An Effort Prediction Framework for Software Code Quality Measurement Based on Quantifiable Constructs for Object Oriented Design

Prof. Waweru Mwangi
Institute of Computer Science and Information Technology
Jomo Kenyatta University of Agriculture and Technology
Juja, Kenya.

Dr Wafula Joseph
Institute of Computer Science and Information Technology
Jomo Kenyatta University of Agriculture and Technology
Juja, Kenya.

Stephen N. Waweru
Student: Masters in Software Engineering
Institute of Computer Science and Information Technology
JKUAT – Juja, Kenya.

Abstract: As the Object Oriented Technology enters into software organizations, it has created new challenges for the companies which used only Product Metrics as a tool for monitoring, controlling and maintaining the software product. The structural architecture focus of this research paper is to prove that the mechanisms of Object Oriented Design constructs, namely *Inheritance, Encapsulation and Polymorphism* are the keys to foster reuse and achieve easier maintainability and less complex software codes. This research paper proposes an effort prediction automated framework for software code quality measurement; based on quantifiable constructs for object oriented design, the framework measures the effort of maintaining and reusing the three constructs of Object Oriented Design that is; *Encapsulation, Inheritance and Polymorphism.* The adoption of the Object Oriented Design constructs in this paper is to calculatedly produce easy to maintain, reusable, better and cheaper software in the market. This research paper proceeds to automate the proposed framework system that will be able to predict the effort of measuring the constructs of Object Oriented Design. In order to achieve this, we have utilized one predictor which has been extremely studied: software metrics. The final outcome of this paper is an effort prediction automated tool for software code quality assessment, which predicts effort of maintaining and reusing Object Oriented Programming Languages based on the three OOD constructs. The results acquired are beneficial to be used by software developers, software engineers and software project managers for aligning and orienting their design with common industry practices.

Keywords; Object Oriented Design, maintainability, reusability, encapsulation, inheritance, polymorphism

1.0 Introduction

The backbone of any software system is its design. It is the skeleton where the flesh (code) will be supported. A defective skeleton will not allow harmonious growth and will not easily accommodate change without amputations or cumbersome with all kinds of side effects. Because requirements analysis is most times incomplete, we must be able to build software designs which are easily understandable, alterable, testable and preferably stable (with small propagation of modifications). The Object Oriented (OO) constructs includes a set of mechanisms such as *inheritance, encapsulation, polymorphism and message-passing* that are believed to allow the construction of Object Oriented Designs where those features are enforced. It is widely accepted that object oriented development requires a different way of thinking than traditional structured development1 and software projects are shifting to object oriented design. The main advantage of object oriented design is its modularity and reusability. Object oriented metrics are used to measure properties of object oriented designs. Metrics are a means for attaining more accurate estimations of project milestones, and developing a software system that contains minimal faults [1]. Project based metrics keep track of project

maintenance, budgeting etc. Design based metrics describe the complexity, size and robustness of object oriented and keep track of design performance. Compared to structural development, object oriented design is a comparatively new technology. The metrics, which Ire useful for evaluating structural development, may perhaps not affect the design using OO language. As for example, the "Lines of Code" metric is used in structural development whereas it is not so much used in object oriented design. Very few existing metrics (so called traditional metrics) can measure object oriented design properly. As discussed by Tang [8], claim that "metrics such as Line of Code used on conventional source code are generally criticized for being without solid theoretical basis". One study estimated corrective maintenance cost saving of 42% by using object oriented metrics [9]. There are many object oriented metrics models available and several authors have proposed ways to measure object oriented design.

2.0 Software Quality Overview

The word "Quality" has various meanings.

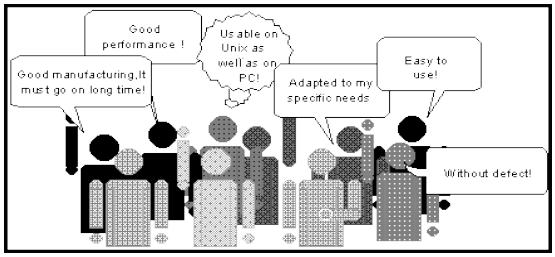


Figure 1. Various quality meanings

The definition given by the ISO/IEC 8402 standard is: *"The totality of features and characteristics of a product or a service that bear on its ability to satisfy stated or implied needs"*. Software quality can not be specified only as software without error. The software quality specification must be more accurate and detailed. The formalization of the software quality can be done using a quality model.

Figure 2. Constructss of Software Quality Systems

2.1 Relations among Software Quality elements

The quality characteristics are used as the targets for validation (external quality) and verification (internal quality) at the various stages of development. They are refined (see Figure 1) into sub-characteristics, until the *attributes or measurable properties* are obtained. In this context, *metric* or *measure* is a defined as a measurement method and *measurement* means to use a metric or measure to assign a value.

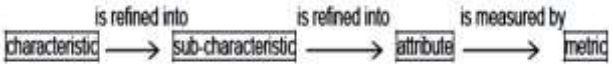


Fig. 3: Relations among the quality model elements

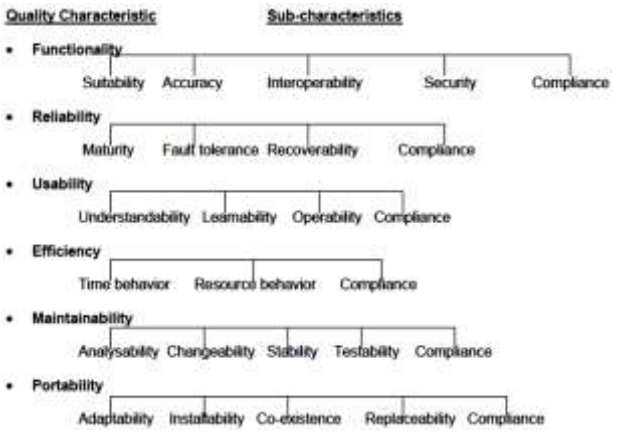


Figure 4. Subcharacteristics of ISO 9126-1 Quality Model

2.3 The Software Quality Model

Our quality model defines a terminology and clarifies the relationships between the reusability, maintainability and the metrics suite. It is a useful tool for guiding software engineers in data interpretation. It was defined based on a set of assumptions. The definition of our quality model is based on: (i) an extensive review of a set of existing quality models [14], (ii) classical definitions of quality attributes and traditional design theories, such as Parnas' theory, which are commonly accepted among researchers and practitioners and (iii) the software attributes impacted by the aspect-oriented abstractions. The quality model has been built and refined using Bluemke's GQM methodology [1].

