Software Maintainability and Reusability using Cohesion Metrics

Adekola, O.D[#1], Idowu, S.A[*2], Okolie, S.O[#3], Joshua, J.V[#4], Akinsanya, A.O[*5], Eze, M.O[#6], EbiesuwaSeun[#7]

[#1]*Faculty, Computer Science Department, Babcock University,Ilishan-Remo, Ogun State, Nigeria*

[*2]*Faculty, Computer Science Department, Babcock University,Ilishan-Remo, Ogun State, Nigeria*

[#3]*Faculty, Computer Science Department, Babcock University,Ilishan-Remo, Ogun State, Nigeria*

[#4]*Faculty, Computer Science Department, Babcock University,Ilishan-Remo, Ogun State, Nigeria*

[*5]*Faculty, Computer Science Department, Babcock University,Ilishan-Remo, Ogun State, Nigeria*

[#6]*Faculty, Computer Science Department, Babcock University,Ilishan-Remo, Ogun State, Nigeria*

[#7]*Faculty, Computer Science Department, Babcock University,Ilishan-Remo, Ogun State, Nigeria*

Abstract - *Among others, remarkable external quality attributes of interest to software practitioners/ engineers include testability, maintainability and reusability.Software engineers still combat softwarecrisis and even chronic software affliction not because there is no standardized software development process but because enough attention is not given to seemingly insignificant but crucial details of internal design attributes such as cohesion and coupling especially in object-oriented systems. Consequently, the aftermath is increased maintenance cost, effort and time which negatively plague both the developers and users community. Also, reusability being an important part of quality design and time-to-market is equally affected.*

This work addresses how to use internal attribute as cohesion could improve software maintainability and reusability. This research also addresses general design principles of object-oriented and other reuse-oriented systems.

Keywords: *Testability, Maintainability, Reusability, Cohesion, Coupling, Software Affliction*

I. INTRODUCTION

Maintainability in software constitutes effort and ease required to modify, correct or improve the quality of a design or product. Maintenance could be done as a result of a need to add new features or functionalities, fix a bug or to increase the strength of the software. Maintenance constitutes an essential part of software's lifetime. Ahn et al., (2003) estimated that maintenance takes up to 80% of the total costof producing software applications. Expectation of achieving more reliable, quicker time-to-market and maintainable systems. A lot of research has gone into the areas of software reuse and maintenance due to the fact that these among other issues concern intimately system developers/architects/engineers rather than end-users. Therehas been enormous growth in software reuse research from the days of structured programming concepts to object-oriented methods and beyond (e.g. component based development) (Wang, 2000).

Most times, software developers have the capability of creating or producing software that functionas desired. Theirutmostchallengeis finding ways to produce software quickly enough to meet up with the growing demand for more products and at the same time having to maintain increasingly thriving software "crisis". Pressman& Maxim, (2015) express the phenomena as software affliction, a long-lasted pain or distress. This work does not look in the direction of software development life cycle or process flow but rather seeks to consider internal attributes such as source line of code (SLOC), complexity, cohesion and coupling with a narrowed focus on cohesion. The most important software internal quality metrics are cohesion and coupling (Chidamber&Kemerer, 1994). Generally, internal attributes (such as size and cohesion) whether in traditional or object orientedmethods are critical indicators of external attributes which include

understandability, maintainability and reusability. A clever but vital area to consider when it comes to attempting to alleviate software afflictionis toexamine what could be done with internalproperties such as complexity, cohesion, coupling etc.Common questions from literature include the following: are highly cohesivemodulesorcomponentsmore readily reusable, does attention to cohesion yield maintainable systems, do wehave a one-size-fit-all when it comes to software metrics to measure major external attributes?A software metric is a quantifiable measurement of some attributesor an attribute of a software product or process (Frakes& Kang, 2005).

A mapping of empirical world to formal, relational world is what is termed as measurement (Fenton &Pfleeger, 2010). Therefore, a measure is regarded as thenumber or symbol assigned to anentity bysuch mapping in a bid to get an attribute characterized.

