therefore, if a component's reusability is low, then its potential for reuse becomes low as well (Nyasente et al., 2014a).

According to Budhija et al.(2013), the maximum benefits of software reuse can only be attained if we focus on the concept of reusability in a disciplined manner. A similar view is expressed by Washizaki et al. (2003) when they state that, among the several quality characteristics, reusability is particularly important when reusing components, and it should be measured in order to reuse them effectively.

2.7.   Measuring Software Reusability

Issues related to software development such as; quality, productivity, cost of development etc, can be addressed by focusing and improving component reusability (AL-Badareen, Selamat, Jabar, Din, & Turaev, 2010; Ilyas & Abbas, 2013; Mishra, Kushwaha, & Misra, 2009). According to Pressman (2010), there is only one sure way of improving software quality, and that is through measurement. Therefore, developers must measure reusability of components if they need to improve it.

Component reusability is determined by certain attributes that can be measured using metrics, and the task involved in reusability measurement is to relate reusability attributes with appropriate metrics, and find out how these metrics collectively determine the reusability of components (Sandhu et al., 2009). In other words, a model that relates reusability attributes with reusability factors that can be measured using metrics is required in order to assess the reusability of components.

2.8.   Related Work in Reusability Measurement

Software reusability has been an active area of research in Software Engineering for more than two decades, and a number of frameworks for quantifying reusability have

been presented. Some of the research works that were identified in literature are reviewed in this section.

2.8.1. The Basic Reusability Attributes Model

Caldiera and Basili (1991) present a basic reusability attributes model (shown in figure 2.2) for identifying and qualifying reusable software components. The model attempts to characterize reusability attributes directly through measures of an attribute, or indirectly through measures of evidence of an attribute's existence. The model consists of three attributes that are believed to influence the reusability of components, namely; reuse costs, functional usefulness, and quality of components. These attributes are determined by factors, which are directly or indirectly measured by McCabe's Cyclomatic Complexity metrics, Halstead's Volume metrics, Regularity, and Reuse Frequency.
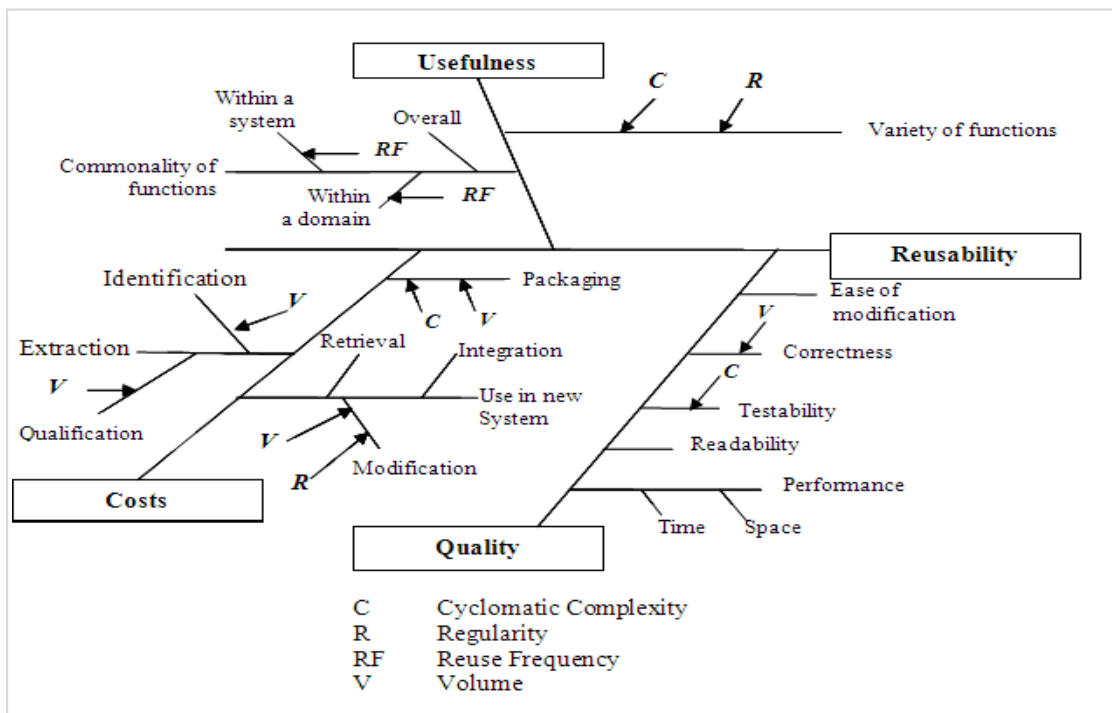
Figure 2.2: Basic reusability attributes model (Caldiera & Basili, 1991)

2.8.1.1. Strengths and Limitations of the Basic Reusability Attributes Model

Although the basic model is elementary, it captures important characteristics affecting software component reusability (Caldiera & Basili, 1991). The authors outline a criterion for objectively determining the reusability of components, using the reusability characteristics. The major strength of the model lies in the fact that, it can be used to objectively assess component reusability—owing to the fact that the authors propose objective metrics for measuring the reusability characteristics.

Notwithstanding its strengths, the basic reusability model only focuses on developing a catalog of reusable components from already existing components. That is, it only addresses the problem of how to analyze existing components and identifying the ones that are suitable for reuse (Caldiera & Basili, 1991). The framework lacks predictive power, and therefore, it cannot be useful in predicting reusability of components when they are being developed.

2.8.2. Black-box Component Reusability Model

Washizaki et al. (2003) present a framework for measuring the reusability of OO black-box components. The authors consider understandability, adaptability, and portability—as attributes that determine reusability. They also proposed a suite of five metrics for measuring reusability: existence of meta-Information (EMI), rate of component observability (RCO), rate of component customizability (RCC), self-completeness of component's return value (SCCr), and self-completeness of component's parameter (SCCp). These metrics target the JavaBeans architecture, and are defined according to the model shown in figure 2.3.
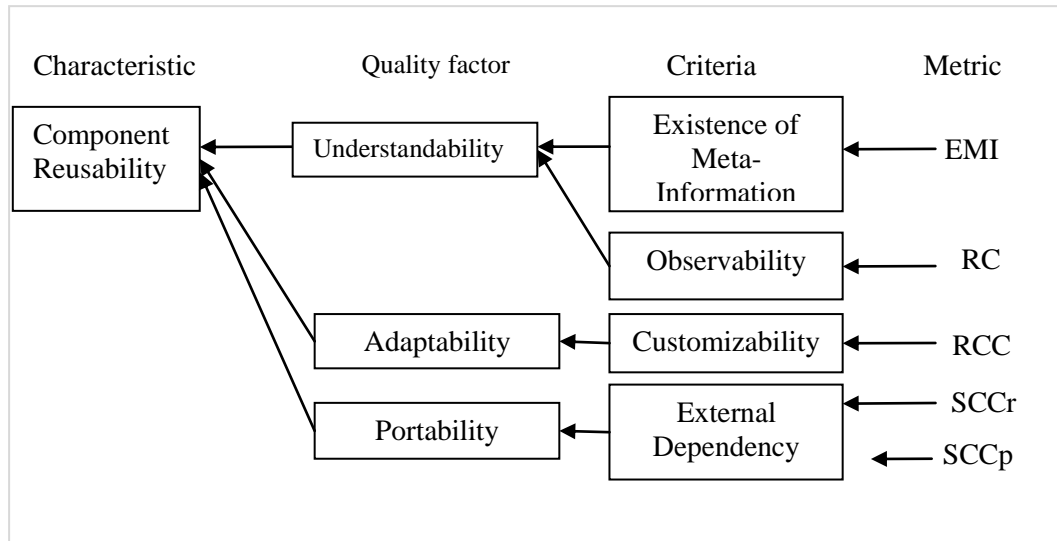
Figure 2.3: Black-box component reusability model (Washizaki et al., 2003)

2.8.2.1. Strengths and Limitations of the Black-Box Component Reusability Model

The major strength of the black-box component reusability framework is its ability of measuring reusability of components based on limited information that can be obtained from the outside of the component without any source code. This makes the framework suitable for assessing reusability when the source code of components cannot be obtained (Washizaki et al., 2003). Another strength of the framework is that, it includes a suite of objective metrics that were defined based on an empirical study—with confidence intervals that were set by statistical analysis of a number of JavaBeans components (Washizaki et al., 2003). Therefore, the measures of reusability obtained from using the framework can be relied on.

On the other hand, the framework has some limitations. The most visible limitation is that, it includes metrics that are useful in measuring the reusability of black-box components for the activity of development with reuse, and only targets the JavaBeans architecture (Washizaki et al., 2003). That is, the framework is architecture dependent

and does not address the problem of how to measure reusability of white-box components, and how to predict reusability when components are being developed.

2.8.3. The Reusability Framework for Ad-Hoc Reuse

Hristov et al. (2012) present a reusability assessment framework (shown in figure 2.4) for ad-hoc software reuse. Their framework structures existing reusability metrics for component-based software development. They proposed eight attributes that should be considered in assessing the reusability of components in ad-hoc reuse scenarios. These attributes include availability, documentation, complexity, quality, maintainability, adaptability, reuse, and price. These attributes are determined by various factors, which can be directly or indirectly measured using various metrics that the authors propose.
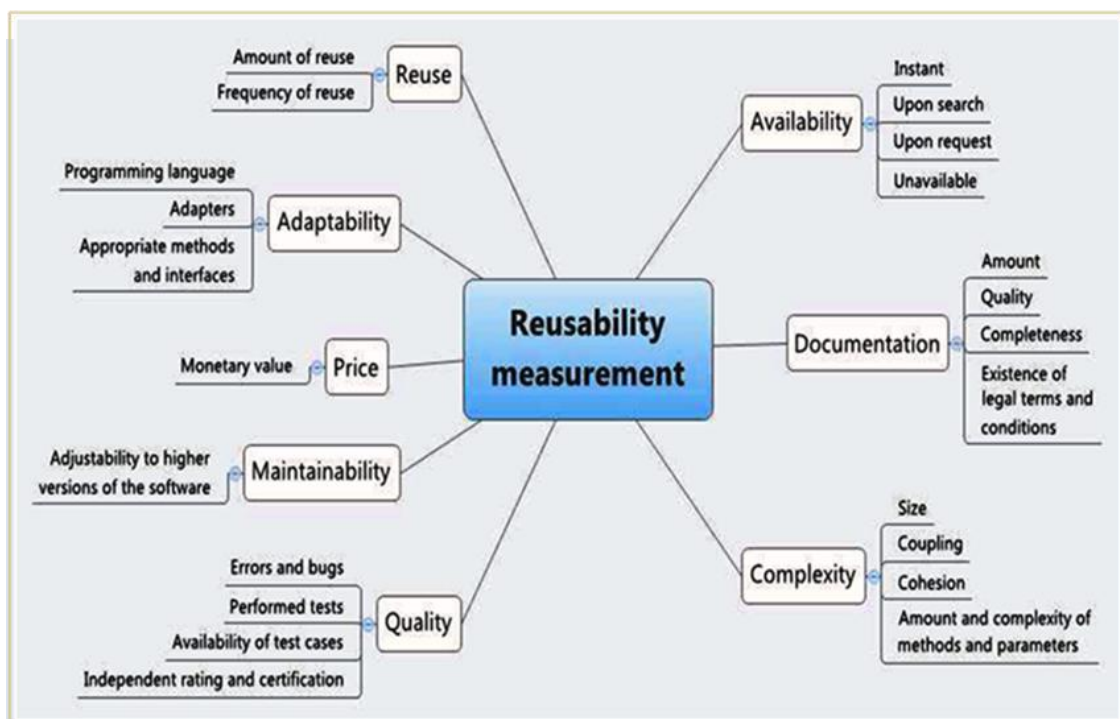
Figure2.4: Reusability model for ad-hoc reuse (Hristov et al., 2012)

Hristov et al. (2012) further defined a reusability calculation model shown in equation 2.12, where metrics values resulting from measuring every part of the reusability model are aggregated, in order to get a component's reusability (R_{cc}).

$$R_{cc}=w_1.avail + w_2.Doc + w_3.Compl + w_4.Qual +$$
$$w_5.Maint + w_6.Price + w_7.Adapt + w_8.Reuse \qquad (2.12)$$

Where:

w_1 - w_8 are weights and the rest are composite metrics for the reusability attributes (shown in figure 2.4, above).

2.8.3.1. Strengths and Limitations of the Reusability Framework for Ad-hoc Reuse

The Major strength of the reusability framework for ad-hoc reuse is in its comprehensiveness and ease of use (Hristov et al., 2012). The framework clearly relates major reusability attributes with measurable factors and includes objective metrics for measuring these factors. The authors also define a reusability equation that outputs a single reusability value, which is easy to interpret.

However, the framework by Hristov et al. (2012) is suitable for assessing the reusability of software components in ad-hoc reuse scenarios only. Although, the framework gives insights into reusability assessment in planned reuse scenarios, it cannot be relied on in assessing reusability in such reuse scenarios. This is because some attributes (e.g. reuse and price) can only be assessed if the component has already been reused before.

2.8.4. The Reusable Software Components Framework

AL-Badareen et al. (2010) present a reusable software component framework for systematic reuse (see figure 2.5).They categorize reusability characteristics into two main categories: (i) characteristics to assess components before they are stored in the reuse library, and (ii) characteristics to assess components in order to build a new system. The first category considers the general characteristics that are required by any

system, and they include; software coexistence, adaptability/interoperability, generality and compliance.
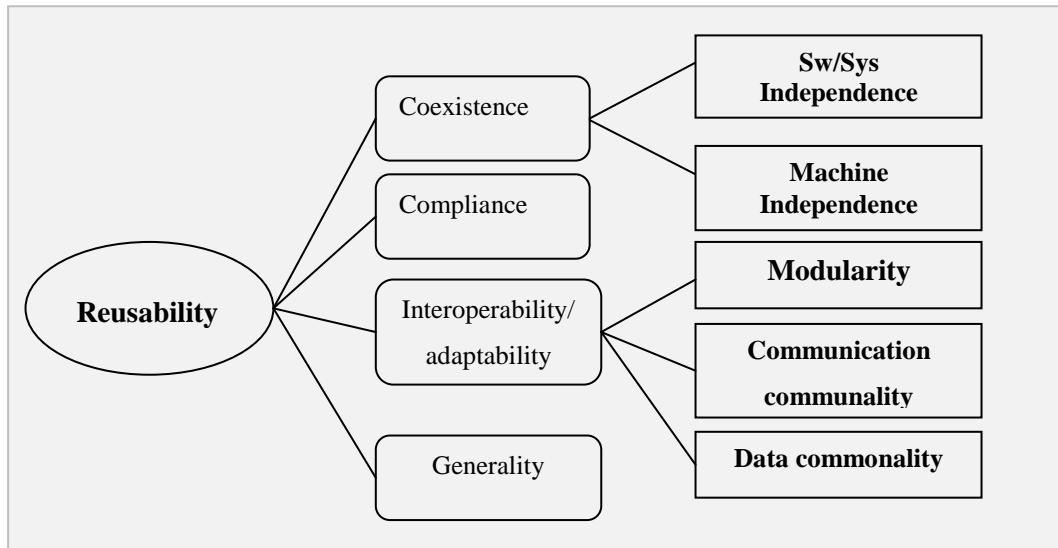
Figure2.5: Reusable software components model (AL-Badareen et al., 2010)

AL-Badareen et al. (2010) define software coexistence as the ability of the system or the sub-system to work in different environments. Software coexistence includes software system independence and hardware independence. The former represents the degree to which the program is independent of nonstandard programming language features, operating system characteristics, and other environment constraints. Whilst the latter is the degree, to which the software is de-coupled (independent) from its operating hardware.

On the other hand, adaptability/interoperability of the software is the ability of the software to communicate with other systems. It includes modularity, communication communality, and data communality. System modularity represents the degree to which a system or computer program is composed of separate and independent components such that a change to one component has minimal impact on other components.

Communication commonality represents the degree to which standard interfaces, protocols and bandwidth are used. Data commonality is explicit use of standard data structures and types throughout the program (AL-Badareen et al., 2010).

According to AL-Badareen et al. (2010), software generality is the degree to which a system or computer program is composed of distinct components such that a change to one component has minimal impact on other components. This enables components of a system to be used in different contexts with ease. Compliance verifies whether the software follows any standard or international certificates in order to build reusable software.

The second category of characteristics defined by AL-Badareen et al. (2010) comprise of specific characteristics that help in the system development. These characteristics are component suitability, documentation, and modifiability. The suitability of the component is measured in order to see whether the component is able to perform the intended function properly. The documentation of the component simplifies the job of the software developer to understand it. While component modifiability is required to change the components as it is required in the new system.

2.8.4.1. Strengths and Limitations of the Reusable Software Components Framework

A noticeable strength of the framework by AL-Badareen et al. (2010) is its simplicity. The framework provides a structured criterion for evaluating reusable components when adopting them for reuse. The framework describes reusability assessment in a straightforward manner. The authors relate major reusability attributes with measurable factors that influence them, and then they outline a criterion for assessing the reusability characteristics.

Although the reusable software components framework by AL-Badareen et al. (2010) provides a basis for understanding reusability in systematic reuse, the authors do not give an objective way of measuring reusability attributes. That is, they do not specify metrics for measuring the reusability attributes. They instead propose that the characteristics be evaluated through tests. This is a limitation because, tests for evaluating some of the reusability characteristics—(e.g. coexistence), can only be conducted if the component is in existence. For example, system and hardware independence, are determined by running the software in different software and hardware environments respectively. This limits the suitability of the framework to the process of extraction, evaluation, processing, adoption, and storage of reusable software components (AL-Badareen et al., 2010).

2.8.5. Summary of the Reviewed Reusability Assessment Frameworks

Table 2.1 gives a summary of the reviewed reusability assessment frameworks. It outlines the suitability and limitations of the said frameworks.

Table 2.1: Summary of the reusability assessment frameworks

Framework	Source	Suitability	Limitations
Basic Reusability Attributes Model	Caldiera and Basili (1991)	-Suitable for assessing the reusability of existing components, so as to come up with a catalog of reusable components.	-It cannot predict reusability of components during development for reuse.
Black-box component reusability model	Washizaki et al. (2003)	-Suitable for measuring reusability of JavaBeans components when developing with reuse.	-It cannot measure reusability of white-box components -It only gives insights but it cannot assess reusability of non-JavaBeans components.

Reusability Framework for Ad-hoc Reuse	Hristov et al. (2012)	-Suitable for assessing the reusability of components in ad-hoc reuse scenarios.	-It only gives insights, but cannot predict reusability of components in planned reuse.
The Reusable Software Components Framework	AL-Badareen et al. (2010)	-Suitable for the process of extraction, evaluation, processing, adoption, and store reusable software components.	-It does not specify metrics to measure the reusability attributes. - The reusability attributes are evaluated by conducting tests on components, thus it cannot be used to predict reusability when developing for reuse.

2.9 . Unresolved Issues in OO Reusability Assessment

The existence of reusability assessment frameworks in literature reveals efforts by researchers in trying to improve software reusability. However, these research works have been conducted against a backdrop of uncertainty as to which attributes should be used to measure reusability (Hristov et al., 2012). In addition, OO metrics measure principal structures whose design affects quality attributes (Cho et al., 2001). Therefore, it can be concluded that, an effective OO reusability assessment framework should:

(i)   Clearly define the attributes that influence the reusability of components;

(ii) define factors that influence each of the reusability attributes (reusability factors);

(iii) relate the reusability factors with OO design structures that influence them;

(iv) relate various metrics with the OO structures that they measure; and,

(v)   establish how these metrics collectively determine reusability (Nyasente et al., 2014a).

A review of literature reveals that no framework of the nature described above exists. Therefore, in spite of the existence of reusability assessment frameworks, and various

OO metrics, the issue of effective reusability assessment in OO technology remains unresolved.