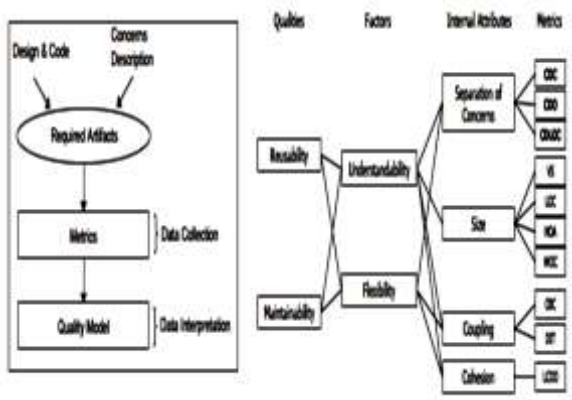


Figure 5. Software quality assessment Framework

Effects of Software Quality in Software Code Complexity

Figure 3 shows some of the elements that software quality consists of. The structural complexity can have a direct impact on how easy the product will be to maintain, because to maintain, one must first understand how the existing code works, then make the required modifications and lastly verify that the changes are correct. The lower the complexity, the more maintainable a system is, and thus it decreases the time needed to fix bugs and speed up the integration/development of new features. Also, the complexity will have an indirect influence on the reliability because the easier it is to test a system the more errors are likely to be discovered before they reach the customer. This will contribute further to the perceived quality of the product

Figure 6: Hierarchy of Software Quality

2.4 Software Quality Characteristic Measures

To the extent possible CISQ measures quantify software some of the Quality Characteristics defined in ISO 25010 which is replacing ISO 9126. ISO 25010 defines a Quality Characteristic as being composed from several quality sub-characteristics. Each quality sub-characteristic consists of a collection of quality attributes that can be quantified as Quality Measure Elements. These Quality Measure Elements can either be counts of structural components or violations of rules of good architectural or coding practice. This specification extends these definitions to the detail required to create measures for each Quality Characteristic that can be computed from statically analyzing the source code. Figure 1 presents an example of Software Quality Characteristic measurement framework suggested in ISO/IEC 25010 and ISO/IEC 15939 using a partial decomposition for Maintainability.

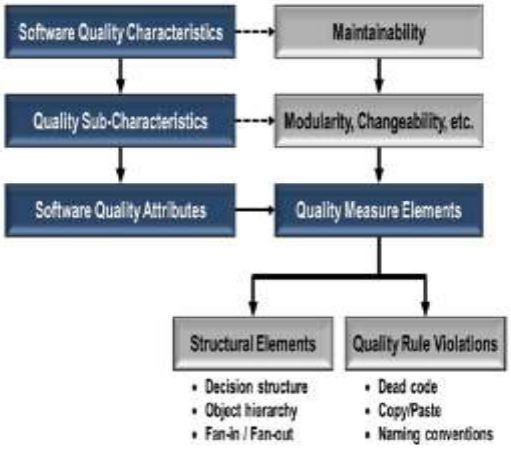


Figure 7. ISO/IEC 25010 & 15939 Framework for Software Quality Characteristics Measurement

Items in the blue boxes in Figure 1 represent the elements of the measurement framework in ISO/IEC 25020 and 15939. Items in the gray boxes are the example instantiations of these framework elements for Quality Characteristic of Maintainability. In particular, the Software Quality Attributes of ISO/IEC 25010 correspond to the Quality Measure Elements in ISO/IEC 15939. Throughout this specification we will refer to the countable structural measure elements and Quality Rule violations as Quality Measure Elements. Scores for individual Quality Measure Elements are summed to create the measure for a Quality Characteristic.

2.5 Need for Software Quality Measurement

Amongst others, software measurement assists software designers in the software development process by enabling them to resolve such issues as estimating:

i. Size of the software product early on in the design phase

ii. Time and effort needed to develop a software product
iii. Error-proneness of the intended software and potential error hot spots
iv. Resources needed for an effective development process
v. Level of maintainability of software product from early design documents
vi. Complexity of the developed code even when developer has not yet started writing any Code
vii. Testability of software by quantifying structural design diagrams

2.6 Software Quality Attributes

Software quality attributes and proposed as a set of eleven design properties in the Figure 4. show a design property definition that are Design Size, Hierarchies, Abstraction, Encapsulation, Coupling, Cohesion, Composition, Inheritance, Polymorphism, Messaging, Complexity and a mathematical formulas in the Table 2, show a design metrics for maintainability estimation model.

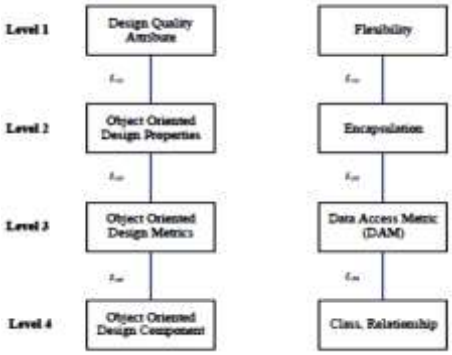


Figure 8. The Quantification Process of the Maintainability Estimation Model.

The above figure describes the quantification process of the maintainability estimation model. The flexibility and extensibility calculate from Computation Formulas for Quality Attribute. Figure 2 reflects the structure of maintainability estimation model

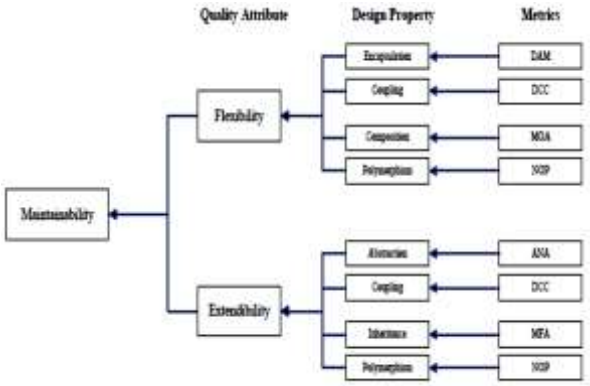


Figure 9. The structure of the maintainability Estimation model

3.0 Software Maintainability and Reusability overview

Despite the subjectivity of any attempt to measure maintainability, great effort has been put into constructing formulas for describing maintainability. Following the opinion that maintainability "is the set of attributes that bear on the effort needed to make specified modifications" [16], we describe maintainability according to this approach as a function of directly measurable attributes *A*1 through *An*, that is:

$$M = f(A1, A2, \ldots, An) \quad \textbf{(1)}$$

On an informal level, this approach is quite appealing it is intuitive that a maintainable system must be e.g. consistent and simple. However, there may be great difficulties in measuring those attributes and weighting them against each other and combine them in a function *f*. Any such attempt is therefore bound to a quite limited context a particular programming language, organization, type of system, type of project; the skill and knowledge of the people involved must also be considered then drawing conclusions. The maintainability is a quality factor with influence in the software maintenance phase. Many researchers reported that 50-70% of the total life cycle is spent on software maintenance phase can provided earlier feedback to help a software designer improved the quality of software systems and reduced the increasing high cost of software maintenance phase. The maintainability is defined by IEEE standard glossary of Software Engineering 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". Software maintenance

includes all post implementation changes made to software entity.