Cohesion implies the degree of relatedness among class members while coupling simply connotes theinterconnectedness of modules within a software or a program. A class with high cohesion will be a difficult candidate for refactoring, i.e. difficult to split into separate classes (Dallal& Briand, 2009).From their meanings, high cohesion and low couplingshould implya good design as far as software is concerned. High cohesion,which is ranged within [0,1] shows good design as well as goodquality software (Chidamber&Kemerer, 1994; Okike, 2010a; Okike&Osofisan, 2008; Okike&Rapo, 2015). Further, cohesion is a measure of the degree of connectivity among a single class- suffice to say it reveals or indicates the relationship within a module. For coupling, it is in order to say it indicates relationships between modules or components of a software or system. Cohesion measures how one function performed by an entity relate to another. It is characteristic of most metrics to evaluate cohesion by considering if methods of a class access similar sets of instance variables (Mal &Rajnish, 2014).Coupling connotes how much a component knows about the inner workings (elements) of the other.

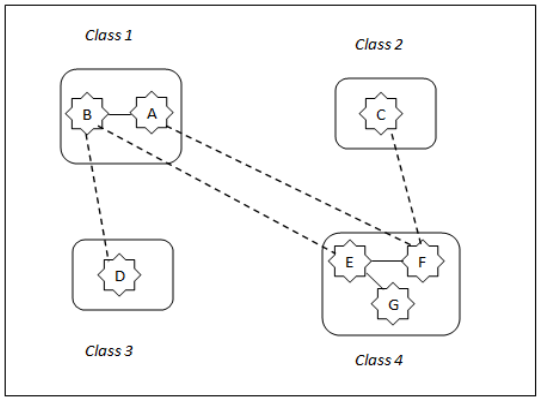

Fig 1: Simple illustration of cohesion and coupling; the thick lines indicate cohesion and the dotted lines indicate coupling(Source: Adapted from Dhanvani, 2013)

There are many cohesion metrics in literature but the metric to employ for a particular instanceneeds to be found out. Mal andRajnish, (2014), discussed a number of metrics which were empirically validated against notable open source software projects.

II. METHODOLOGY

This research embodies case studies, systematic literature reviews and surveys. Important requirements were identified in related papers. The relevant documents obtained were qualitatively analyzed for convergence, and relevant details were extracted using inductive approach. Existing measures were evaluated. This work leveraged on Chidamber and Kemerer metrics and Rajnish and Mal metrics and made proposition on inclusion of method-method interaction as part of consideration for cohesion measures.

III. LITERATURE REVIEW

In software engineering, qualities such as maintainability, reusability, flexibility, writeability and demonstrability are described as developer-orientedquality attributes (Berander, Damm, Eriksson, Gorschek, Henningsson, Jönsson, Kågström, Milicic, Mårtenssonn, Rönkkö, &Tomaszewski, 2005). While these are external software quality attributes, internal qualities are the likes of size, complexity, cohesion and coupling. The internal versus external quality attributes relationship could be fashionably intuitive, for example the more complex a code is the more difficult it is to maintain.

But,the exact functional form of the relationship is not very lucid. Therefore, this is connotes a subjectof serious research issue.

IV. COHESION

Using class illustration, cohesion means the extent or degree which a class carries a single, well-focused purpose. By implication, a better design should own high cohesion.It means a class encapsulates only properties and operations that are closely related. Interestingly, high cohesion is a desirable property of a program in that it positively impacts understandability, maintenance and reuse (Girish, 2014). An intimate example could be to illustrate cohesion as communication between father, mother, and child within a family while its counterpart called coupling could be communication in between two different families.