2.10. Proposed Solution

This research will attempt to resolve the issues—highlighted in section 2.9 above, by developing an easy to use metrics-based framework that can be used in measuring the reusability of OO components. To achieve this objective, the following will be undertaken:

i.   a survey involving OO developers will be conducted in order to have an in-depth understanding on reusability, as well as understand the manner in which reuse is conducted;

ii. literature will be analyzed in order to determine major reusability attributes, factors that influence these attributes, and measurable OO structures that influence these factors;

iii. a model that relates the reusability attributes, with reusability factors, and OO constructs will be developed;

iv. A reusability equation will be defined; and,

v.   a reusability calculation tool/system, which is be an automation of the reusability equation shall be developed.

2.11. Summary

This chapter has explained the concept of measurement, its importance to software engineering, and how it can be achieved through software metrics. The broad categories of metrics have been explored, and the usefulness of each is highlighted. Some traditional software metrics are also discussed and their relationship with reusability assessment is highlighted. This chapter has also explained what software reuse is, its benefits to software development, and the difference between reuse and reusability. The reason why reusability measurement is important—with regards to software reuse is also

explained; and some reusability assessment frameworks that exist in literature have been examined. The suitability of these frameworks, as well as their limitations has been brought to the fore. Lastly, the ideal features of an effective framework for assessing OO reusability—(as derived from literature) are highlighted. This brought forth a number of unresolved issues in OO reusability assessment. This informed the need for a survey involving OO developers—that aimed at establishing the status of reuse, the methods used in assessing reusability, and the efficiency of these methods. This information was crucial in providing a basis for developing the reusability assessment framework that this research culminates to.

CHAPTER THREE
RESEARCH METHODOLOGY

3.1. Introduction

To attain the purpose of this study, which is to present and implement a metrics-based framework for measuring the reusability of OO components; an inquiry into the current industry practice regarding reuse and reusability assessment was conducted. The inquiry played an important role in attaining the objectives of this study because, it provided an in-depth understanding of reuse and reusability,—which was core in the development of the reusability assessment framework.

According to Sommerville (2011), stakeholders' involvement is crucial for the success of software engineering projects. Based on this, a survey involving OO software developers was conducted in order to establish the status of reuse and reusability. The survey was able to established challenges in reusability assessment and shortcomings of the methods—used in reusability assessment. The survey findings validated published claims that; reuse is lacking adoption by software developers due to lack of proper methods of assessing reusability (Hristov et al., 2012); hence most of the software components being developed lack adequate reusability (Nyasente et al., 2014a).

This study can be divided into three major components: (i) the survey involving OO software developers, (ii) framework development and validation, and (iii) Implementation of the framework. The rest of this chapter presents the methodologies used in attaining the goals of each of these components.

3.2.   Methodology for Establishing the Current Status regarding Reuse and Reusability

3.2.1. Research Design

A descriptive research methodology was adopted in order to establish the current status of reuse and reusability assessment. A survey was administered in order collect primary data from a selected sample of OO software developers. According to Kombo and Tromp(2006), if a study requires the collection of information by interviews, questionnaires etc—with the aim of ascertaining the state of affairs of a phenomenon, then the survey research design would be appropriate for such a study. This study adopted the survey research design—since information was to be collected from OO developers, with the aim of ascertaining the state of affairs with regards to reuse and reusability assessment.

3.2.2. Target Population and the Sampled Population

This study targeted all OO software developers in the republic of Kenya. The size of the population in question could not be established because; no published sources concerning this population could be found. This made it difficult to even estimate the population size, because there is no credible basis for such an estimate. Considering the fact that, it is not practical to draw a sample from a target population (Daniel & Cross, 2013); respondents were selected from software development companies and other organizations that had software development departments—within Nairobi. Nairobi was selected as a study area for reasons of practicability, efficiency, and ease of access.

3.2.3. Sampling design

Non-probability sampling was adopted in selecting sample items for the study. The researcher purposely targeted OO software developers because; the study revolved around reuse and reusability assessment in OO software, and the researcher believed that OO developers had sufficient knowledge on the subject matter—hence reliable for the study.

According to Kothari (2004) non-probability sampling (also known as deliberate sampling, purposive sampling and judgment sampling), is a type of sampling where items for the sample are selected deliberately by the researcher; and his choice concerning the items remains supreme. One of the limitations of purposive sampling is that the researcher never knows if the sample is representative of the population (Kombo & Tromp, 2006).

Also with purposive sampling, sampling error cannot be estimated and the element of bias, great or small is always there (Kothari, 2004). However, the power of purposive sampling lies in selecting information rich cases for in-depth analysis related to the central issue being studied (Kombo & Tromp, 2006). This type of sampling design is often adopted for small inquiries and researches by individuals, because of the relative advantage of time and money inherent in this method of sampling (Kothari, 2004).

3.2.4. Sample Selection

The researcher deployed homogeneous sampling technique to draw a sample of fifty-four (54) respondents from 21 organizations situated in Nairobi Kenya. The organizations were either software development companies, or organizations that had an active department for software development. Homogeneous sampling technique was chosen since the study targeted only OO software developers. According to Kombo and Tromp (2006), homogeneous sampling technique is a type of purposive sampling that picks up a small sample with similar characteristics to describe some particular subgroup in depth.

The organizations from which the sample was drawn from include; Higher Education's loans board, University of Nairobi, Ihub, Kenya Methodist University, Asta, Integral soft limited, Next technologies, Technobrain, Craft silicon, E-mobilis, Amband limited, International livestock research institute (ILRI), Innovation IT solutions, Institute of

31

Software Engineering, Jafftek, Naisoft, Symphony, Egalaxy Kenya, fabtech, TouIT and Movetech solutions

3.2.5. Instrumentation

The study largely employed quantitative methods to collect primary data from respondents, using schedules—consisting of mostly closed ended questions. Schedules were preferred because the researcher believed that the subject on software measurement is somewhat complex, and some respondents required further explanations regarding the survey. Qualitative methodology was also used to gain an in-depth understanding of other complex issues influencing OO reuse and reusability assessment, which would not have been understood, if only quantitative methodology was adopted.

3.2.6. Data Analysis

Statistical package for social sciences (SPSS) version 21.0, was religiously used for the statistical analyses. Coding of variables in quantitative research is very critical for better interpretation of results. The questions and responses from the schedules were coded and entered into the computer using Microsoft Excel 2007 software. This data was later imported into SPSS and analyzed. Appropriate statistical methods were applied on the data to get the results, which were analyzed.

3.3. Methodology for Framework Development and validation

The major contribution of this study is to develop an effective metrics-based framework for assessing the reusability of OO components. This objective was achieved by conducting literature analysis. According to Berndtsson, Hansson, Olsson, and Lundell (2008), literature analysis is a systematic examination of a problem, by means of an analysis of published sources, undertaken with a specific purpose in mind. In the context of this study, literature analysis was conducted with the two objectives in mind: (i) to

develop the reusability assessment framework for OO components, and (ii) to validate the developed framework for superiority.

To achieve objective (i), literature analysis was conducted in order to identify the key elements of a reusability assessment framework for OO software and determine how these elements can be collectively used in measuring reusability. On the other hand, objective (ii) was achieved by performing comprehensive literature analysis for the purpose of establishing a benchmark for evaluating the superiority of the developed framework.

3.4. Methodology for Framework Implementation

The third component of this study is the implementation and testing of the developed reusability assessment framework. This objective was accomplished by developing a reusability assessment system. This section outlines the methodology used to develop the system.

3.4.1. System Design

The system that has both a front-end and back-end applications was built. The front-end application was implemented using the .NET framework and Visual basic programming language. The Microsoft Visual studio 2010 professional, integrated development environment was used in the development of the application. The system's database on the other hand, was implemented using Microsoft SQL server 2008 Database management system. The system database was designed using Toad data modeler; a database design tool that allows users to visually create, maintain, and document new and existing database systems.

3.4.2. System Architecture

The system was implemented using the n-tier application architecture, in which the system is composed of independent components that work in multiple 'tiers' or layers. Writing of multi-tier applications is a common practice in writing independent components that may be stored and run in different machines(Bradley & Millspaugh, 2009). Bradley and Millpaugh contend that the three-tier application model (shown in figure 3.1), is the most widely used multi-tier approach. The tiers of the model are: Presentation, Business, and the Data tier.
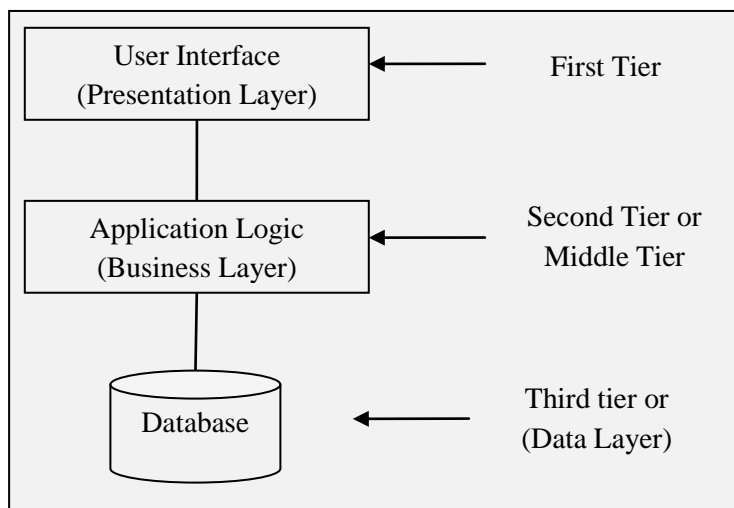
Figure 3.1: The n-tier Architecture for System Development

The presentation tier is also called the client layer and it comprises of components that are dedicated to presenting the data to the user (the user Interface). The user interface for the reusability assessment system consists of windows application forms that comprise of graphical icons. The business tier on the other hand, encapsulates the business rules or the business logic of the application. This tier deals with business rules for data manipulation and transformation into information, and it is also responsible for

processing the data retrieved and sent to the presentation layer. Lastly, the data layer comprises of the Database Components such as database Files, Tables, and Views.

3.5. Summary

This chapter has analyzed the research methodology used in this study. It describes the methodologies used for data collection and analysis, framework design, framework validation, and framework implementation (system design). The chapter starts by highlighting the need for an inquiry into the current industry practice on reuse and reusability assessment in OO software. Next, the research design used for the inquiry and the motivation for the choice is discussed. The population, sampling design, the sample size, and instruments for data collection and analysis are subsequently described. Finally, the methodologies used to develop, validate, and implement the reusability assessment framework are described.

CHAPTER FOUR
DATA ANALYSIS AND DISCUSSIONS

4.1.   Introduction

One of the major components of this research is to establishing the current industry status regarding OO software reuse and reusability assessment.The objective in this case was to examine how reusability assessment is being conducted and determine whether the methodologies used in reusability assessment are effective. To achieve this objective, a survey was administered in order collect primary data from a selected sample of OO software developers.

Data collection was done through schedules that had the same structure and questions—in order to provide consistent results, as well as enable statistical comparisons of different cases. The schedules had five major sections namely; programmer's general background, reuse and reusability issues within the software development cycle, the reuse practice in the organizations, software reusability assessment and, software metrics and reusability assessment. The survey findings on each of these sections are discussed below.

4.2.   Programmer's General Background

This section captured three aspects: years the respondent had worked as a programmer, programming languages known by the respondent and other software development related skills that the respondents had—besides programming. The latter aspects was of interest, because the researcher believes that the practice of reuse and reusability assessment may be influenced by other software development related skills—such as Software engineering (SE), object-oriented analysis and design (OOAD), system analysis and design (SAD), software project management (SPM) etc.

4.2.1. Respondents' Software Development Related Skills

Statistics about other software development related skills possessed by the respondents are presented in table 4.1.

Table 4.1: Software development related skills possessed by respondents

Knowledge	No.	Percent (%)
OOAD, SAD	13	24.1
OOAD, SE	1	1.9
OOAD, SE, SAD	22	40.7
OOAD, SE, SAD, CASE	1	1.9
OOAD, SE, SAD, Mobile software Development	1	1.9
OOAD, SE, SAD, Project Management	1	1.9
OOAD, SE, SAD, Project Scheduling	1	1.9
OOAD, SE, SAD, SPM	1	1.9
OOAD, SAD	1	1.9
SAD	6	11.1
SAD, Database Programming	1	1.9
SAD, Project Management	1	1.9
SE	1	1.9
SE, SAD	3	5.6
Total	54	100.0

A closer look at the results in table 4.1 reveals that, majority—32 (59.3%) of the respondents, had software engineering and software project management skills. Therefore, it can be concluded that, at least 59.3% of the respondents had at least some theoretical knowledge on software metrics. This is based on the fact that, software measurements is a core area in software engineering and software project management (Pressman, 2005).

4.3. Reuse and reusability Related Issues within the development cycle

This section of the schedule was intended to examine some key aspects of software reuse within the stages of software development. Most importantly, it explored reuse within the development cycle, reusability related issues that hamper effective reuse, as well as software development procedures and practices that influence reusability.

4.3.1. Reuse within the Development Cycle

The notion of reuse is an old idea that has been around since human beings became involved in problem solving (Prieto-Díaz, 1993). This also applies to software engineering (Caldiera & Basili, 1991)—where programmers reuse existing components to build new software. This phenomenon was also explored through the survey. The nature of reuse was explored by asking respondents to indicate whether they reused requirements documents, design, and code; when developing new software. An analysis of gathered responses shows that majority of respondents reuse different existing components in developing new software. Statistics regarding this aspect are shown in the table 4.2.

Table 4.2: Statistics on components reuse

Component Reused	No. of Respondents Reusing the Component	Total Number of Respondents
Requirements Documents	41 (75.9%)	54 (100%)
Design	40 (74.1%)	54 (100%)
Code	54 (100%)	54 (100%)

4.3.2. Reusability Related Challenges

Although software reuse is an old concept; it is faced with challenges, and has not acquired real momentum in software engineering (Caldiera & Basili, 1991). The survey also sought to ascertain challenges that impede successful reuse. In particular, challenges that are associated with code reuse, testability, and maintainability were explored.

4.3.2.1. Challenges in Code Reuse

As it can be seen from table 4.2 above, code reuse has the most interesting statistics, where 54 (100%) of the respondents indicated that they often reused code from existing software to build new software. Therefore, it was of interest to know the most significant challenges that respondents face when reusing code. To achieve this, respondents were asked to state the most significant challenges that they faced when reusing code. The most prominent responses to this include: code understandability; integration of existing code into the new system code; debugging errors associated with the reused code; difficulty in finding code that perfectly fits into the new system code; and insufficient in-text documentation (comments).

4.3.2.2. Challenges in Software Testing and Maintenance

According to Ghezzi et al. (2003), maintenance is inevitable when reusing software components. In other words, reusability of a software component is influenced by the ease or difficulty with which that component can be maintained (Nyasente et al. 2014a). Therefore, any challenges that are associated with maintenance are directly related to reusability. In light of this, the researcher sought to know whether or not respondents faced any challenges regarding software testing and maintenance. Majority of the respondents, 36 (66.7%) indicated that they faced challenges when testing and maintaining software. This information is shown in table 4.3.

Table 4.3: Statistics on software testability and maintainability challenges

Statement	Response	Number of respondents
Do you experience challenge when testing and maintaining software?	Yes	36 (66.7%)
	No	18 (33.3%)
	Total	54 (100%)

The most prominent testability and maintainability challenges—as stated by the respondents include; difficulty in modifying existing components, time constraints, difficulty in debugging, difficulty in testing and maintaining software that is developed elsewhere, generating sufficient test cases and test data, lack of experience in testing, poor documentation, lack of testing tools, and lack of a clear testing criteria.

Software complexity is one possible cause for most of the challenges that developers face when reusing code and when testing and maintaining software. According to Laird and Brennan (2006), unnecessary complexity brings about problems such as additional defects, difficulty in understanding code, difficulty in debugging, and maintainability issues.

4.3.3. Procedures and Practices that Influence Reusability

According to Ghezzi et al. (2003), software developers can follow some guidelines in order to produce less complex software. In effect, this will improve other reusability related aspects such as understandability, maintainability and portability (Nyasente et al., 2014a). To explore this aspect, respondents were asked to indicate whether they followed guidelines relating to coupling, cohesion, and inheritance, when designing software. Table 4.4 shows statistics for the given responses.

Table 4.4: Proportion of developers following OO design guidelines/criteria

OO design guideline/criteria	No. of respondents who follow the guideline/criteria	No. of respondents who do not follow the guideline/criteria	Total No. of Respondents
coupling and Cohesion	32 (59.3%)	22 (40.7%)	54 (100%)
Control inheritance hierarchy	41 (75.9%)	13 (24.1%)	54 (100%)

A comparison of the statistics in table 4.4, above with those in the previous table,(table 4.3) reveals that some of the respondents who follow design guidelines—that are intended to produce software that are easy to test and maintain, also experience significant challenges in software testing and maintenance. The true picture of this phenomenon is revealed by creating contingency tables (tables 4.5 and 4.6).

Table 4.5: Cohesion and coupling criteria vs. software testability vs. maintainability challenges

		Experience significant challenges when testing and maintaining software?		Total
		Yes	No	
Follow cohesion and coupling criteria in class design?	Yes	20 (62.5%)	12 (37.5%)	32 (100%)
	No	16 (72.7%)	6 (27.3%)	22 (100%)

As it can be observed from table 4.5, majority 20 (62.5%) of the respondents who follow cohesion and coupling criteria in class design experience significant challenges when testing and maintaining software; however, this percentage is higher by 10.2%, for the respondents who do not follow the said criteria. This situation also holds when it comes to the aspect of controlling inheritance hierarchies during class design. As it can be seen from table 4.6 below, the percentage of respondents who experience significant challenge when testing and maintaining software is lesser for respondents who control inheritance hierarchies during class design, as compared to the percentage of respondents who do not.

Table 4.6: Controlling of inheritance hierarchies vs. software testing and maintenance challenges

		Experience Challenge When testing and maintaining software?		Total
		Yes	No	
Control inheritance hierarchy during class design?	Yes	24 (58.5%)	17 (41.5%)	41
	No	12 (92.3%)	1 (7.7%)	13

By observing the above contingency tables, it can be concluded that, some challenges that are related to software testing and maintenance can be resolved if developers follow design guidelines relating to cohesion, coupling and inheritance.

A closer look at the marginal totals in tables4.5 and 4.6 above, reveal that a significantly high number of respondents who follow the specified design guidelines still face testability and maintainability challenges. This prompted an investigation into the effect that the said guidelines have on software testing and maintenance. This is achieved through studying the linear correlation between the number of respondents who follow the design guidelines, and the number of respondents who face software testability and maintainability challenges. The linear correlation analysis results are shown in the table below.

42

Table 4.7: Correlation between the number of developers who follow design guidelines and those facing testability and maintainability challenges

		Experience Challenge When testing and maintaining software	Follow Cohesion and Coupling Criteria in Class design	Control inheritance hierarchy during class design
Experience Challenge When testing and maintaining software	Pearson Correlation	1	-.107	-.306[*]
	Sig. (2-tailed)		.443	.024
	N	54	54	54
Follow Cohesion and Coupling Criteria in Class design	Pearson Correlation	-.107	1	.415[**]
	Sig. (2-tailed)	.443		.002
	N	54	54	54
Control inheritance hierarchy during class design	Pearson Correlation	-.306[*]	.415[**]	1
	Sig. (2-tailed)	.024	.002	
	N	54	54	54
**. Correlation is significant at the 0.01 level (2-ailed).				
*. Correlation is significant at the 0.05 level (2-tailed).				