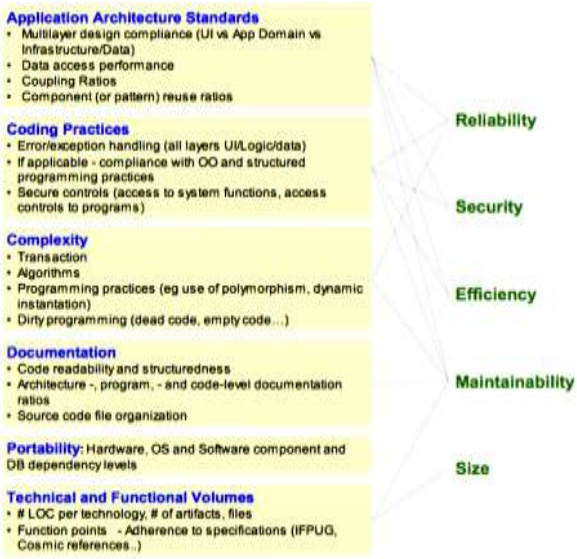


Figure 3. Relationship between software desirable characteristics (right) and measurable attributes (left) [15]

Maintainability refers to the easiness or toughness of the required efforts to do the changes. Before any changes can be made to a software entity, the software must be fully understood. After the changes have been completed, the revised entity must be thoroughly tested as well. For this reason, maintainability can be thought of as three attributes: understandability, modifiability, and testability. Harrison sees software complexity as the primary factor affecting these four attributes [17], *Abstraction, Encapsulation, Inheritance, and Polymorphism* which are closely related to the software code complexity (See Figure 4).

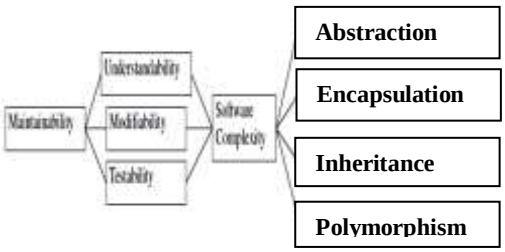


Figure 10. Harrisons Maintainability Model

In the same application, the time required per module to determine the changes indicates understandability; the time to change indicates modifiability; the time to test indicates testability. Instead of collecting the measurement after the product is completed, our approach is to *forecast* the maintainability based on the source code and display the measurement at any time the programmer wishes. The source code can be at any stage of the development, and the measurement will be computed automatically. This will provide a real time *grade* of the software in the dimension of maintainability.

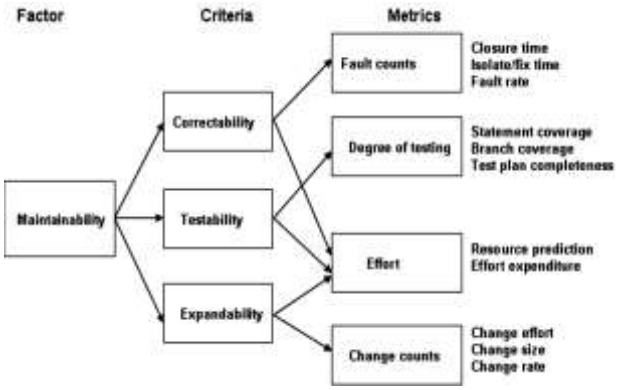


Figure 7. Fenton's decomposition of the maintainability [21]

4.0 Research Concept

4.1 Research Problem Statement

Increasingly, object-oriented measurements are being used to evaluate and predict the quality of software [18]. A growing body of empirical results supports the theoretical validity of these metrics [3]. The validation of these metrics requires convincingly demonstrating that the metric measures what it purports to measure and the metric is associated with an important external metric, such as reliability, maintainability and fault-proneness [4]. Often these metrics have been used as an early indicator of these externally visible attributes, because the externally visible attributes could not be measures until too late in the software development process. Object oriented metrics evaluate the object oriented concept: methods, classes, cohesion, coupling and inheritance. Object oriented metrics focus on the internal object structure. Object oriented metrics measure externally the interaction among the entities. Object oriented metrics measures the efficiency of an algorithm.

4.2 Research Paper Objective

To experiment and validate a set of metrics suitable for evaluating the use of the mechanisms that support the main concepts of the Object-Oriented constructs and the consequent emphasis on maintainability and reusability, that are believed to be responsible for the increase in software quality and development productivity. To propose, evaluate and implement an effective and unique software

maintainability and reusability metrics based on *Inheritance, Encapsulation and Polymorphism* in object oriented systems so that it can be used by the industry as well as academia. To automate, implement the parameter and associated aspects based on the effort of maintaining and reusing software code and develop the parser so that it can automatically calculate the maintainability and reusability effort in the level of *Inheritance, Encapsulation and Polymorphism.*

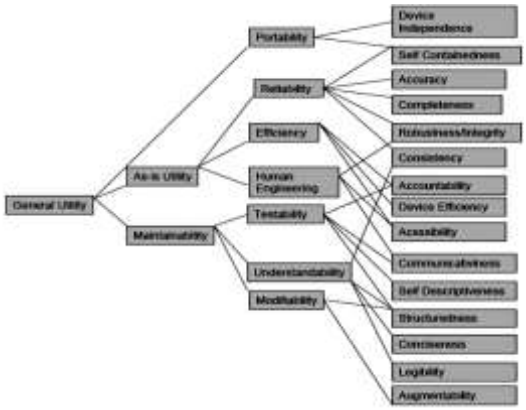
4.3 Research Motivation

However, the last decade showed that even using the object oriented constructs, coping with very large software systems is a hard task: Very large software systems can become several millions of lines of code long, with many different people having taken part on its implementation lasting into months or years. Many problems can affect those systems, naming but a few: The original developers left and there is nobody who fully understands the original design decisions. Missing sparse or erroneous documentation. Obsolete programming tools, platform migrations and outdated hardware make it hard to find people knowing such techniques or willing to deal with such problems. A good example in this case is the so-called millennium bug, also known as the Y2K - problem, where suddenly a huge number of experts was needed to test software written in languages which are no longer used nowadays. Maintenance is often done by less experienced programmers which have to face not only the problem of complexity but also the problem of dealing with code from unknown areas. In fact, experienced programmers which often tend to move on to other projects and areas of interest, take also a great deal of domain specific knowledge with them which the maintainers sometimes lack. Several design errors have made the evolution of the system nearly impossible: small changes can affect large parts of the system. There is duplicated code everywhere, which means the programmers used to copy and paste often. Duplicated code can cause code bloat, error propagation and decrease flexibility (a change has to be done in many places). Even with all those points speaking for a reprogramming from scratch of the system, there is one main point speaking against it: *The system is working.* Maintenance of such systems is thus the only possible approach states that maintenance, in its widest sense of 'post deployment software support', is likely to continue to represent a very large fraction of total system cost. Rebuilding the system from scratch would mean months or years of development, but with the ongoing technology race such a long delay can mean financial ruin.

5.0 Literature Review

A number of software metrics have been defined in literature yet not all of them have been proved to be enough significant. Therefore, some fundamental principles and characteristics have to be considered while defining new software metric. Zuse [19] provide a comprehensive overview of different metrics properties proposed by researchers. Conte et al. [20] suggest that a metric should be objective; that is, the value must be computed in a precise manner. He proceeds to say that a metric must be at least valid, reliable and practical. It must also be intuitive (have face validity) and possess some form of internal validity.

Figure 11. Boehm's Quality model (1978)



Several maintainability models/methodologies were proposed to help the designers in calculating the maintainability of software so as to develop better and improved software systems. Starting from 1970s to 2012 various maintainability predicting models or techniques were developed.