Consider the following unified modeling language(UML) class example:

Staff
+checkMail() +sendMail() +validateMail() +printLetter()

Figure: 2. A class design with cohesion.
Note: These functionalities might be appear logical but might not belong together

Staff
- salary -emailAddress
+setSalary(empSalary) +getSalary() +setEmailAddress(empEmail) +getEmailAddress()

Fig 2: A class design with high cohesion

In Figure 2, the Staff class should not consistcheckMail, validate emails or sendMail. These functions could go into a supposed E-mail class, hence, high cohesion would be feasible. In Figure 3, the Staff class has only actual information for setting and getting Staff related data. It does not include operations that should be handled by or separated to another class.

A. Types of Cohesion

The following are the variousclassifications of cohesion that require research attention:

1. Functional Cohesion: This means the various constituent elementsthat make up the module or component are grouped together simply because each or almost everyone in the group contributes to the module's single responsibility or well-focused task.
2. Informational cohesion: Here,the entity is said to represent a cohesive body of data and a set or group of independent actions or behaviours on the particular body of data.
3. Sequential Cohesion: This grouping occurs when parts of the modules output represent the input to the other.
4. Communication Cohesion: This is when parts of the module are organized in a group because they use or work on the same data.
5. Procedural Cohesion: This is when parts of the module are grouped together because they followa certain sequence or order of execution.
6. **Logical Cohesion:** This is whenparts of the module are put in the same group because they are logically grouped to do the same task but might have different nature.
7. Coincidental**Cohesion**: This is when modules have nothing really in common except for something like convenience.
8. Temporal cohesion: This is when sometime a component is used to initialize a system or set variables.

Generally, the fewer the quantity of instance variablesthe higher the strength of cohesion. The more the variables a method operates upon the more the likelihood of cohesiveness that method would exhibit to its class; this is a positive virtue (Okike&Rapo, 2015).Highly cohesive classes or modules, in general, are easier to maintain and are less frequently in need of changes.Such classes or modules are more usable than others simply because their design follows a well-focused purpose.

V. COUPLING

While cohesion is interaction between two or more elements within a module, coupling is interaction /

relationship between two modules. Coupling means the degree to which one class or component knows about another classor component (Beck & Diehl, 2011). Considering two classes named X and Y, given that X knows Y through its interface only, that is, Xinteracts or relate with Y through its application programming interface (API) then both classes are loosely coupled. But if class X, other thaninteracting class Y through itsinterface also interacts or relates through the non-interface property of class Y then one can say they are regarded as tightly coupled. If the designer decides to change class Y's non-interface part for a positive reason, class X breaks because of this tight coupling. Tight coupling makes writing tests harder.

Holub (2005) stated that the main problem with inheritance implementation is that it introduces unnecessary coupling in the form of fragile base class problem (i.e. when changes made to a base class impacts the functionality of many derived classes and spread throughout the system). Most practices of object-oriented programming recommend keeping the inheritance graph as shallow as possible. Overuse of inheritance worsens coupling, leading to less flexible and reusable classes. The use of composition instead of inheritance is also preferred.

B. Types of Coupling:

The following include the different types of cohesion:

1. Content Coupling: This israted highest and occurs when one of the module or componentdepends or relies on the internal workings of the other module. Invariably, when you make changesto the second module you will automatically need to make changes to the one that is dependent.
2. Common Coupling: This occurs when more than one modules share the same global data. Consequently, a change in the shared (common) resource engenders changes in those modules.
3. External Coupling: This is when more than one modules share an externally imposed data format and communication protocol.
4. Control Coupling: This is whena module controls or dictates the flow another and passes information from one to the other. That is, one component passes parameters to control the activity of another component.
5. Message Coupling: This is come about though state decentralization. This is seen as the loosest form of coupling, suchthat components communication is carried outvia message passing.
6. Data Coupling: This is recorded if only data are passed between modules.
7. Stamp Coupling: This is when data structure is used in transferring information from onemodule to another.