As it can be observed from the above table, there is a negative correlation (of −0.306 with a *p* value of .024) between the number of respondents who control inheritance hierarchies during class design, and the number of respondents who experience challenges when testing and maintaining software. During data collection respondents replied in the affirmative or otherwise, whether or not they control inheritance hierarchies during class design, and whether or not they experience challenges when testing and maintaining software. During the coding of variables, 1 represented YES,

whilst 2 represented NO—for the two cases. The negative correlation between the variables is an indication that, when the average value for one variable tends to 1 (YES), the average value for the other will tend to 2 (NO). This means that, the more the number of respondents who control inheritance hierarchies during class design, the lesser the respondents who face testability and maintainability challenges.

The fact that table 4.7, does not show any correlation between the number of respondents who follow coupling and cohesion criteria during class design, and the number of respondents who face testability and maintainability challenges; does not necessarily mean that there is no relationship between the two. To study whether a relationship exists or not, partial correlation between the number of respondents who control inheritance hierarchies during class design, and the number of those who experience testability and maintainability challenges is studied—where the effect of following cohesion and coupling criteria in class design is controlled on the two variables. The results of the partial correlation are shown in table 4.8.

Table 4.8: Partial correlation between the number of developers who control inheritance hierarchies and those who face testability and maintainability challenges

Control Variables			Control inheritance hierarchy during class design	Experience Challenge When testing and maintaining software
Follow Cohesion and Coupling Criteria in Class design	Control inheritance hierarchy during class design	Correlation	1.000	-.290
		Significance (2-tailed)	.	.035
		df	0	51

As it can be observed from table 4.8, the partial correlation (-.290) is smaller than the simple correlation (-.306). This suggests that *following cohesion and coupling criteria in class design* partly contributed to the liner correlation between the number of respondents who control inheritance hierarchies during class design, and the number of those who experience testability and maintainability challenges. This means that following cohesion and coupling criteria eliminates some of the challenges associated with software testing and maintenance.

Notwithstanding the fact that following the said guidelines resolves some testability and maintainability problems, a significantly high number of respondents who follow the guidelines still face testability and maintainability challenges—(as shown in tables 4.5 and 4.6). One possible explanation to this is that; developers hardly follow the guidelines to the latter—or rather, an objective way of assessing how well they follow the guidelines is lacking.

4.3.4. Use of Technology in Software Development vs. Reusability

Another aspect that was explored is the use of technology at different stages of software development, and its effect on reusability. From the data collected, 34 (63%) of the respondents indicated that they often use computer-aided software engineering (CASE) tools in requirements modelling and analysis, 29 (53.7%)indicated that they used computerized support in class design, whilst 30 (55.6%) of them indicated that they often use code generators to translate design into code. This information is given in table 4.9.

Table 4.9: Statistics on technology use in software development

Tool/Technology	No. of Respondents Using Technology	Total No. of respondents
CASE tools in Requirement modeling	34 (63%)	54 (100%)
Computer support for Class design	29 (53.7%)	54 (100%)
Use of Code Generators	30 (55.6%)	54 (100%)

According to Mahapatra, Das, and Pradhan (2012), CASE tools are often used in various stages of systems development life cycle to improve software quality and productivity. This perspective is explored by creating contingency tables (tables 4.10, 4.11, and 4.12), where the levels of satisfaction with respect to the benefits of reuse, (i.e. software quality, productivity, and effort), are compared for respondents who use technology to aid certain software development activities and those who do not use technology.

Table 4.10, shows the levels of satisfaction regarding quality of software between respondents who use CASE tools in requirements modeling and those who do not.

Table 4.10: Satisfaction levels regarding software quality vs. use of CASE tools in requirements modeling

		Satisfied with quality of developed software					**Total**
		Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree	
Use of CASE tools in requirement modeling and analysis	Yes	0 (0%)	1 (2.9%)	7 (20.6%)	13 (38.2%)	13 (38.2%)	34 (100%)
	No	1 (5%)	1 (5%)	5 (25%)	10 (50%)	3 (15%)	20 (100%)
Total		1 (1.9%)	2 (3.7%)	12 (22.2%)	23 (42.6%)	16 (29.6%)	54 (100%)

The marginal totals of the above contingency table show that, respondents who use CASE tools in requirement modeling are highly satisfied with quality of software that they develop than those who do not use CASE tools. The same situation replays when the levels of satisfaction—with respect to time and effort needed to test and modify software—for those using computerized support in class design, are compared with the levels of satisfaction of those who do not use computerized support in class design. The cross tabulation analysis results are displayed in table 4.11.

Table 4.11: Satisfaction regarding time and effort in testing and modifying software vs. use of computerized support in class design

		Satisfied with time & effort required to test, deliver & modify delivered software					**Total**
		Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree	
Use Computerized support in class design	Yes	1 (3.4%)	4 (13.8%)	8 (27.6%)	11 (37.9%)	5 (17.2%)	29 (100%)
	No	2 (8%)	10 (40%)	4 (16%)	3 (12%)	6 (24%)	25 (100%)
Total		3 (5.6%)	14 (25.9%)	12 (22.2%)	14 (25.9%)	11 (20.4%)	54 (100%)

It is evident from the above table that respondents who use computerized support in class design are highly satisfied with the time and effort required to; test, deliver, and modify software, as compared to respondents who do not use computerized support. Interestingly, the levels of satisfaction with respect to software quality are not significantly different for respondents who use code generators and those who do not. This is shown in table 4.12.

Table 4.12: Satisfaction levels regarding software quality vs. use of code generators

		Satisfied with quality of developed software					**Total**
		Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree	
Use of Code generators	Yes	1 (3.3%)	1 (3.3%)	6 (20%)	16 (53.3%)	6 (20.0%)	30 (100%)
	No	0 (0.0%)	1 (4.2%)	6 (25%)	7 (29.2%)	10 (41.7%)	24 (100%)
Total		1 (1.9%)	2 (3.7%)	12 (22.2%)	23 (42.6%)	16 (29.6%)	54 (100%)

One important conclusion that can be drawn from table 4.12 is that, software quality is not largely dependent on code quality: other factors such as quality of design, and how well the software meets requirements, are very important when it comes to software quality.

4.4. The Reuse Practice in Organizations

The success of any reuse program is largely dependent on how reuse itself is practiced. Prieto-Díaz (1993) contends that, the problem in software engineering is not lack of reuse, but lack of wide spread systematic reuse. This aspect was explored—with a keen interest on policies and traditions that govern reuse, as well as perceptions towards reuse within the organizations.

4.4.1. Organizations' Reuse Policies and Traditions

From the data collected, only 13 (24.1%) of the respondents indicated that their organizations have a software reuse program/policy. On the other hand, 54 (100%) of the respondents indicated that they often reuse parts of existing software in new software developments. This information is given in table 4.13.

Table 4.13: Organizations' software reuse policies and the reuse practice

Aspect inquired about	No. of responses in the affirmative	Total No. of respondents
Reuse policy in place within the organization	13 (24.1%)	54
Reuse parts of existing software in new software development	54 (100%)	54

4.4.2. Perceptions towards Reuse and Reusability

Successful reuse demands for a new way of thinking, and a new way of thinking requires change—which in turn disturbs status-quo, costs money, and requires commitment at all levels (Prieto-Díaz, 1993). To examine the perception of respondents towards reuse, respondents were to indicate the extent to which they agreed or disagreed to some statements that are related to the software reuse practice within their organizations. The respondents were given five options to choose from: *Strongly Disagree (1), Disagree (2), Neutral (3), Agree (4), strongly Agree (5)*.Table 4.14 gives a summary of the responses.

Table 4.14: Respondents' perceptions on software reuse

Statement	N.	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
-Cases of developing software from scratch have significantly diminished over time	54	4 (7.4%)	6 (11.1%)	2 (3.7%)	21 (38.9%)	21 (38.9%)
-The time and effort required to modify available classes within the organization to fit new reuse contexts is often insignificant as compared to creating new classes	54	1 (1.9%)	3 (5.6%)	10 (18.5%)	24 (44.4%)	16 (29.6%)
-The cost and effort for developing software has significantly diminished over time.	54	2 (3.7%)	7 (13.0%)	15 (27.8%)	13(24.1%)	17 (31.5%)
-I prefer developing classes from scratch than reuse classes that are developed by my colleagues	54	5 (9.3%)	3 (5.6%)	18 (33.3%)	16 (29.6%)	12 (22.2%)

From the table above, the total number of respondents who *agreed* and those who *strongly agreed* with the first three statements were 42 (77.8%), 40 (74%) and 30 (55.6%) respectively. However, the fourth statement: I prefer developing classes from scratch than reuse classes that are developed by my colleagues—(a contradiction of the first three statements), got interesting responses. Most of the respondents (51.8%) indicated that they prefer developing classes from scratch rather than reuse classes that are developed by others, while only 14.9% of respondents disagreed with this view. This may be as a result of two factors: (i) the not-invented-here syndrome—a situation where

developers feel hindered in their creativity and independence if they reuse someone else's software (Sametinger, 1997), and (ii) some of the existing components have inadequate reusability. These two issues can be resolved by; developers changing their perceptions towards reuse, and organizations setting up reuse programs—as well as motivate their software developers to reuse.

4.4.3. Payoff from Reuse

As it can be observed from table 4.13, 54(100%) of the respondents reuse existing components in developing new software. The extent to which organizations benefit from this reuse was examined—by asking respondents to indicate the extent to which they agreed or disagreed with four statements that are related to software reuse benefits. Table 4.15 gives a summary of the responses for each of the statements.

Table 4.15: Respondents' views on the payoff from reuse

Statement	N.	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
I am satisfied with the time and effort that is always required to, test, deliver, and maintain new software to our clients.	54	3 (5.6%)	14 (25.9%)	12 (22.2%)	14 (25.9%)	11 (20.4%)
I am satisfied with budget and cost aspects for developing new software applications and their maintenance.	54	8 (14.8%)	10 (18.5%)	21 (38.9%)	12 (22.2%)	3 (5.6%)
I am satisfied with the quality of new software applications we develop as an organization.	54	1 (1.9%)	2(3.7%)	12 (22.2%)	23 (42.6%)	16 (29.6%)
I am satisfied with the overall productivity of developers in the organization.	54	1 (1.9%)	9 (16.7%)	16 (29.6%)	15 (27.8%)	13 (24.1%)

Table 4.15, shows that the respondents who *agreed,* and those who *strongly agreed* with the first two statements were less than 50%, (i.e. 46.3% and 27.8%) respectively. This is

in spite of the fact that all 54 (100%) of the respondents indicated that they reuse parts of existing software when developing new software. This means that, organizations are not gaining maximum payoff from reuse. It is also a consequence of the informal nature of reuse across the organizations. This can be seen from table 4.13, where only 24.1% of respondents indicated that their organizations had a reuse program/policy. This explanation is consistent with the assertion by Prieto-Díaz (1993) that, substantial pay-off from reuse is only achieved if conducted systematically and formally.

Although 39 (72.2%) of the respondents indicated that they were satisfied with the quality of new software applications that they developed, majority were not satisfied with the effort and time it took to develop these software. This means that it takes a lot of effort and time to achieve the desired quality.

4.5.   Software Reusability Assessment

The survey also sought to explore reusability assessment within the organizations, as well as the attributes used to assess reusability. Respondents were required to respond in the affirmative or otherwise, whether they assessed reusability when developing *for* reuse or *with* reuse. Only 21 (38.9%) of the respondents responded in the affirmative, whereas majority 33 (61.1%) of the respondents responded otherwise. This information is given in table 4.16.

Table 4.16: Statistics on Reusability assessment

Statement	N.	Yes	No
Do you always ascertain if classes are reusable when developing or reusing them?	54	21 (38.9%)	33 (61.1%)

4.5.1.   Reusability Attributes and Reusability Factors

One of the objectives of this study was to establish the characteristics that should be used in assessing reusability of OO components. This information that was crucial in the

development of a reusability assessment framework, which is the overriding purpose for this research. This information was sought by asking respondents who indicated that they assessed reusability when developing components *for* or *with* reuse, to state the characteristics that they used in assessing reusability. The characteristics that were stated include; ability of a component to perform required functionality, ease of testing, portability across different platforms, proper use of abstraction and inheritance, easy to understand and adapt, class independence, proper documentation, have public accessor methods and be part of a hierarchy with an interface, consistency in naming methods, classes should be as generic as possible, well commented and documented, tested and used before (Reuse history). Most of these characteristics are consistent with those listed in various literatures (e.g. (Caldiera & Basili, 1991; Ghezzi et al., 2003; ; Nyasente et al., 2014a Washizaki et al., 2003)).

4.5.2. Existing Methods for Reusability Assessment

The methods used by respondents to assess reusability were also explored by asking respondents in the category that assessed reusability to state whether or not they had formal methods for assessing reusability. Out of 21 respondents only 4 (19%), indicated that they had formal methods, whilst 17 (81%) indicated otherwise. This information is shown in table 4.17.

Table 4.17: Statistics on developers with formal reusability assessment methodologies

	Have formal methods for assessing whether classes have reusability characteristics or not		Total
	Yes	No	
Ascertain if classes are reusable when developing for or with reuse	Yes	4 (19%)	17 (81%)	21

Respondents who had formal methods for assessing reusability were further required to state methods that they used. The methods that were stated by the respondents include—observing/checking source code, reading documentation, intuition, and checking inline comments. These methods are not reliable in reusability assessment, as they are largely subjective, and they provide no means of ascertaining the extent to which a given reusability attributes is present in a component (Nyasente et al., 2014a).

4.6.   Software Metrics and Reusability Assessment

The survey also sought to explore the use of software metrics within the software development industry. Aspects that were explored with this regards include; organizations' software measurement policies, respondent's experience with software metrics, and use of metrics in reusability assessment.

4.6.1.   Software Measurement Programs/Policies in the Organizations

From the data collected, only 12 (22.2%) of the respondents indicated their organizations had software measurement programs/policies, whereas 42 (77.8%) of the respondents indicated otherwise. This information is shown in table 4.18.

Table 4.18: Statistics on organizations with software measurement programs/policies

Statement	Response	No. of respondents
Does your organization have a software measurement program/policy?	Yes	12 (22.2%)
	No	42 (77.8%)
	Total	54 (100%)

The small number of organizations with software measurement programs/policies shows that organizations are not keen in embracing software measurement as a way of improving software quality. This may be as a consequence of the lack of an agreement within the software engineering industry on how to measure software quality. That is,

there is no common agreement as to which attributes form a sufficient basis for assessing software quality (Pressman, 2010).

4.6.2. Respondent's Experience with Software Metrics

To explore the experience of respondents regarding software metrics, five statements were put forward, and respondents were required to choose one. The summary of responses for each of the statements is shown in table 4.19.

Table 4.19: Respondents' experience with metrics

Responses	No. of Respondents
Never heard about them	6 (11.1%)
Heard about them but never used them	11 (20.4%)
I have knowledge on Metrics but never used them	24 (44.4%)
I have used Metrics before but stopped using them	1 (1.9%)
I always use software Metrics	12 (22.2%)
Total	54 (100%)

4.6.3. Use of metrics in reusability assessment

Although table 47.19 shows that, 12 (22.2%) of the respondents always use software metrics, none of them responded in the affirmative when asked whether or not they used metrics to measure reusability. This is shown in table 4.20. Further explanation was sought, and it was established that those who use metrics, used software parametric models and estimation tools to estimate projects' duration, effort, and cost of developing software.

Table 4.20: Statistics on reusability measurement

Statement	Response	Number of Respondents.	Total Number of Respondents
Do you measure the reusability of classes when developing for or with reuse?	No	54 (100%)	54 (100%)

4.6.4. Impediments to Reusability Measurement

In the quest of exploring the impediments to reusability measurement, respondents were asked why they did not measure reusability. This question elicited the following responses: *as a programmer the only concern is to develop and deliver working software within time; I do not know how to measure reusability; I do not know how reusability measurement is helpful; I have limited knowledge on the subject; I have never considered measuring software to be useful; it is hard to apply metrics in practice; I lack practical knowledge on how to measure reusability; there is lack of reusability measurement tools; our measurement policy does not cover product quality; as a developer, I have tight deadlines hence I focus on delivering software on time by all means...*

The above reasons can be attributed to; (i) lack of sufficient knowledge on software metrics and software quality measurement, (ii) organizations' measurement policies fail to cover some quality aspects such as reusability, (iii) lack of parametric tools for measuring reusability, and, (iv) time constraints.

4.7. Conclusion

This chapter has presented findings of a survey involving OO developers—regarding reuse and reusability assessment. The developers general background was explored, where majority of developers were found to have knowledge on software measurement and metrics. The survey also explored the various aspects within the software development cycle, with a keen interest on reuse. Evidence showing that use of technology has a positive impact on reusability was gathered. It was also established that reusability does not only rely on quality of source code alone, but also on other factors—such as quality of design. The survey was also able to establish that developers face some significant challenges when reusing software. This is attributed to reusability related issues (i.e. most of the components lack reusability). It was also of importance to

explore the root cause of this, and it was found out that reuse was largely being practiced in an opportunistic manner; since none of the organizations—where the developers worked, had reuse programs or policies. Most importantly, the survey established that, majority of developers did not assess reusability, and the few who did, had no objective methods of doing so. The survey established that the reusability assessment methods used by developers are inefficient, since they provide no way of ascertaining the level to which a given component possesses certain reusability attributes. This necessitates the development of an efficient method for assessing reusability.

CHAPTER FIVE
FRAMEWORK DEVELOPMENT AND IMPLEMENTATION

5.1.   Introduction

The importance of reusability assessment with regards to software reuse was made evident through literature review. This notwithstanding, literature also pointed out to the lack of an effective framework for measuring the reusability of OO software components as a major unresolved issue. This formed the basis for conducting a survey, in order to determine if literature was consistent with industry practice. The findings of the survey were consistent with published literature, as it was established that the methods used by practitioners in reusability assessment were not effective—as they were largely subjective. Consequently, a metrics-based framework for assessing the reusability of OO software components is presented in this section.

5.2.   Framework Development

A thorough understanding of reusability as well as adequate and easy to use metrics is requisite in reusability assessment. Thus, a framework describing reusability of software components as well as structuring appropriate metrics for quantifying reusability is required (Hristov et al., 2012). This research presents a novel framework that; relates major reusability attributes with factors that are determined by measurable OO principal constructs, and structures metrics for measuring the OO constructs in a way that is easy to use.