26	2009	MO Elish and KO Elish	Treenet model using stochastic gradient boosting
27	2010	Ruchi khan Model(MEMOOD)	LOC (Line of Code), MI(), NC (Number of Class),NA (Number of Attributes),NM (Number of Methods), Class Diagram
28	2010	L Ping	Hidden Markov Model
29	2010	C Jin, JA Liu	Support vector machine
30	2011	Gautam Kaur Model (COMPOUND MEMOOD MODEL)	LOC (Line Of Code), DLOC (Difference Line of Code),MI (Maintainability Index), CC (Cyclomatic Complexity)
31	2012	Malhotra- Chug Model	WMC, DIT, NOC, RFC,LCOM, MPC, DAC, NOM,SIZE1, SIZE1, CHANGE
32	2012	Alkaaru Hindewmaan, Wanchai Rivepiboon	design size, hierarchies, abstraction, encapsulation, coupling, cohesion etc using Maintainability Estimation Tool (MET)
33	2012	Dubey et. al Model	DIT, NOC, MPC, RFC, LCOM, DAC, WMC, NOM, SIZE1 and SIZE2 using Multi Layer Perception (MLP) neural network model

Table 1. Maintainability models developed between 2009 - 2012

5.1 The Current Maintainability Model

The model was originally developed as an improvement over the classic Maintainability Index of Paul Oman and others. In collaboration with Delft University of Technology and the University of Amsterdam, several empirical studies have been conducted on the model.

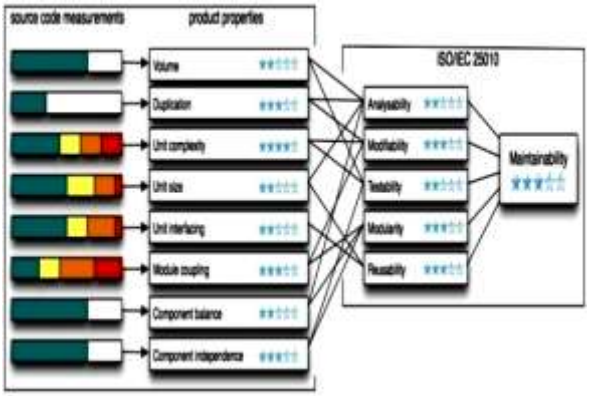


Figure 12. Maintainability index of Paul Oman

These studies validate the strong predictive effect that the model has for the efficiency of development and maintenance tasks, such as resolution of defects and implementation of enhancements. The Maintainability Model is used by SIG to determine software product quality in software risk assessments and during software monitoring..

5.2 Taxonomy of Metrics

Software metric is the measurement, usually using numerical ratings, to quantify some characteristics or attributes of a software entity. Typical measurements include the complexity and readability of the source codes, the length and quality of the development process and the efficiency and performance of the accomplished applications. Some major measurements are listed in table 3.

Role	Measurements
User	Usability, Simplicity, Stability, Cost...
Designer	Extendibility, Scalability, Manageability...
Programmer	Complexity, Maintainability...

Table 2 Different Measurements in terms of Different roles

Software Engineering Metrics

Metrics are units of measurement. The term "metrics" is also frequently used to mean a set of specific measurements taken on a particular item or process. Software engineering metrics are units of measurement that are used to characterize:

i. Software engineering products, e.g., designs, source code, and test cases.
ii. Software engineering processes, e.g., the activities of analysis, designing, and coding.
iii. Software engineering people, e.g., the efficiency of an individual tester, or the productivity of an individual designer.

5.3 Taxonomy of Object Oriented Metrics

Different writers have described different metrics, according to object oriented design, there are some similarities found in different metrics model. The following table shows similar OO metrics. We have categorized metrics in class, attribute, method, cohesion, coupling, and inheritance category because most of the object oriented metrics are defined in above mention categories.

Table 6 . Similarity of Object Oriented Design Metrics

5.4 Measurable OOD Constructs

Category	Class	Attribute	Method	Cohesion	Inheritance
(C&K) [6]	WMC,RFC, LCOM	LCOM	WMC,RFC, LCOM	CBO	DIT,NOC
Chen & Lu [7]	OXM,RM, OACM			CCM, OCM	CHM
Li & Henry [8]	DAC,MPC, NOM	NOM		MPMPC	NOM

Table 3. Object Oriented Metrics

The design methods provide a set of techniques for analyzing, decomposing, and modularizing software system architectures. There is wide applicability of object-oriented design in today's scenario of software development environment because it promotes better design and view a software system as a set of interacting objects. Object-oriented design must exhibit four features: inheritance, data abstraction, dynamic binding, and information hiding.

Objects	Build	Classes
Objects	Have (are composed of)	Attributes
Objects	Inherit	Attributes
Objects	Have (are composed of)	Methods
Objects	Inherit	Methods
Objects	Send	Messages
Objects	Receive	Messages
Messages	Are	Data
Messages	Are	Relations

Figure 4. Components of object-oriented software [8]

5.5 Object Oriented Design Quantifiable Characteristics

A set of components which can help measure, analyze, represent and implement an object-oriented design should include attributes, methods, objects (classes), relationships, and class hierarchies. The diagram in Figure 3 illustrates the mapping of quality carrying component properties to design properties. It also shows the assigning of design metrics to design properties. Finally, it presents the linking between design properties to quality attributes. Some of the design properties have positive influence on the quality attributes while on other quality attributes, they could have negative influence.

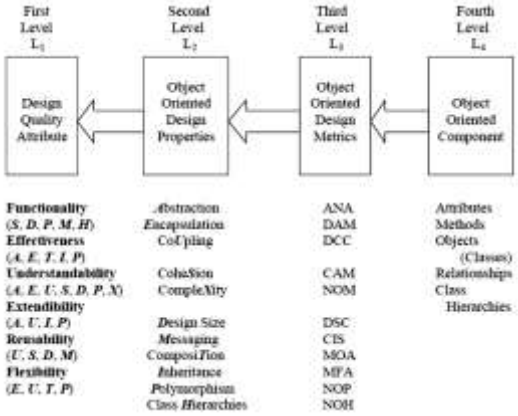


Figure 13. Bansiya, J. and Davis, C. G. "A Hierarchical Model for Object-Oriented Design Quality Assessment," IEEE Transactions on Software Engineering, vol. 28, no. 1, pp. 4-17, 2002.

Design Property	Definition
Design Size	A measure of the number of classes used in a design.
Hierarchies	Hierarchies are used to represent different generalization-specialization concepts in a design. It is a count of the number of non-inherited classes that have children in a design.
Abstraction	A measure of the generalization-specialization aspect of the design. Classes in a design which have one or more descendants exhibit this property of abstraction.
Encapsulation	Defined as the enclosing of data and behavior within a single construct. In object-oriented designs the property specifically refers to designing classes that prevent access to attribute declarations by defining them to be private, thus protecting the internal representation of the objects.
Coupling	Defines the interdependency of an object on other objects in a design. It is a measure of the number of other objects that would have to be accessed by an object in order for that object to function correctly.
Cohesion	Assesses the relatedness of methods and attributes in a class. Strong overlap in the method parameters and attribute types is an indication of strong cohesion.
Composition	Measures the "part-of," "has," "consists-of," or "part-whole" relationships, which are aggregation relationships in an object-oriented design.
Inheritance	A measure of the "is-a" relationship between classes. This relationship is related to the level of nesting of classes in an inheritance hierarchy.
Polymorphism	The ability to substitute objects whose interfaces match for one another at run-time. It is a measure of services that are dynamically determined at run-time in an object.
Messaging	A count of the number of public methods that are available as services to other classes. This is a measure of the services that a class provides.
Complexity	A measure of the degree of difficulty in understanding and comprehending the internal and external structure of classes and their relationships.