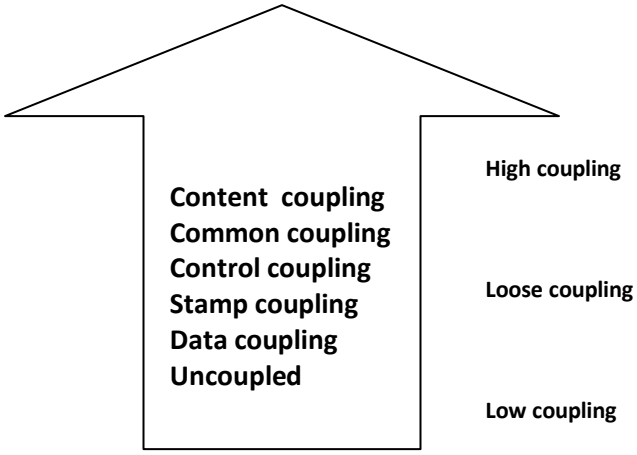

Fig 3: Illustration of degree/hierarchy of coupling(Source: Chawla, n.d)

The goal of a good design is to eliminate unnecessary coupling. This makes maintenance of the system much easier. Loosely coupled systems are made up of components which are independent or almost independent.

VI. MAINTAINABILITY ATTRIBUTES AND BENEFITS

Maintainability refers to the degree to which asoftware or component of a software can be easily modified in order to correct bugs, add quality attributes or adjustthe operating environment andthen improve the efficiency of the entire software.Generally, software maintenance phase demands that needed changes are made to the existing system (Michura, Capretz, & Wang, 2013).As a result of inevitable increases in the size and complexity of software products,software maintenance duties have also become increasingly herculean.Therefore, an urgent concern in the computing industry is the need to maintain and enhance software productsas cost effective as possible and within a short time. To meet this objective, concepts and techniques which lead todesigning more maintainable solution should be given serious consideration. In short, software

maintenance should no longer be a design afterthought; that is, it should beeasy for software maintainers to enhance the quality of the product withoutcompulsorily tearing down and rebuilding the substantial parts of the code. And one should be able to predict what happens to a system if change is required. This is the reason, from design perspectives, to consider paying attention to internal attributes that could improve maintainability.

The following illustrates maintainability and the external versus internal attributes:

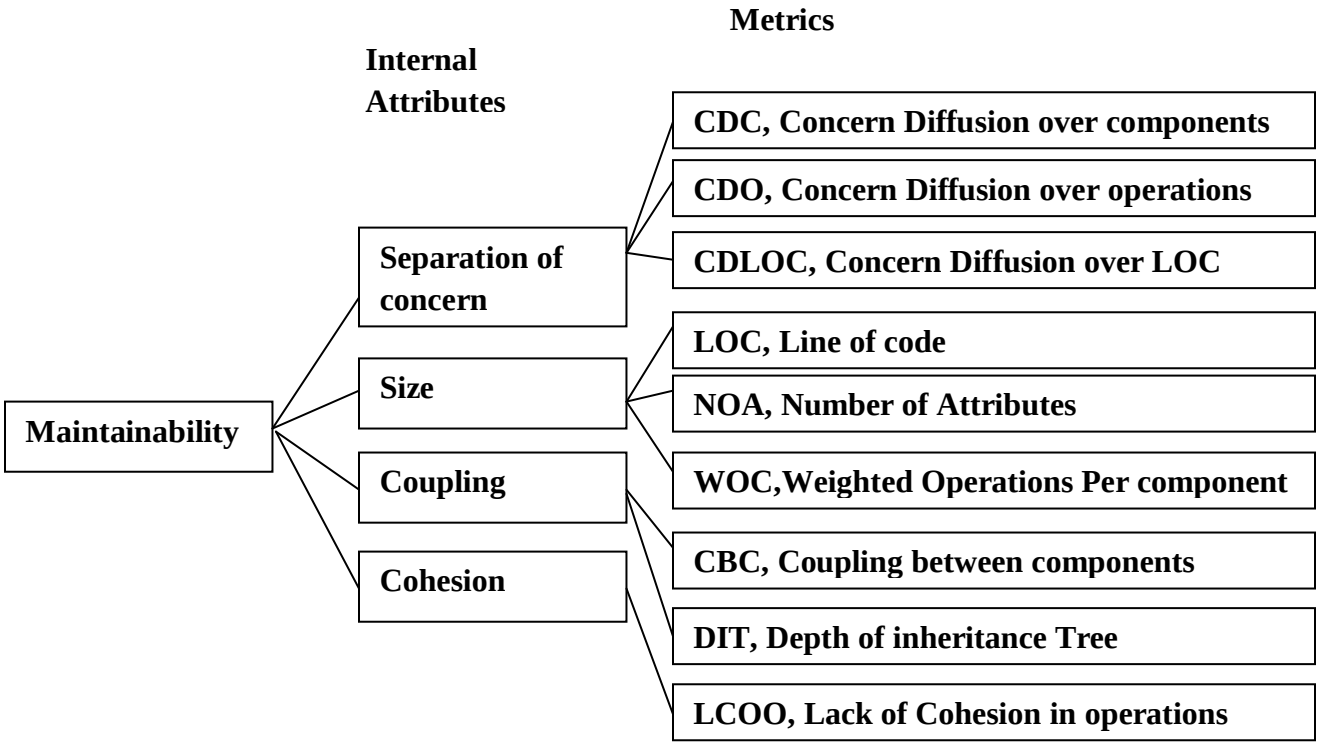

Fig 4: Framework for software measurement validation (Source: Garcia, 2014)

VII. Reusability Attributes and Benefits

There is a continuing effort in taking the advantage of reusing knowledge or artefacts right from procedural techniques to object-orientedsystem development, component oriented development and beyond. Prominent in this area are program library, application product lines, component based development, service oriented systems, legacy systems, program generator, aspect-oriented software development, design patterns, and commercial off the shelf software integration. These are categorised as systematic reuse areas where the full benefit of software reuse can only be achieved (Bhatnagar& Kumar, 2014). A potent weapon in the design of reuse elements or reusable components is how to reduce dependency and increase cohesion (Wang,2000). Properly designed components can be easily customized or replaced which in the end could facilitate maintenance (Crnkovic& Larsson, 2002).

A reusable software component is a software system, subsystem, module or program chunk that can be easily integratedinto newsoftware or program directly or after somenecessary changes have been made. Basili, Briand, &Melo, (1996) indicated that error density (that is, errors per thousand lines of code) dropped from 6.11 for systemsdeveloped without employing reuse to 0.12 for systems built from reusable components. It has also been discovered that 40% to 60% of code is truly reusable from one application to another application, 60% of design and code are reusable in when it comes to business applications, 75% of program functions are common in two or more programs, and only about 15% of the code found in most software is unique to

oneparticular application (Ezran et al., 2002). Maximizing the reuse of tested, certified and organized artifacts can generate improvements in cost, time and quality (Basili et al., 1996). The U.S. Department of Defencewas able to save $300 million annually by increasing its reuse level by only 1% (Computerworld, V27(49)). This presents a good case to venture into research that can improve software reuse.

Effective reuse is not a simple addition toexisting software development processes; it puts strong demandson development methods in order to be successful. Key issues to considerhere is to define models and metrics which can measuresoftware reusability (Antovski&Florinda, 2013). A metric is a quantitative indicator of an attribute of a thing while a model specifies relationships among metrics (Frakes& Kang, 2005).

VIII. Design Principles

Another subject of concern is design principles which have intertwined relationship with giving consideration to known internal attributes. Some of the commonest examples of design principles include striving for a clearly defined, single purpose per component, striving for loosely coupled and highly cohesivecomponents, developing components with overall future use in mind and also putting extra effort into error handling andmaking components robust (Suresh, (2011).