The reusability assessment framework is presented in the subsequent subsections as follows: first the major reusability attributes are presented; then factors that influence the reusability attributes are related with different OO design structures, thereafter candidate metrics for measuring the OO structures are given, and lastly an equation for calculating

the reusability of OO components is defined. The hierarchy of the key elements of the reusability assessment framework is shown in figure 5.1 below.
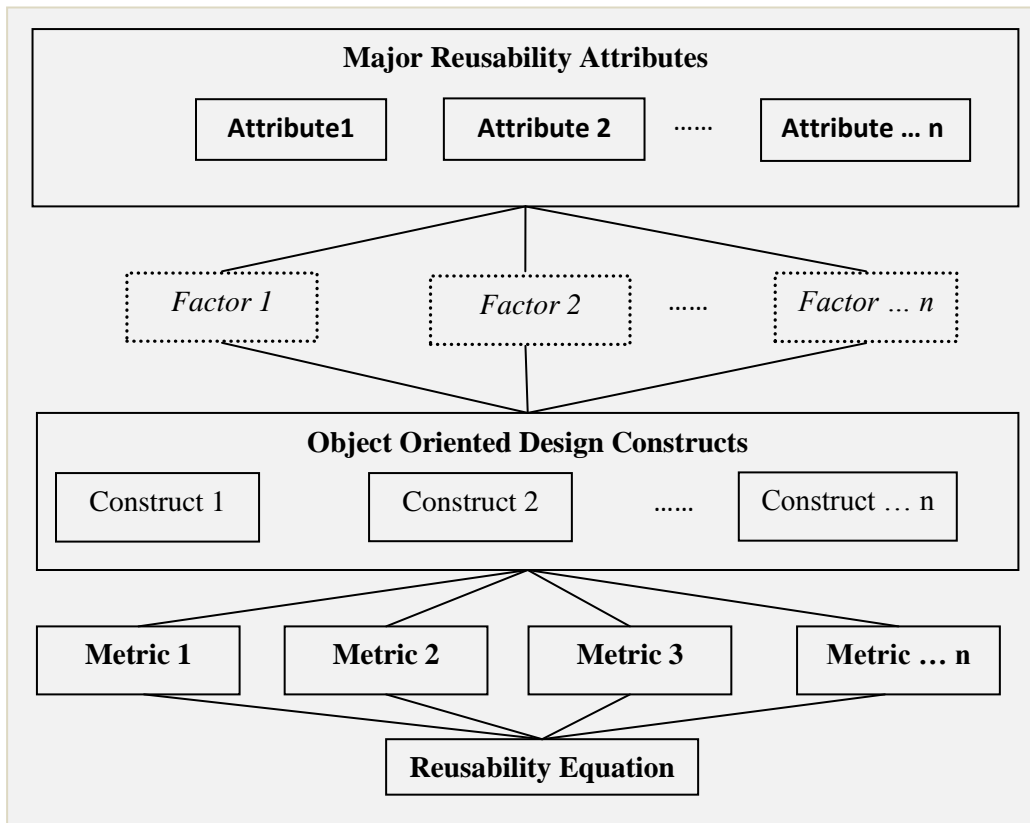
Figure 5.1: Hierarchy of key elements for the OO reusability assessment framework

5.3. Major Reusability Characteristics for Software Components

A review of literature, revels that there are many characteristics that are believed to influence reusability of software components. Such characteristics have been presented in various research works (AL-Badareenet al., 2010; Caldiera & Basili, 1991; Hristov et al., 2012; Nyasente et al., 2014c; Washizaki et al., 2003). This view is also consistent with the survey findings—where respondents listed various attributes, which they believe influence reusability.

One of the problems in reusability assessment—according to Hristov et al. (2012)is in determining which attributes are sufficient in assessing reusability. Westfall (2005) has commented in this context by stating that software entities possess many attributes that are measurable, and if all of these attributes are considered, then there are just too many measures, and it would be easy to drown an organization in the enormity of the task of trying to measure everything. According to Nyasente et al. (2014a), an effective reusability assessment framework should have as few attributes as possible, but at the same time sufficient in assessing all aspects of reusability. That is, overlapping and trivial attributes should be excluded from such a framework.

The survey findings presented in the previous chapter—as well as published literature was analyzed and the major attributes that should be used to assess the reusability of software components were identified. These attributes are presented below.

5.3.1. Generality

Generality is defined by the IEEE Standard 610.12, as the degree to which a system or component performs a broad range of functions. AL-Badareenet al. (2010) as well as Caldiera and Basili (1991) consider generality as one of the reusability attributes in their reusability assessment frameworks—with Caldiera and Basili using the term functional usefulness (usefulness) to mean generality.

Generality is a major reusability attribute because; it increases reusability of a component (Gill & Sikka, 2011; Sommerville, 2011). This means that, if generality of a component increases, its probability to be reused increases, and if it decreases its probability to be reused decreases as well. In other words, if a software component does not possess the generality property, it cannot be reused.

According to Návrat and Filkorn (2005), generality is an inevitable characteristic of reusable assets, and things get reusable only if they are general and allow turning to specific in a clear and straightforward manner. This explanation is consistent with the survey findings—as class generality and adaptability were cited as characteristics that influence reusability.

5.3.2. Understandability

According to Hristov et al. (2012), a software component is more usable if it is can be easily understood. More often than not, a developer will decide to reuse a component based on how well the component meets new requirements. Therefore, the starting point for reusing a component is to understand its functionality, which requires high understandability (Washizaki et al., 2003). According to Washizaki et al., understandability is defined based on the estimated effort needed by a user to recognize the concept behind a component and its applicability.

Intuitively, the effort needed by the user to understand a component in order to recognize the concept behind it and its applicability, is determined by how easy and straightforward the design and implementation of that component is. Thus, understandability is synonymous with simplicity—which is defined by IEEE std 610.12 as; the degree to which a system or component has a design and implementation that is straightforward and easy to understand.

The ease with which a component can be adapted and tested, are among the major reusability attributes stated by respondents in the survey conducted by the researcher. According to Ghezzi et al. (2003), these two qualities and many other require a high level of understandability. Therefore, understandability should be considered as a major reusability attribute.

5.3.3. Portability

Portability is also among the reusability attributes listed by respondents who took part in the survey. According to the IEEE Std 610.12, portability is the ease with which a system or component can be transferred from one hardware or software environment to another. Portability is a major determinant of reusability because; if a module can easily work in different environments, the more it is likely to be reused. For example: consider two modules (say m_1 and m_2) that are equal in terms of functionality, and it happens that m_1 is not compatible with certain software or hardware environments, but m_2 is compatible with "all" environments, then; the probability of m_2 being reused over m_1 is increased. That is, m_1's reuse will be limited to the extent of its incompatibility.

According to Ghezzi et al. (2003), portability is economically important because it helps amortize the investment in the software system across different environments and different generations of the same environment. This means, the payoff from reuse is higher for components that are environment independent. That is, the ability of a component to run in different environments will save the cost of developing a new component for new environments.

5.3.4. Maintainability

IEEE Std 610.12 (1990) defines Maintainability as, the ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment. Ghezzi et al. (2003), distinguish three categories of maintenance, that is; corrective, adaptive, and perfective.

Corrective maintenance deals with the removal of residual errors that are present in the product when delivered, as well as errors introduced into the software during its maintenance. Perfective maintenance on the other hand, involves changing the software to improve some of its qualities. Here changes are due to the need to modify the

functions offered by the application, add new functions, improve the performance of the application, make it easier to use etc. Lastly, adaptive maintenance, involves adjusting the applications to changes in the environment.

Although adaptive maintenance was the only aspect captured in the responses gathered from the survey; all the three aspects of maintenance are important when a component is reused in a new context. When reusing a component, developers want to reuse a component that is easy to: modify in order to fit its reuse context; modify the functions offered by the component, add new functions, improve the performance of the component, make it easier to use etc; and, remove residual errors that are present in the component when delivered—as well as errors introduced into the component during its maintenance. Ghezzi et al. (2003) emphasize on the importance of the maintainability characteristic by stating that, there is evidence that maintenance costs exceed 60 percent of the total costs of software, with corrective and adaptive maintenance accounting for about 20 percent each, whilst perfective maintenance absorbs over 50 percent.

5.3.5. Documentation

Documentation is also among the reusability attributes listed by respondents in the survey. It is of utmost importance because, it is intended to make software components easier to understand (AL-Badareenet al., 2010; Hristov et al., 2012). According to Ghezzi et al.(2003), understandability is a factor in product usability. van Vliet (2000), comments on the importance of documentation by stating that software which is not sufficiently documented is bound to incur high costs later on. For example, maintenance is hampered by lack of proper documentation (Ghezzi et al, 2003; van Vliet, 2000). Although documentation is largely subordinate to understandability, it should be considered as a major reusability attribute (Nyasente et al., 2014a), due to the fact that it gets the worst attention (van Vliet, 2000), which results to effects that counteract the objectives of reuse.

5.4. Relating Reusability Attributes with Reusability Factors and OO Structures

In this section, factors that influence maintainability, portability, understandability, and generality are related with OO design structures that influence them—to facilitate their quantification by measuring the OO structures using appropriate metrics that exists in literature. Documentation on the other hand is not related to any OO design structure, but it can be determined as suggested by (Hristov et al., 2012)—that is by use of four attributes: amount, quality, completeness, and, availability of legal terms and conditions.

Figure 5.2., shows the relationship between the major reusability attributes that are discussed above, and the measurable factors that determine them. A discussion on the factors that influence the reusability attributes follows.
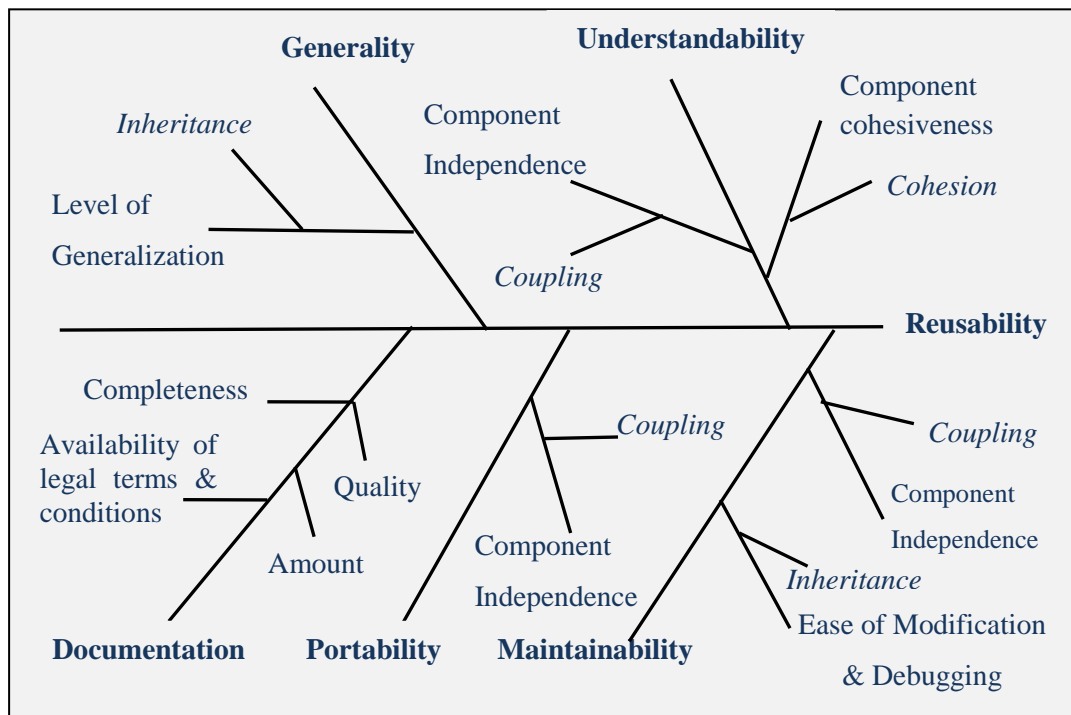
Figure 5.2: Reusability factors for OO components

5.4.1. Factors Influencing Maintainability vs. OO Structures

5.4.1.1. Ease of Modification and Debugging

According to Ghezzi et al.(2003), maintainability involves two aspects; that is reparability and evolvability. The former deals with correction of defects (debugging), whilst the latter involves modifying the software to satisfy new requirements. Software is said to be maintainable if these two aspects can be achieved with a reasonable amount of work (Nyasente et al., 2014 a).

According to Laird and Brennan (2006), the difficulty of maintaining software is brought about by increased software complexity. In OO design, complexity of software is increased if inheritance is not used in proper range, i.e. if inheritance is overused or misused (Chawla & Nath, 2013). This means that, ease of debugging and modification, can be achieved by measuring inheritance, to determine if it has been used in proper ranges and if not, the design should be reviewed and improved.

5.4.1.2. Component Independence

Coupling characterizes a module's relationship to other modules. It determines the interdependence of modules, where modules that are dependent on each other heavily are said to have high coupling (Ghezzi et al., 2003). When classes of a system are highly dependent on each other, it is more likely that changing one class will affect other classes of the system (Sommerville, 2011). This means that, high interdependence between classes makes evolvability (software modification) difficult to perform. For instance, if one class of a given component is modified—say; to fit a new reuse context, other classes of that component to which the class is dependent on may require modification as well. Therefore, reusable classes should exhibit a high degree of independence (i.e. low coupling). This means that component independence can be improved by measuring coupling—in order to determine components with designs that should be reviewed.

5.4.2. Factors Influencing Portability vs. OO Structures

According to Ghezzi et al. (2003), low coupling enables a module to be reused separately. That is, components with low coupling are easy to reuse in new software environments. Sametinger (1997) contends that low coupling is important with respect to component portability, as a component is also indirectly dependent on platforms of components with which it interacts.

Washizaki et al. (2003) consider external dependency as one of the factors that affect portability. According to the authors, external dependency indicates the component's degree of independence from the rest of the software, which originally used it. In other words, external dependency characterizes a component's relationship to other components. Thus, external dependency and coupling are semantically equal in this context.

5.4.3. Factors Influencing Generality vs. OO Structures

Generality of OO software is achieved through generalization, i.e. by factoring out what is common to different components in one class (known as the parent class), and then single out the variations in heir classes (subclasses). More often than not, all features that are likely to be sufficiently general to be reused are factored out in the parent class (Ghezzi et al., 2003). Generalization is implemented using inheritance mechanisms built into the OO languages, where heir classes—that are derived from the parent classes absorb all reusable features that are factored out in the parent class (Sommerville, 2011). According to Gill and Sikka (2011), the level of generalization of a class is determined by its relative abstraction level.

5.4.4. Factors Influencing Understandability vs. OO Structures

5.4.4.1. Component Cohesiveness and Component Independence

According to Ghezzi et al. (2003),a component should have high cohesion and low coupling for it to be understandable. Ghezzi et al. state that, different elements of a module cooperate to perform the functionality of that module; thus, these elements are grouped together for logical reasons and not by sheer chance. A module is said to be highly cohesive when all its elements are strongly related (Ghezzi et al., 2003).

According to Cho et al. (2001), lack of cohesion or low cohesion increases software complexity, whilst high cohesion increases understandability. According to Ghezzi et al. (2003), a high level of component independence enables components to be analyzed and understood separately. That is, if a component is highly dependent on other components, reference to the components to which it is dependent on is required in order to understand it. This cross-reference is minimized if the degree of component independence is high; hence, understanding that component becomes easier (Nyasente et al., 2014a).

5.5. Candidate Metrics for the Framework

Upon a survey of literature, insights on the actual metrics that can be used to measure the reusability attributes discussed above were gained. The preliminary suggestions of the metrics are given below:

5.5.1. Measuring Maintainability

The two maintainability factors: component independence and, ease of maintenance and debugging are determined by coupling and inheritance respectively. Thus, coupling metrics such as the coupling between object classes (CBO) metric can be used to determine component independence (Chidamber & Kemerer, 1991, 1994), with low values for CBO indicating high degree of independence. The number of children (NOC)

Metric on the other hand, can be used to determine the ease of debugging and modification. NOC is an inheritance-based metric defined by (Chidamber & Kemerer, 1991, 1994). Low values for NOC indicate a low degree of Component's complexity, hence easy to modify and debug it.

5.5.2. Measuring Understandability

The factors that influence understandability: component independence and component cohesiveness; are related to coupling and cohesion respectively. The CBO metric can be used to determine component independence—with low CBO values being desirable. Cohesiveness on the other hand, can be measured using cohesion metrics such as the lack of cohesion in methods (LCOM) metric (Chidamber & Kemerer, 1991, 1994), where low values of LCOM are desirable.

5.5.3. Measuring Portability and Generality

Component portability is determined by component independence. Thus, the CBO metric can be used to determine portability. Generality of a component on the other hand, is determined by a component's level of generalization, which is determined by its relative abstraction level (Gill & Sikka, 2011). This concept is related to inheritance; therefore, inheritance-hierarchy-based metrics—such as the generality of class (GC) metric, can be used to measure the generality of classes(Gill & Sikka, 2011)—where high values of GC indicate a high degree of generality.

5.5.4. Measuring Documentation

There are four factors used to determine documentation: amount of documentation; quality; completeness; and, availability of legal terms and conditions. The amount of documentation can be measured through size, e.g. in kilobytes (kB) etc, whereas the existence of legal terms and conditions is a Boolean metric: either this information is provided or not (Hristov et al., 2012).

Hristov et al. (2012) state that, quality and completeness are subjective measures that should be measured on an ordinal scale based on advice of an expert. However, quality can be determined by evaluating certain features for producing quality documentation, whereas completeness of the documentation can be determined by evaluating whether or not all of its components are available (Nyasente et al., 2014a).

According to Sommerville (2001), documentation quality can be determined by considering document structure, documentation standards and writing style. Document structure is the way in which the material in the document is organized. This has a major impact on readability and usability and it is important to design this carefully when creating documentation. Good structure allows each part of documentation to be read as a single item, and reduces problems of cross-referencing when changes have to be made.

Documentation Standards on the other hand, ensure that produced documentation has a consistent appearance (Sommerville, 2001). According to Sommerville, documentation standards are dependent on the nature of the project; therefore, it is important that appropriate standards that suit each project are chosen. In addition to structure and standards, good documentation is fundamentally dependent on the writing style (i.e. the writer's ability to construct clear and concise technical prose). That is; good documentation requires good writing.

Lastly, Sommerville (2001) gives a suggestion of seven parts that documentation for large systems that are developed to a customer's specification should include, and three parts that documentation for small systems that are developed as software products should have. Reusable components fall into the latter category (Nyasente et al., 2014a), and documentation for such systems should include at least the following parts: specification of the system, an architectural design document, and, the program source code (Sommerville, 2001).

5.6.   Equation for Calculating the Reusability of OO Components

To facilitate the measurement of OO components' reusability, it is necessary to define a reusability equation. The equation is based on the four elements, discussed above i.e. the major reusability attributes, factors that influence the reusability attributes, OO constructs that influence each factor, and the metrics for measuring these constructs. These elements are shown in figure 5.3.
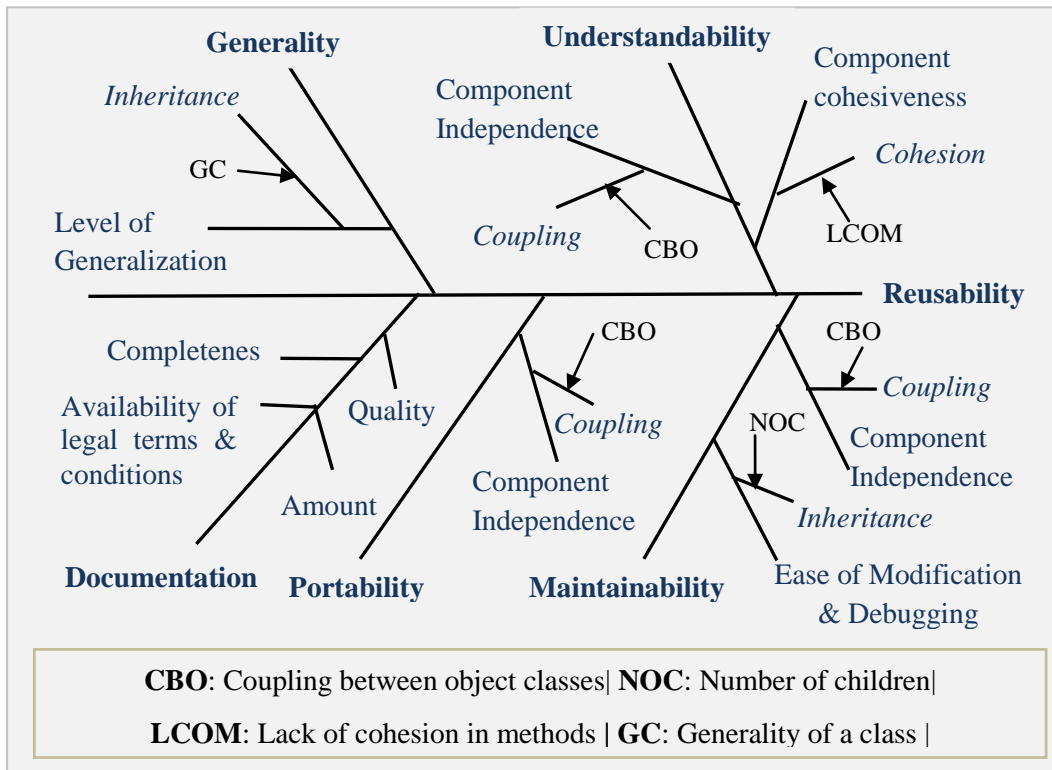
Figure 5.3: Reusability attributes model for OO components

Theoretically, the reusability of a software component (denote by R_c), can be calculated by adding up values obtained from measuring the five reusability attributes using appropriate metrics—(as indicated in figure 5.3). Therefore, reusability can be calculated using the relationship:

$$R_C = Maintainability + Portability + Documentation + Generality$$
$$+ Understandability \qquad (5.1)$$

69

The five reusability attributes are considered to be of equal importance; hence, weighting values are assigned to them, because some attributes are influenced by more factors than others are. Therefore, the reusability of an OO software component can be calculated using the expression:

$$R_c = w_1.Mai + w_2.Port + w_3.Doc + w_4.Gen + w_5.Und \qquad (5.2)$$

Where:

w_1 to w_5 are weighting values, and *Mai*, *Port*, *Doc*, *Gen*, and *Und*; are composite metrics for maintainability, portability, documentation, generality, and understandability.