Figure 5 Quantifiable (measurable) characteristics in OOD

6.0 Proposed Framework

The proposed framework provide a medium through which software engineers can build robust, easy to maintain, relatively cheaper and more importantly provide a quantitative way of measuring the success of a project. This Paper main objective is to achieve better software code maintainability. Maintenance is the major resource waster in the whole software life-cycle. Maintainability can be evaluated through several quality sub characteristics like *analyzability*, *changeability*, *stability* and *testabilility* [ISO9126]. The software metrics are widely advocated as fundamental elements of an engineering approach to planning and controlling OO software development. The Proposed Framework refers to the basic structural mechanism of the *Object Oriented Constructs*. As a consequence the proposed framework metrics ranges from 0 (no use) to 1 (maximum use). Being formally defined, the proposed frameworks avoid subjectivity of measurement and thus allow replicability. In other words, different people at different times or places can yield the same values when measuring the same systems. The Framework makes no reference to specific language constructs that allow for implementation of OO mechanisms in more or less detail. A mapping concepts called binding between the proposed Framework and the *Adopted Language* is required. The proposed framework is also expected to be a system size independent. Size independence allows inter-project comparison, thus fostering cumulative knowledge. The Proposed Framework measure THREE main structural mechanism of *Object Oriented Design*;

The First subset describes how much the design *Hides Method and Attributes Internally* within implementation details.

a) **Maintainability Effort for Attributes [MEA]**
b) **Maintainability Effort for Methods [MEM]**

The Second subset describes class *Hierarchy and Reusability.*

a) Reusability Effort for Attributes [REA]
b) Reusability Effort for Methods [REM]

The Third subset measures the *Degree of Method Overriding* in the class inheritance structure

a) **Method Overriding Effort (MOE)**

6.1 Definition 1 - MEA

MEA measures how attributes are encapsulated in a class. Visibility is counted in respect to other classes. MEA represent the average amount of hiding among all classes in the system. A private method/attribute is fully hidden

$$MEA = \frac{SO_{Pri[a]} + SO_{Pro[a]}}{SO_{Pri[a]} + SO_{Pro[a]} + SO_{Pub[a]}}$$

where;

$SO_{Pri[a]}$: *Sum of Private Attributes* $= \sum Pri_A (C_i)$

$SO_{Pro[a]}$: *Sum of Protected Attributes* $= \sum Pro_A (C_i)$

$SO_{Pub[a]}$: *Sum of Public Attributes* $= \sum Pub_A (C_i)$

therefore;

$$MEA = \frac{\sum Pri_A (C_i) + \sum Pro_A (C_i)}{\sum Pri_A (C_i) + \sum Pro_A (C_i) + \sum Pub_A (C_i)}$$

that is;

$$\textbf{MEA} = \frac{\sum_{i=1}^{C} Pri_A(C_i) + \sum_{i=1}^{C} Pro_A(C_i)}{\sum_{i=1}^{C} Pri_A(C_i) + \sum_{i=1}^{C} Pro_A(C_i) + \sum_{i=1}^{C} Pub_A(C_i)}$$

where;

$Pri_A (C_i)$ *: Sum of Private Attributes in a class*

$Pro_A (C_i)$ *: Sum of Protected Attributes in a class*

$Pub_A (C_i)$ *: Sum of Public Attributes in a class*

C_i *: Total number of Classes*

6.2 Definition 2 - MEM

MEM measures how methods are encapsulated in a class. Visibility is counted in respect to other classes. MEM represent the average amount of hiding among all classes in the system. A private method/attribute is fully hidden

$$MEM = \frac{SO_{Pri[m]} + SO_{Pro[m]}}{SO_{Pri[m]} + SO_{Pro[m]} + SO_{Pub[m]}}$$

where;

$SO_{Pri[m]}$: *Sum of Private Methods* $= \sum Pri_M (C_i)$

$SO_{Pro[m]}$: *Sum of Protected Methods* $= \sum Pro_M (C_i)$

$SO_{Pub[m]}$: *Sum of Public Methods* $= \sum Pub_M (C_i)$

therefore;

$$MEM = \frac{\sum Pri_M (C_i) + \sum Pro_M (C_i)}{\sum Pri_M (C_i) + \sum Pro_M (C_i) + \sum Pub_M (C_i)}$$

that is;

$$\textbf{MEM} = \frac{\sum_{i=1}^{C} Pri_M(C_i) + \sum_{i=1}^{C} Pro_M(C_i)}{\sum_{i=1}^{C} Pri_M(C_i) + \sum_{i=1}^{C} Pro_M(C_i) + \sum_{i=1}^{C} Pub_M(C_i)}$$

where;

$Pri_M (C_i)$ *: Sum of Private Methods in a class*

$Pro_M (C_i)$ *: Sum of Protected Methods in a class*

$Pub_M (C_i)$ *: Sum of Public Methods in a class*

C *: Total number of Classes*

6.3 Framework *Definition 1 & 2 Overview*

The MEA numerator is the sum of the invisibilities of all attributes defined in all classes. The invisibility of a attributes is the percentage of the total classes from which this attribute is not visible. The MEA denominator is the total number of attributes defined in the system under consideration. The MEM numerator is the sum of the invisibilities of all methods defined in all classes. The invisibility of a method is the percentage of the total classes from which this method is not visible. The MEM

denominator is the total number of methods defined in the system under consideration.

6.4 Definition 3 - REM

REM is a measure of a class methods *inherited density complexity*.

$$\textbf{REM } (C_i) = \frac{\textbf{NI}_M (C_i)}{\textbf{NN}_M (C_i) + \textbf{NO}_M (C_i) + \textbf{NI}_M (C_i)}$$

where;

REM (C_i): *Reusability Effort for Methods*

$NI_M (C_i)$: *Number of Inherited Methods in a class*

$NN_M (C_i)$: *Number of New Methods in a class*

$NO_M (C_i)$: *Number of Overridden Methods in a class*

Ci : *Total Number of Classes*

$REM(C_i) = \sum NI_M (C_i) / \sum NN_M (C_i) + \sum NO_M (C_i) + \sum NI_M (C_i)$

that is;

$$REM (C_i) = \frac{\sum_{i=1}^{C} NI_m(C_i)}{\sum_{i=1}^{C} NM_M(C_i) + \sum_{i=1}^{C} NO_M(C_i) + \sum_{i=1}^{C} NI_M(C_i)}$$

where;

REM (C_i) : Reusability Effort for Methods

$NI_M (C_i)$: Number of Inherited Methods in a class

$NN_M (C_i)$: Number of New Methods in a class

$NO_M (C_i)$: Number of Overridden Methods in a class

T_c : Total Number of Classes

6.5 Definition 4 - REA

REA is a measure of a class attributes inherited density complexity.