The software-intensive industry is relentless in finding ways to develop software faster, cheaper, more predictably, with requested functionality and quality and with sufficient maintainability. The key thing is to improve the process for developing and maintaining the appropriate software (Sommerville, 2010).
Robert C. Martin, from his research experience, compiled and proposed different object-oriented design principleswith a common acronym called SOLID Design principles (Martin, 2012). The meaning of the acronym is:

S- Single Responsibility Principle - A class should have one, and only one, reason to change
O- Open Close Principle- A class should be open for extensionand closed for modification
L-Liskov Substitution - Derived classes must be substitutable for their base
I-Interface Segregation Principle - Make fine grained interfaces that are client specific
D- Dependency Inversion Principle- Depend on abstractions, not on concretions

IX. Focus on Object oriented metrics suites

There are many traditional metrics found in literature which had been applied to measure many quality criteria like size, complexity, comment percentage and so on. Metric as cyclomatic complexity (McCabe, 1976) proved to be one of the best indicators to test reliability in a system. However, these traditional software measures would not scale well scale when it comes to handling object-oriented systems (Goldberg & Rubin, 1995).This is simply because basic object-oriented characteristics like polymorphism, inheritance, classes and object are notincorporated in their design (Kaur & Kaur, 2015). In object-oriented programming, much of the explicit branching statements, e.g. if, while and casestatements, have been replaced by implicit branching due to inheritance andevent-driven programming. This implies that cyclomatic complexity metric alonecannot decipher the complexity of an object-oriented program. The following are some OOD metrics but the more or less commonest is the CK metrics suite (Chidamber&Kemerer metrics) (Suresh, Pati& Ku, 2012).

X. CK Metrics Suite

This suite is referred to as being best indicators for fault proneness. It is a convenient tool to predict the reliability of a system. The metric suite helps developers to make better design decision and at the same time estimate testing effort(Suresh, Pati, & Ku, 2012).

The ChidamberandKemerer metrics suite originally consists of six metrics created to test some specific system characteristics. They are highlighted as follows:

1)WMC (Weighted Method per Class)

This is one of the metrics that has been remarkedaseffective in predicting testing and maintenanceeffort. It could perform better when complemented or combined with other metrics.
If there exists a Class C1, having methods as m1,m2,…, mndefined as members of a class. Let c1, c2,...,cn be labeled as the complexity of the methods, then WMC is illustrated with the following simple equation:

$$WMC = \sum_{i=1}^{n} ci, \text{ for } i \leftarrow 1 \text{ to } n. \quad (1)$$

Such that ci is regarded as the complexity of the methods or functionsassociated in the ith class.
If every method's complexity is unity, then that implies the value of WMC will become n, meaning the number of methods involved.

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

Thisis calculated as the maximum length of path from a class to the root class in the inheritance (hierarchy) tree. The higher depth shows more complexity in predicting its behavior.

*3)****NOC (Number of Children)***

This connoted the quantity of immediate sub-classes that are subordinate to a known class in the class hierarchy. This shows the influence of a class on either the system or the design.

*4)****CBO (Coupling between Objects)***

Coupling between Objects metric for a known class is the sum total of the quantity of other classes to which it is coupled. This helps in determining the complexity of testing on the design.

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

This is the set of methods that have the likelihood of being executed as a result of response to a message received by an object of that class. RCF is also a measure of the possible communication between the class and other classes. Large RFC showstendency of more fault.

6) ***Lack of Cohesion in Methods (LCOM)***

LCOM attempts to find the degree of methods similarity and is theoretically grounded upon ontology of objects according to Bunge, (1972). The ontology defines the set of characteristics or properties thatobjects share.
If there existsClass C1 having n methods M1, M2, …Mn. Then{ Ii } is the set of instance variables that are accessed by method Mi. There are n such set {I1 },…,{I n }.
Then the following two disjoint sets are defined:

$$A = \{ (Ii, Ij) \mid Ii \cap Ij = \phi \}, \quad (2)$$

$$B = \{ (Ii, Ij) \mid Ii \cap Ij \neq \phi \} \quad (3)$$

If all n sets { I1}…{ In } are ϕ then A =ϕ

LCOM is defined from the cardinality of the sets as follows:

$$LCOM = |A| - |B|, \quad \text{if } |A| > |B| \text{ or zero (0) otherwise} \quad (4)$$

LCOM is described as an inverse cohesion measure. When LCOM is (0) then a class is cohesive.