The composite metrics values for the reusability attributes should be adjusted to a common scale to facilitate comparison of reusability of different components in the same context (Hristov et al., 2012). Hristov et al. contend that normalizing these values to the range of (0...1), is common in software metrics. The values of the weights; w_1, w_2, w_3, w_4 and w_5 are 0.2, 0.1, 0.4, 0.1, and 0.2 respectively. This is because, a varying number of factors determine each reusability attribute, and there are ten factors in the reusability attributes model. The weighting value for a given attribute is obtained by dividing the number of factors that influence it, by the total number of factors in the model (10).

To obtain the reusability of a software component (R_c), metrics values for each of the reusability attributes should be obtained by using appropriate metrics to measure the factors that influence that attribute—(with metric values for attributes that are determined by multiple factors being normalized to the range of (0...1)), then these composite metrics values should be aggregated into the expression shown in equation 5.2.

5.7. Experimentation of the Framework

To demonstrate how the new framework can be used to assess reusability, it is used to measure the reusability of a sample java component—obtained from (Deitel P. & Deitel H., 2011). The UML block diagram of the component is shown in figure5.4. The class diagram was modified to make class Employee a subclass of Java.lang.Object because; all java classes inherit from class Object (Deitel P. & Deitel H., 2011).
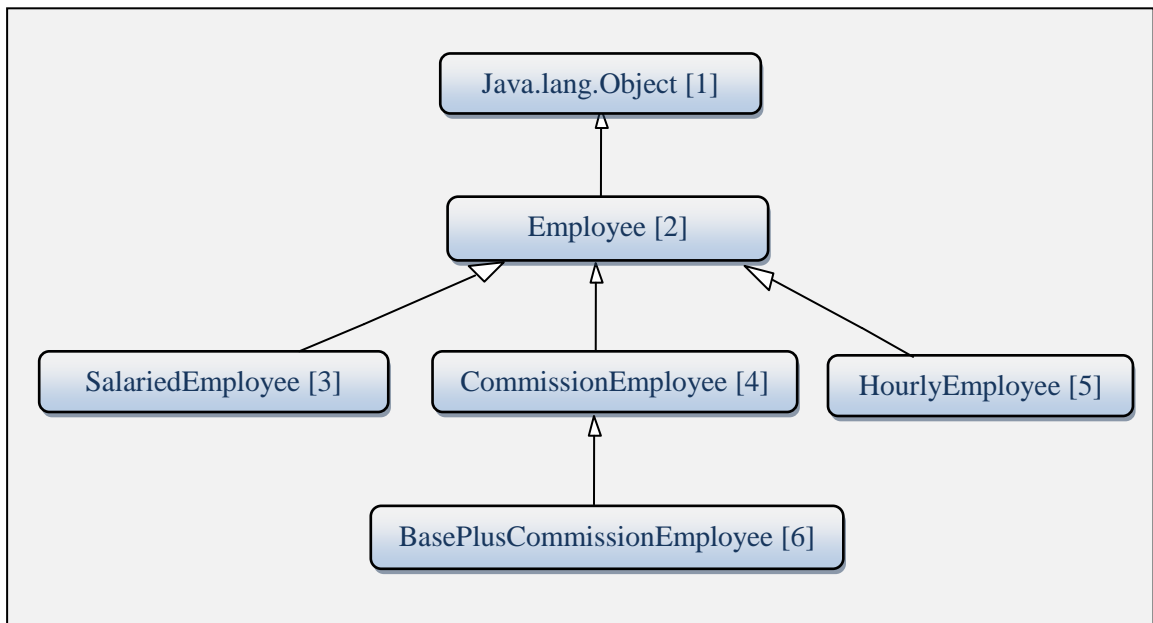
Figure 5.4: Class hierarchy for a sample OO component (adapted from (Deitel P. & DeitelH., 2011))

The methods as well as instance variables in each of the classes are listed in tables 5.1 to 5.5.

Table 5.1: Instance variables for the Employee class

Method	Instance Variables
<<Constructor>>Employee	firstName, lastName, socialSecurityNumber
setFirstName	firstName
getFirstName	firstName
setLastName	lastName
getLastName	lastName
setSocialSecurityNumber	socialSecurityNumber
getSocialSecurityNumber	socialSecurityNumber
toString	firstName, lastName, socialSecurityNumber
earnings	ABSTRACT

Table 5.2: Instance variables for the SalariedEmployee subclass

Method	Instance Variables
SalariedEmployee	firstName, lastName, socialSecurityNumber, weeklySalary
setWeeklySalary	weeklySalary
getWeeklySalary	weeklySalary
earnings	weeklySalary
toString	firstName, lastName, socialSecurityNumber, weeklySalary

Table 5.3: Instance variables for the HourlyEmployee subclass

Method	Instance Variables
HourlyEmployee	firstName, lastName, socialSecurityNumber, wage, hours
setWage	wage
getWage	wage
setHours	hours
getHours	hours
earnings	wage, hours
toString	firstName, lastName, socialSecurityNumber, wage, hours

Table 5.4: Instance variables for the CommissionEmployee subclass

Method	Instance Variables
CommissionEmployee	firstName, lastName, socialSecurityNumber, grossSales, commissionRate
setCommissionRate	commissionRate
getCommissionRate	commissionRate
setGrossSales	grossSales
getGrossSales	grossSales
earnings	commissionRate, grossSales
toString	firstName, lastName, socialSecurityNumber, grossSales, commissionRate

Table 5.5: Instance variables for the BasePlusCommissionEmployee subclass

Method	Instance Variables
BasePlusCommission-Employee	firstName, lastName, socialSecurityNumber, grossSales, commissionRate, baseSalary
setBaseSalary	baseSalary
getBaseSalary	baseSalary
earnings	baseSalary, commissionRate, grossSales
toString	firstName, lastName, socialSecurityNumber, grossSales, commissionRate, baseSalary

5.7.1. Measuring OO Features of the Sample Component

5.7.2. Definition of the Metrics

The metrics that were suggested in section 5.5, are defined in this section, and thereafter used to measure different OO features of the sample component shown in figure 5.4 above.

73

5.7.2.1. Coupling Between Object Classes (CBO) Metric

Singh et al. (2011) define coupling as, the measure of strength of association established by a connection from one entity to another. The CBO metric is used to measure how much coupling exists between classes (Sommerville, 2011). The CBO of a class is obtained by counting the number of other classes to which that class is coupled with (Chidamber & Kemerer, 1994). CBO relates to the notion that an object is coupled to another object if methods of one object uses methods or instance variables of another (Chidamber & Kemerer, 1994). According to Chidamber and Kemerer (1991), any evidence of a method of one object using methods or instance variables of another object constitutes coupling.

5.7.2.2. Number of Children (NOC) and Generality of Class (GC) Metrics

The NOC of a class is the number of immediate subclasses subordinated to it in the class hierarchy (Chidamber & Kemerer, 1991, 1994). Generality of Class (GC) on the other hand is the measure of its relative abstraction level, and it is obtained by dividing the abstraction level of the class by the number of abstraction levels in the class hierarchy (Gill & Sikka, 2011).

5.7.2.3. Lack of Cohesion in Methods (LCOM) Metric

Cohesion can be defined as, the degree to which methods of a class are related to one another and work together to provide well bounded behavior (Singh et al., 2011). The LCOM metric is used to measure the cohesiveness of a class, by using instance variables to measure the degree of similarity of methods of a class, and it is defined as (Chidamber & Kemerer, 1994):

> Consider a class C_1 with n methods, $M_1, M_2, ..., M_n$.
>
> Let $\{I_j\}$ = set of instance variables used by method M_i. There are n such sets $\{I_1\}$, ... , $\{I_n\}$.

Let $P = \{(I_i, I_j) \mid I_i \cap I_j = \emptyset\}$ and $Q = \{(I_i, I_j) \mid I_i \cap I_j \neq \emptyset\}$. If all n sets $\{I_1\}$, ... , $\{I_n\}$ are $\emptyset$ then let $P = \emptyset$.

$$LCOM = |P| - |Q|, if \ |P| > |Q|$$
$$= 0 \quad otherwise \qquad\qquad (5.3)$$

Example (Chidamber & Kemerer, 1994): Consider a class C with three methods M_1, M_2 and M_3. Let $\{I_1\} = \{a, b, c, d, e\}$ and $\{I_2\} = \{a, b, e\}$ and $\{I_3\} = \{x, y, z\}$. $\{I_1\} \cap \{I_2\}$ is nonempty, (i.e. $\{I_1\} \cap \{I_2\} \neq \emptyset$), but $\{I_1\} \cap \{I_3\}$ and $\{I_2\} \cap \{I_3\}$ are null sets. **LCOM** is (the number of null intersections – number of nonempty intersections), which in this case is 1. According to Chidamber and Kemerer (1994), the LCOM value provides a measure of relative desperate nature of methods of a class.

5.7.3. Obtained Values for the CBO, NOC, LCOM, and GC Metrics

The values for CBO, NOC, LCOM, and GC metrics that were obtained after measuring different OO structures of the sample component using the criteria outlined above are summarized in tables 5.6 and 5.7, and then the interpretations of these values follow.

Table 5.6: Obtained values for NOC, CBO and GC for the sample component

Metric	Values	Classes						
		1	2	3	4	5	6	**Total**
	Computed	0.2	0.6	0	0.2	0	0	1
NOC	Maximum	1	1	1	1	1	1	6
	Computed	0.2	0.8	0.2	0.4	0.2	0.2	2
CBO	Maximum	1	1	1	1	1	1	6
	Computed	1	0.75	0.5	0.5	0.5	0.25	3.5
GC	Maximum	1	1	1	1	1	1	6

NOC: table 5.6, shows that the computed value of NOC for the sample component is 1. This gives a normalized NOC value of 0.17. This value is obtained by dividing 1 by the

maximum NOC value (i.e. 6). The lesser the value computed for NOC the lesser is the component's complexity, hence easy to debug and modify. The, the NOC value can be viewed as *the difficulty of debugging and modification* (Nyasente et al., 2014a) Therefore ease of debugging and modification can then be obtained by subtracting the "*difficulty of debugging and modification*" from 1, since the highest possible value for ease of debugging and modification is 1. Thus, the value for *ease of debugging and modification* for the sample component is 0.83.

CBO: from table 5.6,the computed value for CBO is 2, compared to the maximum value of 6.When we normalize this value we obtain 0.33.Since the CBO value shows the, degree of interdependence between classes, then the degree of independence can be obtained by subtracting the degree of interdependence from 1; where 1 is the highest possible (normalized) value for the degree of independence. Therefore, the degree of independence for the sample component is 0.67. Although literature suggests that, CBO should be measured by counting the number of distinct non-inheritance related class hierarchies on which a class depends (Cho et al., 2001; Sharma & Dubey, 2012); couplings due to inheritance are considered in the computation of CBO like in the case of (Chawla & Nath, 2013).

GC: table 5.6 shows that the computed value for GC is 3.5, compared to the maximum value of 6. Normalizing this value, we get 0.58. Therefore, the level of generalization for the sample component is 0.58. This is based on the fact that, the GC metric indicates the level of generalization for a component (Gill & Sikka, 2011).

LCOM: The number of non-null intersections of instance variable pairs of methods ($|Q|$), for each of the classes of the sample component is greater than the number of null intersections ($|P|$), (i.e. $|P| < |Q|$); therefore, the LCOM value for all classes of the sample component is 0. This is based on the definition of LCOM (as presented in section

5.7.2.3, above). To find the cohesiveness of a class, its LCOM value is subtracted from 1—as the LCOM value measures the relative desperateness of a class, and the highest possible (normalized) value for cohesion is 1. Therefore, average *cohesiveness* for the component is 1. The computed LCOM and cohesion values for each class of the component are summarized in table 5.7.

Table 5.7: Summary of the LCOM measure for the sample component

Class	No. of Methods	Highest possible LCOM Value	Computed (Normalized) LCOM value	Cohesiveness
1				
2	8	28	0	1
3	5	10	0	1
4	7	21	0	1
5	7	21	0	1
6	5	10	0	1

The computed LCOM values are normalized with respect to the highest possible LCOM value, which is the total number of paired instance variables of methods in a class. That is to say that, a class has the highest value for LCOM if none of its paired methods have similar instance variables (i.e. when $|Q| = 0$).

5.7.4. Metrics Values Obtained from Measuring Documentation

5.7.4.1. Obtained Value for Quality

The computed value for documentation quality is 15 out of a possible value of 15. Therefore, the normalized value for quality is 1. The criteria used in calculating the value for quality is given in table 5.8.

Table 5.8: Criteria for measuring documentation quality for the sample component

Factor	Criterion	Comment	Value on a scale of 1-5	Max value
Quality	Document structure	-Well structured.	5	5
	Document standards	-Good programming practice adopted (e.g. Excellent use of comments). -Standard notation used (e.g. use of UML for class diagrams)	5	5
	Writing style	-Clear and concise technical prose used.	5	5
TOTAL			**15**	**15**

5.7.4.2. Obtained Value for Completeness

Two out of the three documentation components are given, thus documentation can be said to be 67% complete. In other words, the degree of completeness of the documentation is 0.67. The criterion used in assessing completeness is given in table 5.9.

Table 5.9: Criteria for measuring completeness of documentation for the sample component

Documentation Component	Provided	Not Provided
System Specification	✓	
An architectural design document		✓
The program source code	✓	

5.7.4.3. Obtained Value for Availability of Legal terms and Conditions, and amount of documentation

The legal terms and conditions for use of the entire text (book) are available; therefore, a value is 1 assigned. On the other hand, the amount of documentation can assume three possible categorical values: very bulky, bulky, and compact. These values are given ranks: 1, 2, and 3 respectively. Prima-facially the amount of documentation provided for

the component's documentation is small (i.e. compact); therefore, the value for amount of documentation is 3. Since 3 is the highest possible value for the amount of documentation, the obtained value (3) is normalized to give a value **1**.

5.7.5. Aggregating Metrics Values into the Reusability Equation

To calculate the reusability of the sample component, the composite metrics for the five attributes are first calculated, and then aggregated into the reusability equation (equation 5.2). The obtained values from measuring the component are as below:

Mai = 0.5 (Component independence + Ease of modification and debugging)

=> 0.5(0.67 + 0.83) = 0.75

$Port$ = Component independence => 0.67

Doc = 0.25(0.67 + 1 + 1 + 1) = 0.92

Gen = Level of generalization => 0.58

Und = 0.5 (Component independence + Cohesiveness) => 0.5(0.67 + 1) = 0.84

Therefore:

R_c = 0.2 (0.75) + 0.1(0.67) + 0.4(0.92) + 0.1(0.58) + 0.2(0.84) = 0.811

5.7.6. Interpretation of the Reusability Value (R_c)

Since the composite metrics values for the five attributes are normalized to the range of (0...1), the value for Rc will always be between 0 and 1. Rc shows the reusability level of a component—where high Rc values (values close to 1) indicate high reusability. Low values for Rc indicate low reusability—which is an indication of possible flaws in the system design. Therefore, components with low R_c values should be subjected to further review. For the case of the sample component, the obtained R_c value: 0.811, is relatively high (compared to the maximum value of 1). Therefore it can be concluded that reusability for the component is high (i.e. 81%).

5.8. **Framework Implementation**

One of the objectives of this study was to build a prototype of a reusability assessment system—based on the reusability framework that is presented in the preceding subsections of this chapter. This section presents the design and development process of the initial version of the system.

5.9. **System Design and Development**

5.9.1. **Requirements Analysis**

This section begins by describing the different users of the system and their roles, and subsequently presents system requirements—(both functional and non-functional), based on user needs and roles.

5.9.2. **System Users and Their Roles**

a) **Administrator**: This is a user who has administrative rights of the system. The roles of the administrator include:

- Creating user accounts. This includes setting system privileges to users.
- Managing user accounts (editing and deleting user accounts).

b) **Software Quality manager:** This is a user who is responsible for monitoring the software engineering processes and methods used to ensure quality. The specific functions of the software quality assurance manager include:

- Coordinating the daily activities of the quality assurance staff.
- Developing, reviewing, and publishing standards, policies and procedures for all functions involved with or related to the quality and testing of software products.
- Inspecting completed quality control checklists, forms and other documents for conformance to prescribed standards.

- Reviewing and resolving of software quality control problems related to production of software services or products.

c) **Object Oriented Software Developer:** This is a user who designs, installs, tests, and maintains Object-Oriented software systems. The specific functions of the Object-oriented software developer include:

- Reviewing current systems and presenting ideas for system improvements, including cost proposals.
- Working closely with analysts, designers and the Software quality assurance manager.
- Writing and testing code, and then refining and writing as necessary.
- Testing the software products in controlled situations before going live, as well as maintaining of software systems.

5.9.3. System Requirements

The functional and the nonfunctional requirements for the reusability assessment system are summarized in tables5.10 and 5.11 respectively.

a) Functional requirements

Table 5.10: Functional requirements of the reusability assessment system

ID	Requirement
FR-1	Take the number of couplings for each component's class as input, and compute the CBO metric for the component.
FR-2	Take the number of immediate subclasses for every class as input, and compute the NOC metric for the component.
FR-3	Take the number of abstraction levels of the component's class hierarchy, and the abstraction levels of each class in the hierarchy—as input, and compute the GC metric for the component.
FR-4	Take the number of disjoint and non-disjoint method pair of each component's class as input, and compute the LCOM measure of the component.
FR-5	Take the values for document structure, document standards, writing style, availability of legal terms & conditions, and completeness of documentation as input, hence compute the component's documentations 'quality index'.
FR-6	Store the results of requirement FR-1 – FR-5 in a database.
FR-7.	Compute/derive the Reusability of a component (RC), from the results of requirement FR-1 – FR-5.
FR-8.	Generate reusability reports of components.
FR-9	Capture user account details and store them in a Database.
FR-10	Provide for different user views, based on the user type.

b) Nonfunctional requirements

Table 5.11: Nonfunctional requirements of the reusability assessment system

ID	Requirement
NFR-1	The system should guard against accidental deletion and erroneous update of stored data.
NFR-2	The system should provide for user authentication.
NFR-3	The system should check and verify that entered data is in the appropriate format
NFR-4	The system should have adequate understandability, testability, maintainability, and reusability.