$$\textbf{REA } (C_i) = \frac{\textbf{NI}_A (C_i)}{\textbf{NN}_A (C_i) + \textbf{NO}_A (C_i) + \textbf{NI}_A (C_i)}$$

where;

REA (C_i) : *Reusability Effort for Attributes*

$NI_A (C_i)$: *Number of Inherited Attributes in a class*

$NN_A (C_i)$: *Number of New Attributes in a class*

$NO_A (C_i)$: *Number of Overridden Attributes in a class*

Ci : *Total Number of Classes*

$REA(C_i) = \sum NI_A (C_i) / \sum NN_A(C_i) + \sum NO_A (C_i) + \sum NI_A (C_i)$

that is;

$$REA (C_i) = \frac{\sum_{i=1}^{C} NI_A(C_i)}{\sum_{i=1}^{C} NM_A(C_i) + \sum_{i=1}^{C} NO_A(C_i) + \sum_{i=1}^{C} NI_A(C_i)}$$

where;

REA (C_i): Reusability Effort for Attributes

$NI_A (C_i)$: Number of Inherited Attributes in a class

$NN_A(C_i)$: Number of New Attributes in a class

$NO_A (C_i)$: Number of Overridden Attributes in a class

T_c : Total Number of Classes

6.5 Framework *Definition 3 & 4 Overview*

The REM numerator is the sum of inherited methods in all classes of the system under consideration. The REM denominator is the total number of available methods (new methods declared, overriding methods plus inherited methods) for all classes. The REA numerator is the sum of inherited attributes in all classes of the system under consideration. The REA denominator is the total number of available attributes (new attributes declared, overriding attributes plus inherited attributes) for all classes. A class that inherits lots of methods /attributes from its ancestor classes contributes to a high REM / REA. A child class that redefines its ancestors' methods/attributes and adds new ones contributes to a lower REM/REA. An independent class that does not inherit and has no children contributes to a lower REM/REA. REM & REA should be in a reasonable range, not too low and not too high either. Too high a value

indicates either excessive inheritance or too wide member scopes. A low value indicates lack of inheritance or heavy use of Overrides. Another view is that REA should ideally be zero because all variables should be declared Private. For a class lacking an Inherits statement, REM=0 and REA=0. For a class with no attributes, REA=0. For a class with no methods, REM=0.

A method/attribute is inherited if:

i. *It's defined in the base class*

ii. *It's visible in the child class*

iii. *It's not overridden in the child.*

6.6 Definition 5 – MOE

MOE measures the degree of method overriding in the class inheritance structure

$$MOE = \frac{MO}{NM \times Dc}$$

where;

MO : *Method Overrides*

NM : *New Methods*

Dc : *Descendants*

that is;

$$MOE = \frac{\sum_{i=1}^{C} AM_O(C_i)}{\sum_{i=1}^{C} NM_C(C_i) + \sum_{i=1}^{C} D_C(C_i)}$$

where;

MOE : *Method Overriding Effort*

AM_O : *Actual Method overrides in a Class* (C_i)

NM_C : *Number of New methods in a Class* (C_i)

D_C : *Number of Descendants*

C : *Total Number of Classes*

6.7 Framework *Definition 5 Overview*

MOE measures the degree of method overriding in the class inheritance tree. It equals the number of actual method overrides divided by the maximum number of possible method overrides. A call to an object's method can be statically or dynamically bound to a named method implementation. The latter can have as many shapes as the number of times the method is overridden in that class's descendants. In the formula, the numerator equals the *actual* overrides and the denominator is the *maximum* number of possible overrides. If you always override everything, you get a MOE of 1. If your child classes seldom override their parent's methods, you get a low MOE. If your parent classes declare sealed methods, you will end up with a low MOE. Overrides can be used to a reasonable extent to keep the code clear, but that excessively overrides be too complex to understand (because several alternative methods can execute for one call statement).

7.0 Proposed Framework System Experimentations

To help clarify the *Framework Robustness* and *Applicability*, the following C++ code will be used.

Figure 14. UML Class Diagram Representation for the Entire C++ Code

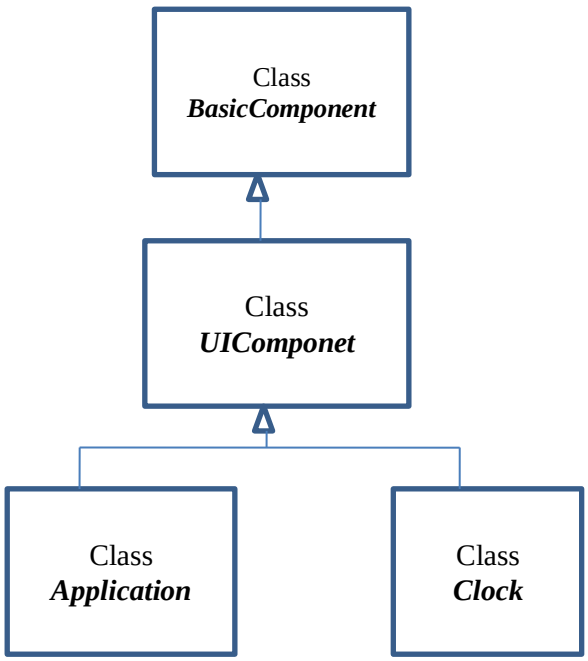
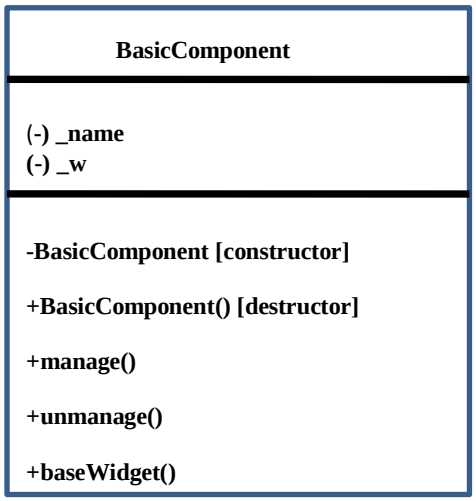