5.9.4.   Use Cases for the Reusability Assessment System

The reusability assessment system is to be used by both OO developers and Software quality assurance managers (SQA managers). Developers will use the system in measuring the reusability of software components, whilst the Software Quality assurance managers will use the system in monitoring the reusability of developed components. The system will have a system administrator, who will have the overall administrative rights of the system. The roles of the three system users are depicted in the system-level use-case diagram shown in figure 5.5.
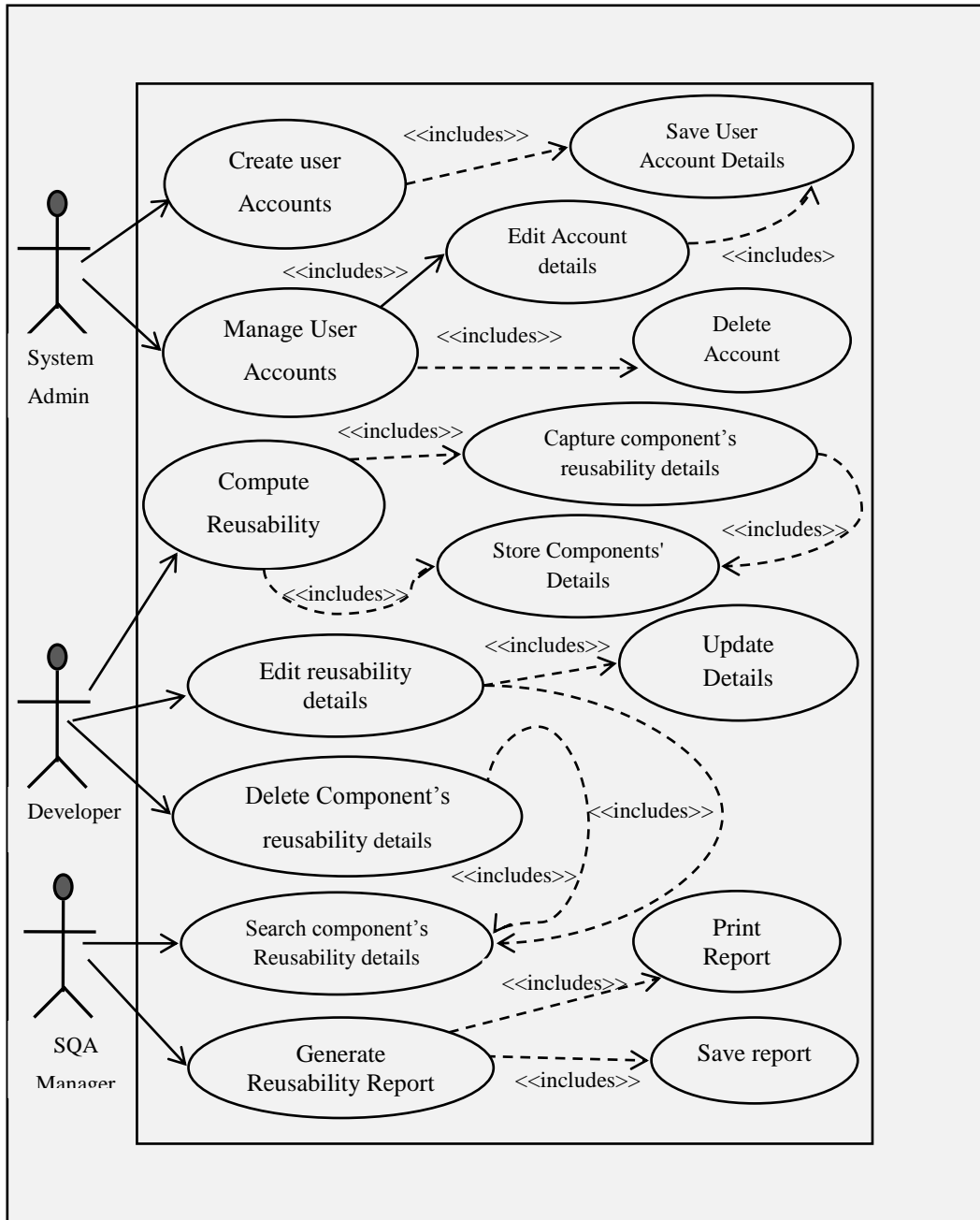
Figure 5.5: System level use-case diagram for the reusability assessment system

84

5.9.5. Database Design

A database for storing the reusability details of components as well as user accounts was built using Microsoft SQL server 2008—which is a relational database management system. The identified entities and attributes for the database are shown in figure 5.6.
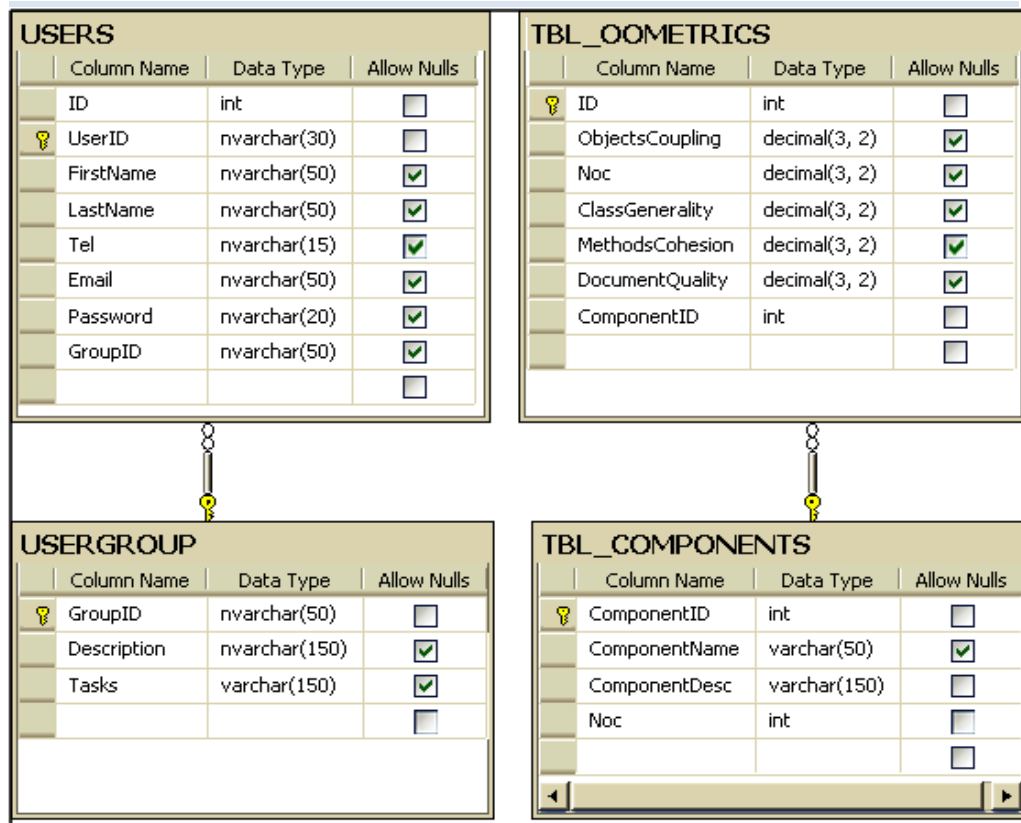
Figure 5.6: Database design for the reusability assessment system

5.9.6. Class Design

5.9.6.1. The Data Tier Class Design

The data layer for the application is comprised of four public classes, i.e. Cypher, MyControls, Validation, and Layer1. The inheritance hierarchy for the data layer classes is shown in figure 5.7.
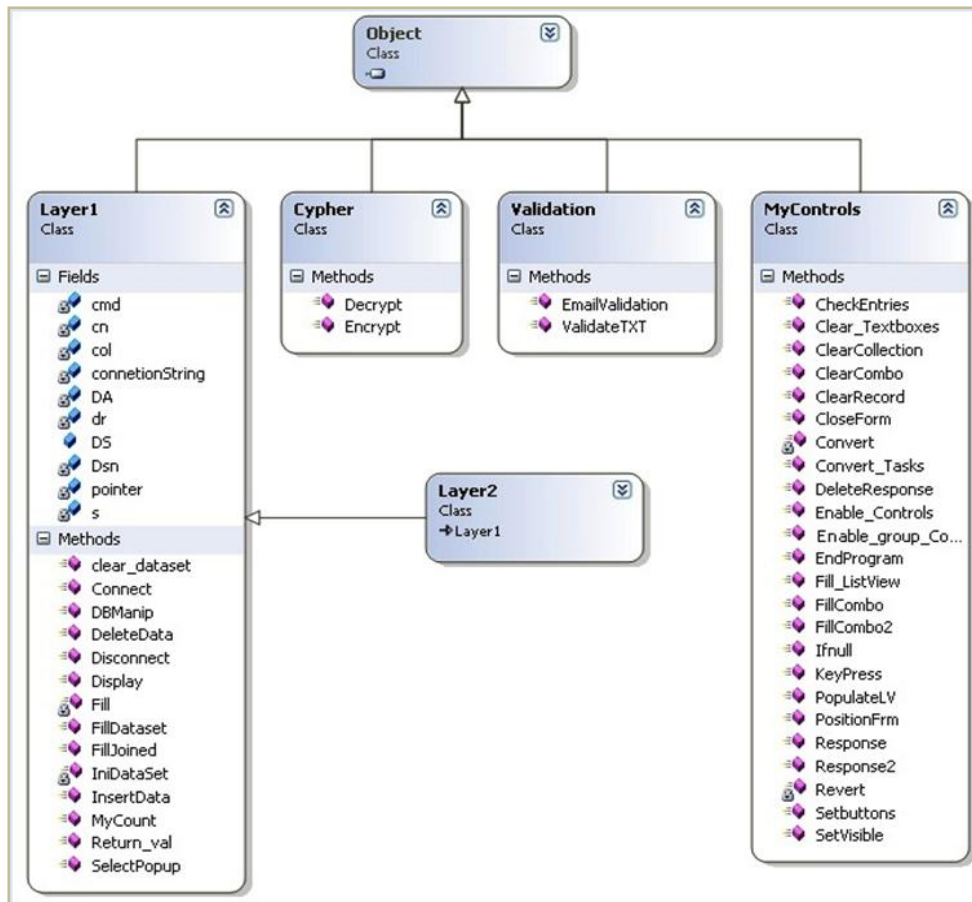
Figure 5.7. Data layer classes for the reusability assessment system

i) **Public Class Cypher**: this class comprises of methods for encrypting and decrypting passwords that are stored in the database.

ii) **Public Class MyControls**: comprises of methods for presenting controls as required by the user.

iii) **Public Class Validation**: comprises of methods that ensures that all user input is provided as required. That is, it validates if all required fields are provided.

iv) **Public Class Layer1**: comprises of fields and methods for writing, reading, and manipulating the database. That is, it contains the SQL for reading and writing to the database. The class *Layer1* has one subclass: *Layer2*.

5.9.6.2. The Business Tier Class Design

The Business layer (Layer 2) for the application encapsulates business logic for data manipulation and transformation of the data into information. It is also responsible for processing the data retrieved from the database and sends it to the presentation layer. The business Layer for the system has one class, namely *Layer2*, which inherits from class, *Layer1*. The business layer class and its members are shown in the figure below.
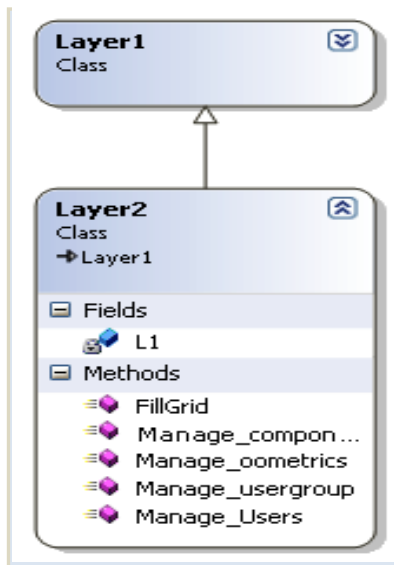
Figure 5.8: The business tier classes for the reusability assessment system

5.9.6.3. The presentation Tier Class Design

This layer comprises of components that are responsible for presenting system data to the user/user interface. It also presents user input to the business layer. The major Layer 3 classes for system are:

i) **Class frmUser:** This Layer 3 component handles the creation and management of user accounts. It includes methods for displaying user account information that

exists in the system database, as well as methods that enable the user to crete new user accounts.

ii) Class frmSystem: This Class consists of methods that display computed metrics information for OO components that exist in the system database, as well as methods that enables a user to enter/supply metrics values for new components.

iii) Class frmComponents: This class includes methods that enables a user to add new OO components (i.e. general description of the components) into the system.

iv) Class frmDocumentation: This class includes methods that enables the user to supply metrics information about the documentation of a given component.

The class diagram for the above Layer 3 classes is shown in figure 5.9.
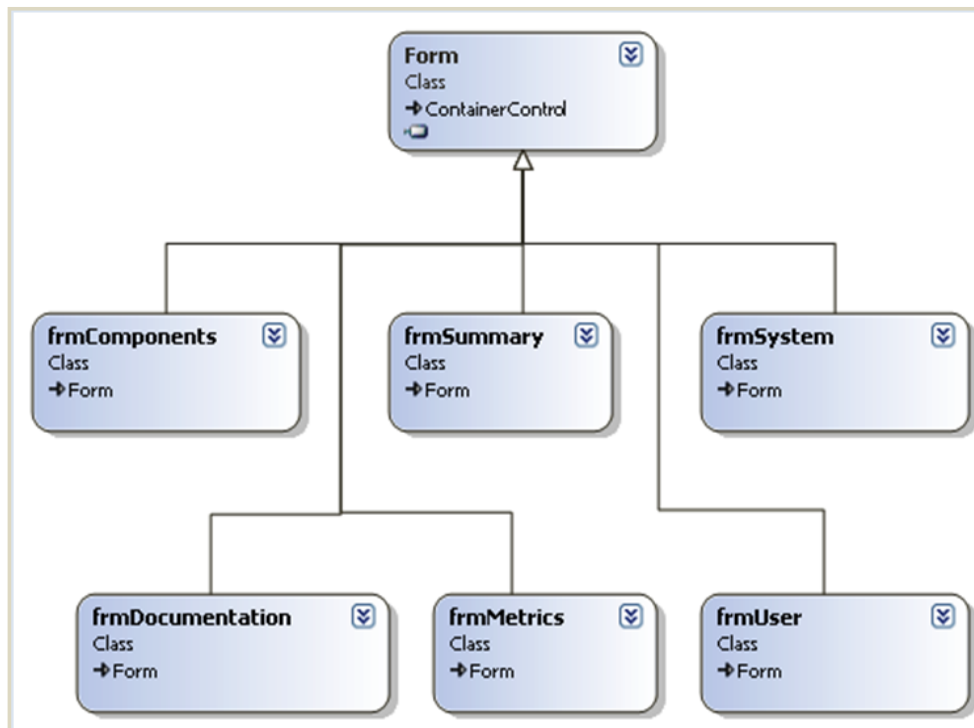
Figure 5.9: Presentation layer classes for the reusability assessment system

5.10. System/Program Flow

The tasks performed by the system can be grouped into two major categories: (i) managing OO components/metrics, and (ii) managing of system users. The stepwise activities and actions undertaken to achieve these tasks are depicted in in figures 5.10 and 5.11 respectively.
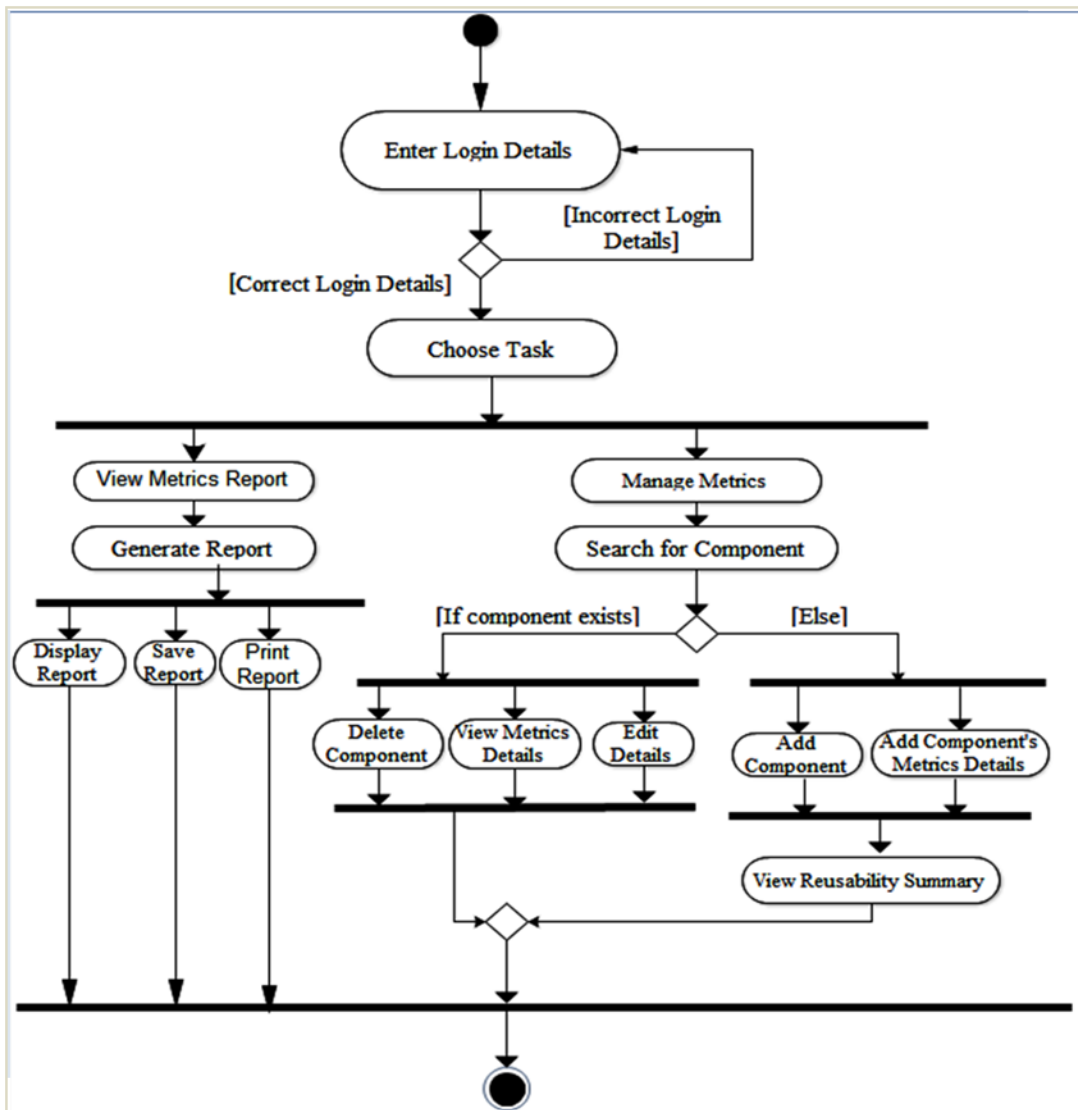
Figure 5.10: Activity diagram for the task of managing metrics and components
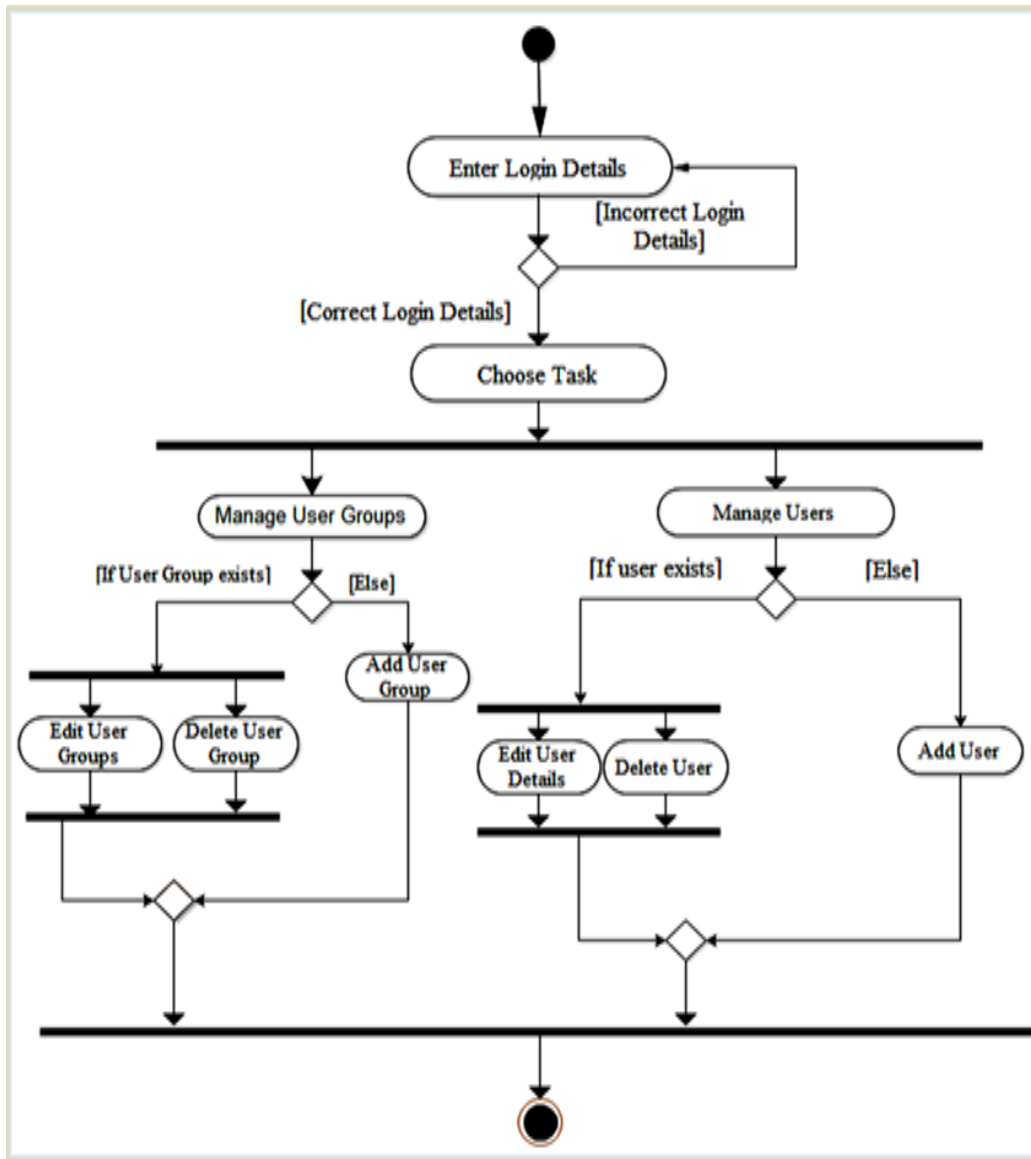
Figure 5.11: Activity diagram for the task of managing users and user groups