Figure 15. UML Class Diagram for Class *BasicComponent*



i) MEM Computation

p() methods = 1

v() methods = 4

d() methods = *p()* + *v()* = 5

= 1/5

= 0.2

ii) MEA Computation

p() attributes = 2

v() attributes = 0

d() attributes = *p()* + *v()* = 2

= 2/2

= 1

iii) REM Computation

h() methods = 5

o() methods = 0

i() methods = 0

d() methods = (5+0) = 0

a() methods = (5+0) = 5

= 0/5

= 0

iv) REA Computation

h() methods = 2

o() methods = 0

i() methods = 0

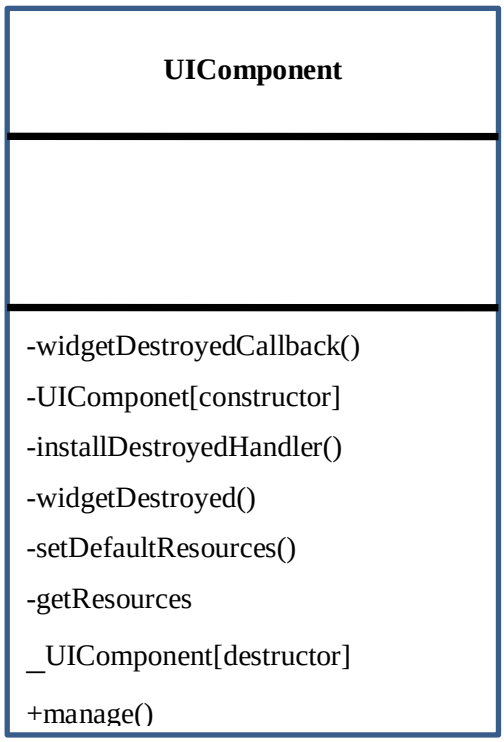
d() methods = (2+0) = 2

a() methods = (2 + 0) = 2

= 0/2

= 0

Figure 16. UML Class Diagram for Class *UIComponent*



i) MEM Computation

p() methods = 6

v() methods = 3

d() methods = *p()* + *v()* = 9

= 6/9

= 0.67

ii) MEA Computation

p() attributes = 0

v()attributes = 0

d() attributes = *p()* + *v()* = 0

= 0/0

= 0

iii) REM Computation

h() methods = 8

o() methods = 1

i() methods = 4

d() methods = (8+1) = 9

a() methods = (9 + 4) = 13

= 4/13

= 0.3

iv) REA Computation

h() methods = 0

o() methods = 0

i() methods = 2

d() methods = (0+0) = 0

a() methods = (0 + 2) = 2

= 2/2

= 1

Figure 17. UML Class Diagram for Class Application

Application
(-) _display **(-) _appContext** **(-) _applicationClass** **(-) _windows** **(-) _numWindows**
-main() -registerWindows() -unregisteredWindows() -initialize() -handleEvents() +Application[constructor] +Application()[destructor] +manage() -unmanage() +iconify() +display() +appContext() +applicationClass +classNam()

i) MEM Computation

p() methods = 5

v() methods = 9

d() methods = *p()* + *v()* = 14

= 5/14

= 0.36

ii) MEA Computation

p() attributes = 5

v() attributes = 0

d() attributes = *p()* + *v()* = 5

= 5/5= 1

iii) REM Computation

h() methods = 11

o() methods = 1

i() methods = 9

d() methods = (11+3) = 14

a() methods = (14 + 9) = 23

= 9/23

= 0.67

iv) REA Computation

h() methods = 5

o() methods = 0

i() methods = 2

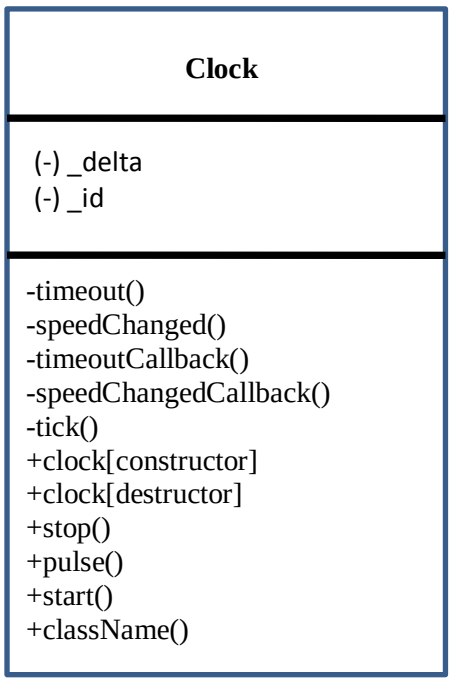
d() methods = (5+0) = 5

a() methods = (5 + 2) = 7

= 2/7

= 0.28

Figure 18. UML Class Diagram for Class Clock



i) MEM Computation

p() methods = 5

v() methods = 6

d() methods = p() + v() = 11

= 5/11

= 0.45

ii) MEA Computation

p() methods = 2

v() methods = 0

d() methods = p() + v() = 2

= 2/2

= 1

iii) REM Computation

h() methods = 10

o() methods = 1

i() methods = 11

d() methods = (10+1) = 11

a() methods = (11 + 11) = 22

= 11/22

= 0.5

iv) REA Computation

h() methods = 2

o() methods = 0

i() methods = 2

d() methods = (2+0) = 2

a() methods = (2 + 2) = 4

= 2/4

= 0.5

Framework Analysis and Derived Conclusions

Class	MEM Results	MEA Results	REM Results	REA Results
Class BasicComponent	0.2	1	0	0
Class UIComponent	0.67	0	0.3	1
Class Application	0.36	1	0.67	0.28
Class Clock	0.45	1	0.5	0.50

Table 6. Summarized analysis computation results for the proposed Framework

The implementation of the class interface should be a stepwise decomposition process, where more and more details are added. This decomposition will use hidden methods, thus obtaining the above-mentioned information-hiding benefits and favouring an MEM increase. A very low MEM value would indicate an insufficiently abstracted implementation. Conversely, a high MEM value would indicate very little functionality. MEM serves as an indicator of the complexity of the class methods. High MEM is an indicator of classes comprised of methods with high complexity. If all methods are private/protected, MEM = 1, High encapsulation decreases the complexity since encapsulated methods dictate the scope from which they may be accessed therefore limiting the number of locations which makes the debugging process easier. If all methods are public, MEM = 0 shows methods are unprotected and chances of errors are high.

Very low values of MEA should trigger the designers attention. In general, as MEA increases, the complexity of the program decreases. MEA measures the total number of attributes encapsulated in the class. The MEA may be expressed as a fraction in which the denominator is the number of total attributes whereas the numerator is the total of encapsulated attributes defined in all the classes. If all attributes are private/protected then MEA = 1 and If all attributes are public then MEA = 0 this shows methods are unprotected and chances of errors are high. Classes with high MEA indicate a higher percentage of methods that require rigorous testing

REM is the inherited methods/total methods available in classes i.e. the ratio of inherited methods to total number methods of available classes. Both REM and REM should be maintained at mediocre ratios since too high a ratio of either indicates excessive inheritance and too low a ratio indicates a poor object-oriented framework. A class with high REM will require more testing effort, as it is easily affected by changes made in other classes. The cause of change in behaviour of this class may be more difficult to trace, as it is not found in the class itself. As a result, classes should not inherit from classes with high REM.

REA is the inherited attributes/total attributes available in classes i.e. the ratio of inherited attributes to the total number of attributes. A class with high REA will require more testing effort, as it is easily affected by changes made in other classes. The cause of change in behaviour of this class may be more difficult to trace, as it is not found in the class itself. As a result, classes should not inherit from classes with high REA.

7.2 The Proposed Framework System Overview

He proposed automated framework Measurement Tool is a software measurement environment to analyze program source code for software reuse and maintenance. It is especially designed for object-oriented software. This tool measures attributes from OOD source code, collects the measured data, computes various object-oriented software metrics, and presents the measurement results in a tabular form. The tabular interface of the tool provides software developers the capabilities of inspecting software systems, and makes it easy for the developers to collect the metric data and to use them for improving software quality. By browsing *reusable units* and *maintainable units*, a developer can learn how to reuse certain software entity and how to locate problematic parts. The application of this easy-to-use tool significantly improves a developer's ability

to identify and analyze quality characteristics of an object-oriented software system. The intended application domain is small, middle and large sized software developed in OOD. The key components of the architecture are: 1) User Interface, 2) JavaCode analyzer, 3) Internal Measurement Tree, 4) Measurement Data Generator, and 5) Measurement Table Generator.