5.11.   Major User Interfaces for the Reusability Assessment System

5.11.1. The Login Module

This module enables users to gain access to the system. For a user to gain access to the system, he must supply the correct user name and password. The user name and password are then matched with the account details (user information) stored in the

system's database. The user gains access to the system if there is a match—otherwise, the user is notified that the supplied information is invalid. The screenshot for the login interface is displayed below.
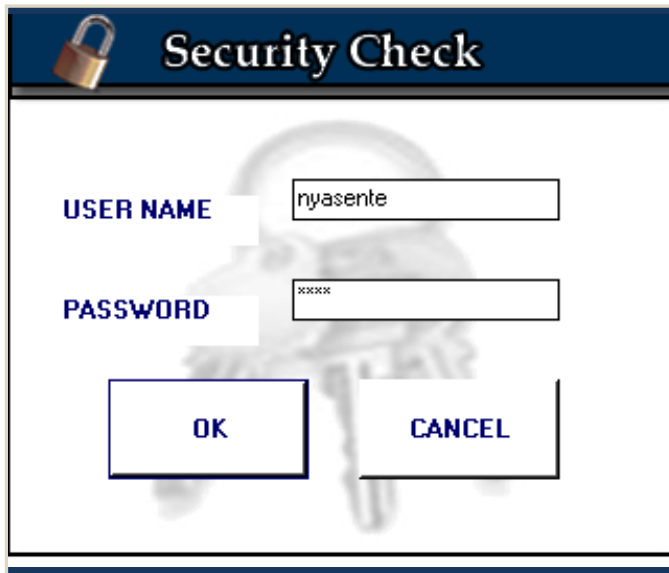
Figure 5.12: The login interface for the reusability assessment system

5.11.2. The System's Main Interface

After a user has successfully logged into the system, the main system user interface is displayed. This interface, displays three major tasks that the user can perform: manage users, manage metrics and view metric reports. The choice of a particular task displays the relevant corresponding sub-interface. The main user interface for system is displayed in figure 5.13.

Figure 5.13: Main user interface for the reusability assessment system

5.11.3. Interface for Managing Users

When the user who is logged in as an administrator chooses the 'manage users task' from the main interface, an interface for managing users (shown in figure 5.14), is displayed. From this interface, the user can view and edit existing user accounts and user groups, as well as create new user accounts or user groups.
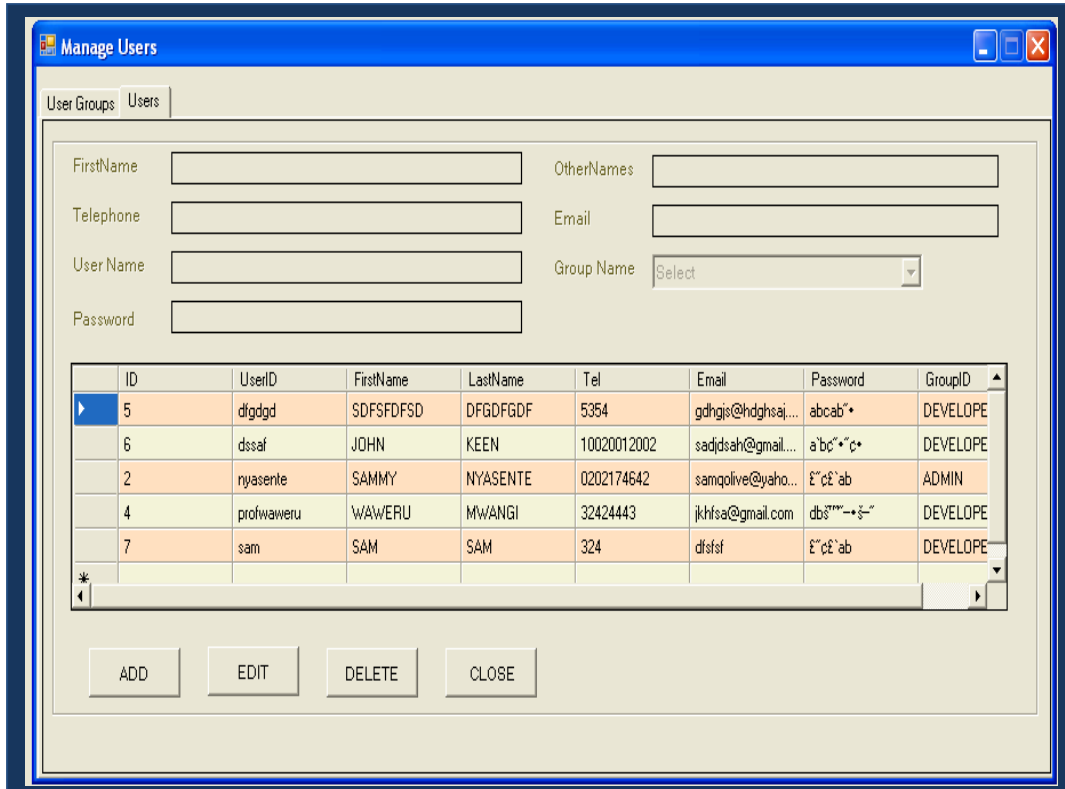
Figure 5.14: The reusability assessment system's interface for managing users

5.11.4. Interface for Managing Metrics

The major tasks that can be performed from this interface include; viewing of reusability details for components that exists in the system's database, editing of components' reusability details, capturing information for a new component's reusability factors, and deleting of records that exist in the system's database. Figure 5.15, shows the screenshot of the said interface.
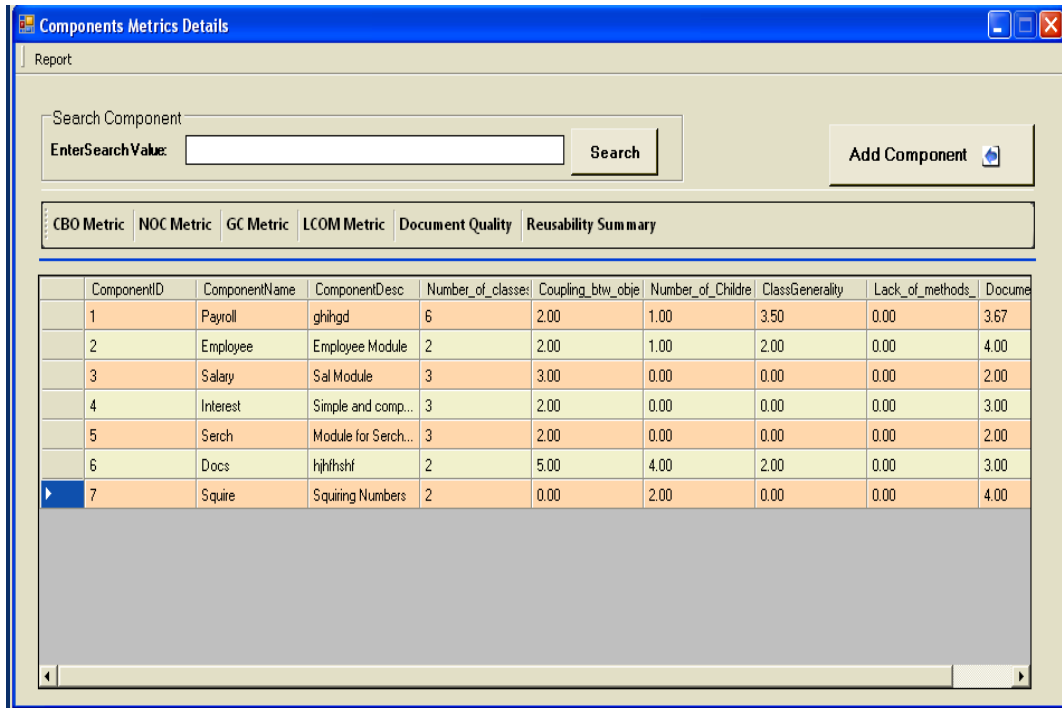
Figure 5.15: The reusability assessment system's interface for managing metrics

5.12.   System Test Conditions and Results

Table 5.12 shows test conditions and results for system testing:-

Table 5.12: System test conditions and test results

S/no	Test	Anticipated Results	Achieved Results
1	User Authentication	The system to display an error message in case of wrong authentication details	The system displayed an error message when a username that does not exist in the system was entered, as well as for wrong passwords.
2	Storing software component reusability details	Storage of component reusability details in a database.	The captured components' reusability details were successfully stored in a database.
3	Storing of user authentication	Storage of system user profiles in a database	The system successfully saved the captured user details into a

	details		database.
4	Editing of stored records	The system should allow a logged on user to modify components' details, as well as allow the administrator to modify user details	Modification of the database by a logged on user was done successfully, and changes written back to the database.
5	Deriving reusability summary	The system should produce a reusability summary for a component based on the provided component's reusability details	The system displays values for maintainability, portability, documentation, generality, understandability as well as the reusability (Rc) for a component
6	System Log off	The system should allow details to be saved and exit	PASS

5.13.   Demonstration of Reusability Measurement Using the System

To demonstrate how the system can be used to calculate the reusability of components, information for the factors that influence the reusability of the sample component (shown in figure 5.4, in section 5.7) is entered into the system. These details are entered by first clicking on the "add Component" button on the user interface for "managing metrics", shown in figure 5.15, above. When this button is clicked, the form shown in figure 5.16 is displayed, where the user is required to enter the component's name, number of classes for the component, and a brief description of the component. The user is required to click on the 'save' button after these values have been entered.

Figure 5.16: Form for adding a new component to the reusability assessment system

The following details are entered for the sample component:

Name of Component: Payroll

Number of classes: 6

Description: Payroll Module

When these values have been entered, they are saved in the system's database by clicking on the "save button", thereafter they are automatically displayed on the form's data Grid. The user is then required to enter information that is required to calculate the CBO, NOC, GC, LCOM, and documentation quality, by clicking the corresponding menu items that are displayed above the data grid. Tables 5.13 through 5.17 give a summary of information that need to be entered into the system in order to calculate each of the metrics for the sample component.

Table 5.13: CBO information for the sample component

Class	Number of couples for the class
Java.lang.Object (1)	1
Employee (2)	4
SalariedEmployee (3)	1
CommissionEmployee (4)	2
HourlyEmployee (5)	1
BasePlusCommissionEmployee (6)	1

Table 5.14: NOC information for the sample component

Class	No. of immediate subclasses subordinated to the class
Java.lang.Object (1)	1
Employee (2)	3
SalariedEmployee (3)	0
CommissionEmployee (4)	1
HourlyEmployee (5)	0
BasePlusCommissionEmployee (6)	0

Table 5.15: GC information for the sample component

Class	Abstraction level
Java.lang.Object (1)	4
Employee (2)	3
SalariedEmployee (3)	2
CommissionEmployee (4)	2
HourlyEmployee (5)	2
BasePlusCommissionEmployee (6)	1
Number of Abstraction levels for the Component's class Hierarchy: **4**	

Number of Abstraction levels for the Component's class Hierarchy: **4

Table 5.16: LCOM information for the sample component

Class	No. of Disjoint Method pairs	No. of non-disjoint method pairs
Java.lang.Object (1)	-	-
Employee (2)	12	16
SalariedEmployee (3)	0	10
CommissionEmployee (4)	4	17
HourlyEmployee (5)	4	17
BasePlusCommissionEmployee (6)	0	10

Table 5.17: Documentation information for the sample component

Factor	Value	
Document structure	5	
Document standards	5	
Writing style	5	
	Provided	**Not Provided**
System Specification	✓	
An architectural design document		✓
The program source code	✓	
	Available	**Not Available**
Availability of Legal terms and conditions	✓	

When each of the required values have been entered, the metrics are calculated and saved into the system's database, and thereafter displayed on the form's data grid. A screenshot for the system data grid is displayed below.

Figure 5.17: Data grid for displaying components' metrics values

A summary on the reusability of a component can then be viewed by first selecting that component (by clicking on the component's ID) on the data grid, and then clicking on the 'Reusability Summary' menu. Figure 5.18 displays a sample form displaying the reusability summary for the sample component.

Figure 5.18: Sample form displaying a component's reusability summary

5.14. Comparison of the Developed Framework with Other Frameworks

In this section, the developed framework (referred to as the new framework) is compared with other existing frameworks—with the aim of validating its superiority over them.

5.14.1. The New Framework vs. the Basic Reusability Attributes Model

The basic reusability attributes model by (Caldiera & Basili, 1991) consists of three reusability attributes: reuse costs, functional usefulness, and quality of components. These attributes are indirectly measured by measuring factors that influence them using four traditional metrics. According to Caldiera and Basili (1991) The three reusability attributes, can only be assessed if the component already exists. Therefore, the model lacks predictive power and cannot be used to assess reusability when developing components for reuse. Secondly, the framework consists of traditional metrics, which cannot be applied to OO software, thus it is not suitable for assessing the reusability of OO software.

In contrast, the reusability attributes in the new framework can be assessed at early stages of software development, thus the framework has predictive power, and can be used to assess reusability when developing components for reuse. Secondly the new framework consists of metrics that measure OO principal structures, making it suitable for measuring reusability of OO components—what the basic reusability attributes model cannot do.

5.14.2. The New Framework vs. the Black-box Component Reusability Model

The black-box component reusability model by (Washizaki et al., 2003), consists of three reusability attributes, namely; understandability, adaptability and portability. The authors relate these attributes with four factors that can be directly measured using five objective metrics. One of the strength of this framework is that, it can be used to assess

reusability in scenarios where source code is not provided (Washizaki et al., 2003). However, the framework consists of metrics that can only be applied to JavaBeans components. That is, it is tailored to the JavaBeans architecture. Secondly, the framework lack predictive power as it is used to assess reusability when developing *with* reuse (Washizaki et al., 2003). The new framework on the other hand uses metrics that are not tailored to any platform—hence it is platform independent. In addition, the framework has predictive power and can also be used at any stage of development—unlike the black-box component reusability model.

5.14.3. The New Framework vs. the Reusability Framework for Ad-Hoc Reuse

The reusability framework for ad-hoc reuse by Hristov et al. (2012) consists of eight attributes that should be considered when assessing reusability of components in ad-hoc reuse scenarios only. That is, it cannot be used to assess reusability when developing components in planned reuse scenarios. In addition, the factors that influence the reusability attributes are measured using traditional metrics, which cannot be applied to OO software. In contrast, the new framework can be used at any stage of OO software development, making it capable of assessing reusability in both planned and ad-hoc reuse scenarios.

5.14.4. The New Framework vs. the Reusable Software Components Framework

The reusable component framework by AL-Badareen et al. (2010) provides a structured criterion for evaluating reusable components when adopting them for reuse. The framework consists of four reusability attributes—with some attributes having factors that influence them. Notably, the framework does not include metrics for measuring the reusability attributes. Instead, the authors provide a criteria for determining if an attribute is present in a component—through conducting tests on the component. This points out to two fundamental weaknesses of the framework. First, the framework has no predictive power, because any test on a component requires that the component be in

101

existence. Secondly, the authors do not provide a way of determining the degree to which a certain attribute is present in a component. In contrast, the new framework uses objective metrics in assessing the reusability attributes, and the degree to which a given attribute is present in a component can be determined, providing an opportunity for improvement.

5.14.5. The New Framework vs. Industry Reusability Assessment Methods

One of the objectives of this research was to determine the methodologies used in industry to assess reusability. This objective was attained through a survey that involved OO software developers. Analysis of the collected data revealed that OO developers do not use metrics in assessing components' reusability. The respondents indicated that they use the following methods in assessing reusability: observing/checking source code, reading documentation, intuition, and checking inline comments. According to Nyasente et al. (2014c),such methods are subjective and cannot be relied on in assessing reusability—as there is no real way of ascertaining the degree of reusability in a given component if they are used. In contrast, the new framework provides for an objective way of assessing reusability, as it uses objective metrics in assessing reusability, thus the degree of reusability can be ascertained.

5.15. Conclusion

This chapter has presented a novel metrics-based framework for assessing the reusability of OO components, as well as its implementation. Literature was analyzed and the key elements for the framework were identified. The elements include; major reusability attributes; reusability factors, OO design constructs and OO metrics. The relationship between these elements is described, and, a reusability attributes model is presented. The last step in developing the framework was to define a reusability equation (equation 5.2)—which the developed system automates. A demonstration of how the framework can be used to assess reusability is also presented and the interpretation of the obtained

reusability value is subsequently given. Finally, a comparison between the new framework and; existing frameworks and other reusability assessment methods is presented—in order to validate the superiority of the new framework.

CHAPTER SIX
SUMMARY, CONCLUSIONS AND RECOMMENDATIONS

6.1. Summary

This research investigated and attempted to address reusability assessment challenges that impede OO developers from effectively reusing existing components in creating new applications. The overriding goal of the research was to identify shortcomings of existing methods of reusability assessment—hence develop and implement an effective framework for OO reusability assessment. To achieve this goal, empirical and theoretical literature analysis was conducted, as well as a survey that involved OO software developers. Literature analysis was conducted in order to identify the characteristics and key elements of an effective reusability assessment framework. This information played a key role in identifying limitations of the reusability assessment frameworks that were found in literature, as well as in the development of a novel framework that had the identified characteristic. The survey on the other hand, was conducted in order to gain an in-depth understanding on reuse and reusability, establish the state of affairs concerning the reuse practice, and to identify methods used in reusability assessment, and the shortcomings of these methods.

6.2. Achievements

This section outlines the achievements attained in relation to the objectives of the study. Generally, the study was aimed at improving reusability of OO software components by addressing issues associated reusability measurement. The first objective of the study was to identify and examine the strategies and methods that software developers use in assessing the reusability of OO components. By conducting a survey, a number of methods were identified. The methods include observing/checking source code, reading documentation, intuition, and checking inline comments.

The second objective was focused on indentifying the shortcomings of the reusability assessment strategies/methods that are currently used. This objective was achieved by comparing methods that exist in industry with what literature describes as ideal reusability measurement techniques. It was established that the methods used in industry were highly subjective; hence unreliable in assessing reusability.

The third specific objective was to determine major attributes that influence reusability; hence develop and implement a framework for assessing reusability of OO components—based on these attributes. The study achieved this objective through literature analysis, and insights gained through a survey. Literature analysis was used to identify major reusability attributes, and other elements that are related to the reusability attributes, which formed part of the framework. The framework was developed by relating the identified reusability elements, and a reusability equation was subsequently defined. The framework implementation was actually an automation of the reusability equation.

The last objective was to test the working of the framework and show that it is superior to the existing methods of measuring reusability. This objective was achieved through experimentation—where the framework was used to measure the reusability of a sample OO component. Lastly, a comparison between the new framework and the ones existing in literature and in industry was conducted for the purpose of validating the superiority of the new framework.

6.3.   Conclusion

Literature shows that software reuse is a popular way of improving software quality, cost of software development, and productivity of programmers. However, reuse is lacking adoption by practitioners. This claim was validated by findings of a survey that involved OO software developers. A review of literature indicated that this was due to

reusability related issues—that can only be addressed through measurement. Therefore, the lack of adoption of reuse by practitioners can be attributed to lack of proper methods for measuring reusability. This informed the formulation of the first and second research questions, which sought to find out methods that are currently used by OO software developers to assess reusability—as well as their shortcoming. This study determined that, the methods used by practitioners were unreliable in assessing reusability—because of their subjectivity.

The third research question sought to find out the attributes that influence reusability because; reusability measurement is achieved by measuring the degree to which some quality attributes, are present in a software component. Five major reusability attributes were identified through comprehensive literature analysis—coupled with some insights gained through the survey findings.

The last research question sought to determine how the reusability attributes can be used to measure the reusability of OO components. This question was addressed through evaluation of how three measurable OO design constructs, namely; inheritance, cohesion and coupling influence the reusability attributes. Literature analysis played a key role in establishing this relationship. In addition, metrics that can be used to measure the reusability attributes were identified from literature. Lastly, a reusability equation for calculating the reusability of OO components was defined and automated.

In general, it can be concluded from this study that reusability assessment can significantly improve the reusability of OO software components—which can subsequently maximize the payoff from OO software reuse.

6.4. Recommendations

As seen from this study, measuring reusability is the only sure way of improving software reuse. However, software measurement is an obscure notion that continues to spur controversy within the software engineering community. This means that, successful software measurement requires commitment from all software engineering stakeholders. This section gives recommendations to different stakeholders, as well as recommendations for improving this study.

6.4.1. Recommendations for Software Development Organizations

The initial costs of establishing a successful reuse program are high, and it requires commitment at all levels. To realize successful reuse Organizations should establish measurement-centric reuse programs, as measurement is the only real way of determining progress in software Engineering. In light of this, organizations should:

- Establish comprehensive software measurement programs that include reusability, as well as establish clear measurement policies.
- Continually encourage and support their staff in embracing measurement as a way of improving different software quality aspects such as reusability.
- Nature the culture of software quality measurement by continually offering training on software metrics and measurement to their staff.

6.4.2. Recommendations for OO practitioners

The following recommendations are given to the OO practitioners in relation to reuse and reusability assessment:

- Adopt the framework that is presented in this study, as it will improve on the reusability of components, which will in turn improve component reuse.
- Adopt Other Software Engineering best design practices that may not have been captured in the framework.

- Integrate other Software development and design tools to aid their software development, as the reusability assessment framework presented in this research does not take the place of such tools.

6.4.3. Recommendations for Improving this Study

This study, has presented a novel reusability measurement framework for OO software that considers three measurable features: inheritance, coupling and cohesion—as the determinants for OO component reusability. The following recommendations on how this research work can be improved are given:

- There is need to study and identify other OO design structures that influence reusability, and include them in the reusability attributes model presented in this research, because; reusability of OO software is not only dependent coupling, cohesion and inheritance—but also on other design features like, polymorphism, information hiding, data abstraction etc.

- The impact of each OO design structure on component reusability should be studied, and then the weights assigned to the reusability attributes should be derived from the impact factor of the design structures that influence them.

- The framework presented in this research should be subjected to rigorous empirical validation, in order to ascertain its efficacy. The outcome of the empirical validation should form a basis for its improvement.

6.4.4. Recommendations for Future Research

The following recommendations are given regarding future research in the field of OO software reuse and reusability assessment:

- More research is required in order to put forward a framework that organizations can adopt in establishing successful measurement programs.

- Researchers should study human/non-technical factors that impede successful reuse and give recommendations on the best ways of overcome them.
- Researchers should work towards an integrated framework for measuring all aspects of OO software quality. In principle, this can achieved by developing a model that relates the key attributes of software quality with measurable OO design constructs, and metrics that can be used to measure these constructs. The possibility and practicability of such a model requires significant amount of research work.

REFERENCES

AL-Badareen, A. B., Selamat, H. M., Jabar, M. A., Din, J., & Turaev, S. (2010). Reusable Software Components Framework. *European Conference of Computer Science (ECCS '10)*, (pp. 126-130). Puerto De La Cruz, Tenerife.

Babu, G. S., & Srivatsa, S. K. (2009). Analysis and Measures of Software Reusability. *International Journal of Reviews in Computing, 1*, 41-46.

Berndtsson, M., Hansson, J., Olsson, B., & Lundell, B. (2008). *Thesis Projects: Aguide for Students in Computer Science and Information Systems* (2nd ed.). London: Springer-Verlag London.

Bradley, J. C., & Millspaugh, A. C. (2009). *Programming in Visual Basic 2008* (7th ed.). New York: McGraw-Hill.

Budhija, N., Singh, B., & Ahuja, P. S. (2013, January). Detection of Reusable Components in object Oriented Programming Using Quality Metrics. *International Journal of Advanced Research in Computer Science and Software Engineering, 3*(1), 351-353.

Caldiera, G., & Basili, V. R. (1991, February). Identifying and Qualifying Reusable Software Components. *IEEE Computer, 24*(2), 61-70.

Chawla, S., & Nath, R. (2013). Evaluating Inheritance and Coupling Metrics. *International Journal of Engineering Trends and Technology (IJETT), 4*(7), 2903-2908.

Chidamber, S. R., & Kemerer, C. F. (1991). Towards a Metrics Suite for Object Oriented Design. *ACM Conference. OOPSLA* (pp. 197-211). Phoenix: ACM.

Chidamber, S. R., & Kemerer, C. F. (1994, June). A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering, 20*(6), 476-493.

Cho, S. E., Kim, M. S., & Kim, D. S. (2001). Component Metrics to Measure Component Quality. *Asia-Pacific Software Engineering Conference (APSEC'01).Eighth.* IEEE.

Daniel, W. W., & Cross, C. L. (2013). *Biostatistics A Foundation for Analysis in the Health Sciences* (10th ed.). New Jersey: John Wiley & Sons.

Deitel, H. M., & Deitel, P. J. (2006). *Java: How to Program* (7th ed.). New Jersey: Prentice Hall.

Deitel, P., & Deitel, H. (2011). *Java: How to Program* (9th ed.). New Jersey: Prentice Hall.

Dubey, S. K., & Rana, A. (2010). A Comprehensive Assessment of Object-Oriented Software Systems Using Metrics Approach. *International Journal on Computer Science and Engineering (IJCSE), 2*(8), 2726-2730.

Farooq, S. U., Quadri, S. M., & Ahmad, N. (2011, January). Software Measurements and Metrics: Role in Effective Software Testing. *International Journal of Engineering Science and Technology (IJEST), 3*(1), 671-680.

Frakes, W. B., & Kang, K. (2005, July). Software Reuse Research: Status and Future. *IEEE Transactions on Software Engineering, 31*(7), 529-536.

Galin, D. (2004). *Software Quality Assurance: From theory to implementation.* Harlow: Pearson Education Limited.

Ghezzi, C., Jazayeri, M., & Mandrioli, D. (2003). *Fundamentals of Software Engineering* (2nd Edition ed.). New Jersey: Prentice-Hall.

Gill, N. S., & Sikka, S. (2011, June). Inheritance Hierarchy Based Reuse & Reusability Metrics in OOSD. *International Journal on Computer Science and Engineering (IJCSE), 3*(6), 2300-2309.

Hristov, D., Hummel, O., Huq, M., & Janjic, W. (2012). Structuring Software Reusability Metrics. *The Seventh International Conference on Software Engineering Advances* (pp. 422-429). IARIA.

IEEE Standard 610.12-1990, IEEE *Standard Glossary of Software Engineering Terminology.*

Ilyas, M., & Abbas, M. (2013, August). Role of Formalism in Software Reusability's Effectiveness. *International Journal of Database Theory and Application, 6*(4), 119-130.

Kan, S. H. (2002). *Metrics and Models in Software Quality Engineering* (2nd ed.). Boston: Addison Wesley.

Kombo, D. K., & Tromp, D. L. (2006). *Proposal and Thesis Writing: An introduction.* Nairobi: Paulines Publications Africa.

Kothari, C. R. (2004). *Research Methodology: Methods and techniques* (2nd ed.). New Delhi: New Age International.

Laird, L. M., & Brennan, M. C. (2006). *Software Measurement and Estimation A Practical Approach.* New Jersey: John Wiley & Sons.

Mahapatra, D. K., Das, T. K., & Pradhan, G. (2012, December). An Integration of JSD, GSS and CASE Tools towards the Improvement of Software Quality. *International Journal of Engineering and Advanced Technology (IJEAT), 2*(2), 306-312.

Mishra, S. K., Kushwaha, D. S., & Misra, A. K. (2009). Creating Reusable Software Component from Object-Oriented Legacy System through Reverse Engineering. *Journal of Object Technology, 8*(5), 133-152.

Návrat, P., & Filkorn, R. (2005). A Note on the Role of Abstraction and Generality in Software Development. *Journal of Computer Science, 1*(1), 98-102.

Nirpal, P. B., & Kale, K. V. (2011, January). A Brief Overview Of Software Testing Metrics. *International Journal on Computer Science and Engineering (IJCSE), 3*(1), 204-211.

Nyasente, S. O., Mwangi, W., & Kimani, S. (2014). A Metrics-based Framework for Measuring the Reusability of Object-Oriented Software Components. *Journal of Information Engineering and Applications, 4*(4), 71-84.

Nyasente, S. O., Mwangi, W., & Kimani, S. (2014). A Note on Complexity and Understandability as Attributes for Assessing the Reusability of Software Components. *IJISET - International Journal of Innovative Science, Engineering & Technology,, 1*(6), 128-131.

Nyasente, S. O., Mwangi, W., & Kimani, S. (2014c). The status of Object-oriented Software Reuse and Reusability Assessment in the Kenyan Software Engineering Industry. *Journal of Information Engineering and Applications, 4*(9), 119-133.

Pressman, R. S. (2005). *Software Engineering: A practitioner's Approach* (6th ed.). New York: McGraw-Hill.

Pressman, R. S. (2010). *Software Engineering: Apractitioner's Approach* (7th ed.). New York: McGraw-Hill.

Prieto-Díaz, R. (1993). Software Reuse: Issues and Experiences. *American Programmer, 6*(8), 10-18.

Rawat, M. S., Mittal, A., & Dubey, S. K. (2012). Survey on Impact of Software Metrics on Software Quality. *International Journal of Advanced Computer Science and Applications (IJACSA), 3*(1), 137-141.

Sametinger, J. (1997). *Software Engineering with Reusable Components.* New York: Springer-
    Verlag.

Sandhu, P. S., Kaur, H., & Singh, A. (2009). Modeling of Reusability of Object Oriented
    Software System. *World Academy of Science, Engineering and Technology, 3*(8), 162-
    165.

Sharma, A., & Dubey, S. K. (2012, June). Comparison of Software Quality Metrics for Object-
    Oriented System. *International Journal of Computer Science & Management Studies
    (IJCSMS), 12*(Special Issue), 12-24.

Singh, G., Singh, D., & Singh, V. (2011, January). A Study of Software Metrics. *International
    Journal of Computational Engineering & Management (IJCEM), 11*, 22-27.

Sommerville, I. (2001).Software documentation. Retrieved from
    ww.literateprogramming.com/documentation.pdf

Sommerville, I. (2011). *Software Engineering* (9th Edition ed.). Boston: Addison-Wesley.

van Vliet, H. (2000). *Software Engineering Principles and Practice* (2nd ed.). New York: Wiley
    and Sons.

Washizaki, H., Yamamoto, H., & Fukazawa, Y. (2003). A Metrics Suite for Measuring
    Reusability of Software Components. *International Software Metrics Symposium*, (pp.
    211-223).

Westfall, L. (2005). 12 Steps to Useful Software Metrics. *The Westfall team*. Retrieved from
    www.westfallteam.com/Papers/12_steps_paper.pdf

APPENDICES

Appendix A: Journal Publications

The following three publications that are related to different aspects of this thesis were made:

i. Nyasente, S. O., Mwangi, W., & Kimani, S. (2014). A Metrics-based Framework for Measuring the Reusability of Object-Oriented Software Components. *Journal of Information Engineering and Applications, 4* (4), 71-84.

ii. Nyasente, S., Mwangi, W., & Kimani, S. (2014). A Note on Complexity and Understandability as Attributes for Assessing the Reusability of Software Components. *IJISET - International Journal of Innovative Science, Engineering & Technology,* 1 (6), 128-131.

iii. Nyasente, S. O., Mwangi, W., & Kimani, S. (2014). The status of Object-oriented Software Reuse and Reusability Assessment in the Kenyan Software Engineering Industry. *Journal of Information Engineering and Applications*, 4 (9), 119-133.

Appendix B: Letter of Introduction

Nyasente Sammy Olive

SCIT, JKUAT

Dear Respondent,

I am a student, at Jomo Kenyatta University of Agriculture and Technology; pursuing a Master of Science Degree (Computer Systems). As a requirement for my Degree program; I am conducting a study on software reusability assessment in Object-Oriented (OO) software, which will culminate in developing of a metrics-based framework for measuring the reusability of OO components.

The interview schedule (attached) aims at collecting information on the current reuse status, reusability assessment, and the challenges to reusability assessment in OO software. The information on the schedule will be kept confidential: names of respondents and institutions they work for will be concealed when compiling the report.

The collected information will go a long way in establishing the challenges facing software reuse and the shortcomings of the current methods employed in reusability assessment. This will form the basis for presenting a more effective way of assessing the reusability of OO software components, with a sole aim of improving software reuse.

Your participation will be highly appreciated.

Yours faithfully

Nyasente Sammy Olive.

Researcher

Appendix C: Data Collection Schedule

PART I

A. Programmer's General Background

1. How many years have you worked as a programmer?

- ○ 1 and under
- ○ 2 - 5
- ○ 6 - 10
- ○ 11 - 15
- ○ 16+

2. Which Object Oriented Programming languages are you conversant with?

- ☐ Java
- ☐ C++
- ☐ C#
- ☐ VB dot Net
- ☐ Delphi
- ☐ Python
- ☐ Others

    (Please specify..)

2. Other than being a programmer, which other software development related skills do you have?

- ☐ OO Analysis and Design
- ☐ Software Engineering
- ☐ System Analysis and Design
- ☐ Others

    (Please specify..)

PART II

A. Reuse and Reusability Issues within the Development Cycle

i: Requirements Modeling and Software Design

1. Do you always reuse requirements documents of existing software when modeling requirements of new related software?

○ Yes

○ No

2. Do you always reuse design of existing software when developing new software?

○ Yes

○ No

3. Do you use CASE tools during requirements modeling and analysis?

○ Yes

○ No

4. Do you often use computerized support in system and component (class) design?

○ Yes

○ No

5. Do you always follow any Cohesion and Coupling criteria when conducting system/ class design?

○ Yes

○ No

6. Do you always control the inheritance hierarchy during class design?

○ Yes

○ No

ii: Coding, Testing, and Maintenance

1. Do you often use code generators to translate design into code?

117

○ Yes

○ No

2. Do you often reuse code from existing software as part of new software code?

○ Yes

○ No

3. If your answer in (2) is Yes, what are the most significant challenges that you experience when reusing code?…………………………………………………………

....

………………………………………………………………………………………….

4. Do you often experience any challenges when testing and maintaining software?

○ Yes

○ No

5. If your answer in (4) is Yes, what are the most significant challenges that you experience when testing and maintaining software? …………………………..

....

…………………………………………………………………………………………

PART III

A. Reuse Practice in the Organization(s)

1. Does your organization have software Reuse Policy?

    ○ Yes

    ○ No

2. Do you and your colleagues ever reuse parts of existing software to build new software?

    ○ Yes

    ○ No

3. With respect to software development in your organization, please indicate the extent to which you agree or disagree with the following statements:

SD = Strongly Disagree

D = Disagree

N = Neutral

A = Agree

SA = Strongly Agree

Cases of developing software from scratch have significantly diminished over time.	○ SD	○ D	○ N	○ A	○ SA
The time and effort required to modify available classes within the organization to fit new reuse contexts is often insignificant as compared to creating new classes	○ SD	○ D	○ N	○ A	○ SA
The cost and effort for developing software has significantly diminished over time.	○ SD	○ D	○ N	○ A	○ SA
I prefer developing classes from scratch than reuse classes that are developed by my colleagues	○ SD	○ D	○ N	○ A	○ SA

B. Payoff from Reuse

1. With respect to cost, effort and productivity of software development in your organization; please indicate the extent to which you agree or disagree with the following statements:

SD = Strongly Disagree

D = Disagree

N = Neutral

A = Agree

SA = Strongly Agree

I am satisfied with the time and effort that is always required to, test, deliver and maintain new software to our clients.	○ SD	○ D	○ N	○ A	○ SA
I am satisfied with budget and cost aspects for developing new software applications and their maintenance.	○ SD	○ D	○ N	○ A	○ SA
I am satisfied with the quality of new software applications we develop as an organization.	○ SD	○ D	○ N	○ A	○ SA
I am satisfied with the overall productivity of developers in the organization.	○ SD	○ D	○ N	○ A	○ SA

PART IV

A. Reusability Assessment

1. Do you always ascertain if classes are reusable when developing or reusing them?

   ○ Yes

   ○ No

2. If your answer in (1) is yes, what are the major characteristics that classes should always have before you consider reusing them? ……………………………………..

   ....

   ………………………………………………………………………………………..

3. Do you know how various Object Oriented design features influence the characteristics you listed in (2) above?

   ○ Yes

   ○ No

4. Do you have a way of assessing whether the classes you develop or reuse possess the characteristics you listed in (2) above?

   ○ Yes

   ○ No

5. If your answer in (4) is yes, state the methodologies you often use. ………………....

   …………………………………………………………………………………………...

PART V

A. Software Metrics and Reusability Assessment

1. Does your organization have a software measurement program/policy?

  ○ Yes

  ○ No

2. With respect to your occupation, what is your experience with software metrics?

  ☐ Never heard about them

  ☐ Heard about them but never interested

  ☐ I have knowledge on metrics but never used them

  ☐ I have used metrics before but stopped using them

  ☐ I always use Software metrics

  Please explain your answer (where applicable) …………………………………...

  ....

3. Do you use metrics to measure the reusability of classes when developing _for_ or _with_ reuse?

  ○ Yes

  ○ No

If your answer is no, please explain why …………………………………………………

  ....

  ....