7.3 Proposed System Architecture

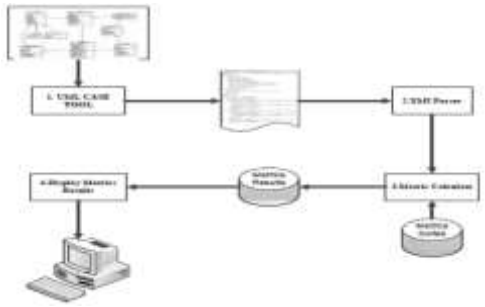


Figure 19 A Structure of Proposed Software Code Quality Assessment Tool

This tool automates the collection of data by parsing and uses these data to calculate the proposed object-oriented designs metrics constructs; that is *Encapsulation, Inheritance and Polymorphism.*

7.4 System Framework Automation Architecture

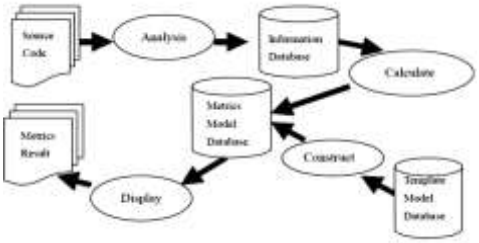


Figure 20. Proposed system architecture

Constructing Phase

Metrics users customize the metrics model based on the implemented metrics.

Analyzing Phase

The analysis front end analyzes source code, extracts program information and stores it into the program information database through the database server.

Calculating Phase

After users select some models in the model database, all the values of the model are calculated from the information database and then are stored into metrics model database.

Tool

The model database is used to store the definition of the proposed metrics and the result values of the calculated metrics.

Displaying Phase

The display part loads the metrics data from metrics model database and provides visual presentation such as chart, graph or illustration to display the metrics result.

8.0 Conclusions

In general the existing OOD metrics suffer from unclear definitions and a failure to capture OO-specific attributes. The attributes of data-hiding, polymorphism and abstraction are not measured at all and the attributes of ***inheritance*** and ***encapsulation*** are only partially measured. The proposed framework is most suitable to assess object oriented programs, and proved successfully used to assess Object Oriented Programs. The automated System is not only useful for assessing programs, but also a tool to find the deficiency in each program under assessment. The System can easily be used to assess programs at process level. The system can be used to assess both large and small programs. The adoption of the Object-Oriented constructs is expected to help produce better and cheaper software. Keeping on the evolution track means we must be able to quantify our software improvements. Metrics will help us to achieve this goal. Despite the criticisms, and with little further empirical or theoretical evaluation, other OOD metrics have been incorporated into a number of software measurement tools and look set to become industry standards.

8.1 Future Work

Our long term plan is to utilise, and build on, the best of the existing work in order to propose a set of basic, language-independent design measures that are theoretically sound as well as being acceptable, understandable and useful to all sections of the software engineering community. More work about empirical validation is necessary using proven statistical and experimental techniques in order to improve their interpretation. More clear interpretation guidelines for these metrics based on common sense and experience are necessary. Building quality systems has been the driving goal of all software engineering efforts over the last two decades. The lack of design and implementation guidance can lead to the misuse of the aspect-oriented abstractions, worsening the overall quality of the system. Important quality requirements, such as reusability and maintainability, are likely to be affected negatively due to the inadequate use of the aspect-oriented languages and respective abstract

9.0 References

[1] I. Bluemke, "Object oriented metrics useful in the prediction of class testing complexity," in *Proceedings of the 27th Euromicro Conference*, pp. 130–136, 2001.

[2] T. J. McCabe, "Complexity Measure," *IEEE Transactions on Software Engineering*, vol. 2, no. 4, pp. 308–320, 1976.

[3] S. R. Chidamber and C. F. Kemerer, "Metrics suite for object oriented design," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, 1994.

[4] M. Bundschuh and C. Dekkers, "Object-oriented metrics," in *The IT Measurement Compendium*, M. Bundschuh and C. Dekkers, Eds., pp. 241–255, Springer, Berlin, Germany, 2008.

[5] I. Vessey and R.Weber, "Research on structured programming: an wm piricist's evaluation," *IEEE Transactions on Software Engineering*, vol. 10, no. 4, pp. 397–407, 1984.

[6] T.Wandand R. Weber, "Toward a theory of the deep structure of information systems," in *Proceedings of International Conference Information System*, pp. 61–71, 1990.

[7] V. R. Basili and B. T. Perricone, "Software errors and complexity: an empirical investigation," *Communications of the ACM*, vol. 27, no. 1, pp. 42–52, 1984.

[8] M. H. Tang, M.H. Kao, and M. H. Chen, "An empirical study on object-oriented metrics," in *Proceedings of the 6th International Software Metrics Symposium*, pp. 242–249, November 1999.

[9] S. Sarkar, A. C. Kak, and G. M. Rama, "Metrics for measuring the quality of modularization of large-scale object-oriented

[10] Y.Zhou, B. Xu, and H. Leung, "On the ability f complexity metrics to predict fault-prone classes in object-oriented systems," *Journal of SystemsandSoftware*, vol. 83,no. 4, pp.660–674, 2010.

[11] H.M.Olague, L. H. Etzkorn, S. L. Messimer, and H. S.Delugach, "An empirical validation of object-oriented class complexity metrics and their ability to predict error-prone classes in highly iterative, or agile, software: a case study," *Journal of Software Maintenance and Evolution*, vol. 20, no. 3, pp. 171–197, 2008.

[12] L. C. Briand and J. W. Daly, "A unified framework for coupling measurement in object-oriented systems," *IEEE Transactions on Software Engineering*, vol. 25, no. 1, pp. 91–121, 1999.

[13] L. C. Briand, J.Wust, J.W.Daly, and D.Victor Porter, "Exploring the relationships between design measures and software quality in object-oriented systems," *Journal of Systems and Software*, vol. 51, no. 1, pp. 245–273, 2000.

[13] F. T. Sheldon and H. Chung, "Measuring the complexity of class diagrams in reverse engineering," *Journal of Software Maintenance and Evolution*, vol. 18, no. 5, pp. 333–350, 2006.

[14] R. Shatnawi, W. Li, J. Swain, and T. Newman, "Finding software metrics threshold values using ROC curves," *Journal of Software Maintenance and Evolution*, vol. 22, no. 1, pp. 1–16, 2010.

[15] W. Li and S. Henry, "Object-oriented metrics that predict maintainability, *"The Journal of Systems and Software*, vol. 23, no. 2, pp. 111–122, 1993.

[16] N. I. Churcher and M. J. Shepperd, "'Comments on' a metrics suite for object oriented design," *IEEE Transactions on Software Engineering*, vol. 21, no. 3, pp. 263–265, 1995.

[18] International Organization for Standardization. *"ISO/IEC 9126 - Information Technology- Software Product Evaluation - Quality Characteristics and Guidelines for their use".*

[19] H. Zuse. *"Software Complexity: Measures and Methods"*. Walter de Gruyer (New York), 1991.