Eg. If class C has methods M1, M2, and M3,
let { I1} = (a,b,c,d,e} be set of instance variables in class C used by method M1
{ I2} = {a,b,e} for method M2
{ I3} = {x,y,z} for method M3

Then { I1} ∩ { I2} ≠ ϕ
{ I1} ∩{ I3} = ϕ
{ I2} ∩{ I3} = ϕ
.
LCOM = the number of null intersections – number of non-empty intersections.
The result is one in this example LCOM = 2 –1 = 1;
So, the larger the number of similar methods, the more cohesive the class is.
If none of the methods of a class display any instance variables, they have no similarity and the LCOM value for the class is zero (0). LCOM is intimately tied to methods and instance variables of a class, therefore, it is described as a measure of properties of an object.

Having a high value of LCOM means low cohesion and then the class might be in a better design if broken into two or moreseparate classes.Poor cohesion also means high complexity which can increase error tendency during system development.

XI. ROBERT C. MARTINS METRIC SUITE

Ideal models of dependency and abstraction are reflected by these metrics. It captures some good design principles and also representsa lucid description of stability in software.This metrics is commonly known as package metrics (Martin, 1994). It consists of the following:

a) Efferent Coupling (Ce): n(classes) outside the package that depend on classes within the package.

b) Afferent Coupling (Ca): n(classes) inside the package that depend upon classes outside the package.

c) Instability(I) =Ce / (Ce + Ca). (5)It implies the adaptability to of package to change.
It ranges within [0-1],such that when I = 0we haveabsolutely or completely stable package,and when I = 1 we have absolutelyinstable package.
d) Abstractness: This is the comparison of quantity of abstract classes (and also interfaces) to the total numberof classesin thepackage under consideration.

Abstractness (A) = abstractClasses / totalClasses. (6)

The range is [0-1];when A = 0 we have absolute concrete package; when A = 1 we have absolute abstract package.

e) Normalized Distance from Main Sequence (D)

D connotes the perpendicular distance of a package from the idealized line given by:

D = A+ I-1. (7)

Where D = 0depicts a package which coincides with the main sequence and
D = 1 depicts a packagethat is said to be far away from the main sequence.

XII. Other Related Works:

The following is a review of related works that indicated efforts in promoting parallel programming and supporting frameworks:

Shumway, (1997)carried out a empirical research on the relationship between class cohesion and size and concluded that there is no significant relationship between cohesion and class size as measured in number of byte code. Also, the data set used did not exhibit high reuse properties which could aid investigating the relationship between reuse and cohesion.
Badri&Badri (2004) proposed a class cohesion measures which attempted to consider other criterion that are characteristic of object orientation in assessing the relatedness of class elements. The researchers stated that class cohesion should not exclusively be based on common instance variables usage criteria.
Michura, Capretz, & Wang (2013) also stated that some system characteristics are essential to deal with issues of system complexity and its associated maintainability. The paper identifies factors that are responsiblefor difficulties in performing changes during maintenance, and also the necessary effects that may come with those changes especially object-oriented systems. However, the experimentation revealed that large systems would benefit more from the proposed metrics compared to small ones.
Rajnish& Mal, (2014) discussed the relevance of cohesion as a key design property in object oriented software as used in measuring connectivity within subsystems. While LCOM searches for absence of cohesion, their proposed work attempts to seek the degree of presence of cohesion. This research mainly modeled the interaction between global variables and methods within a program. Their experiment shows correlation between lines of code, LOC, compared to some existing cohesion metrics.

XIII. Measuring Functional Cohesion

As object-orientation requires a new way of thinking and a different way to design, measuring design elements also demands a different approach beyond traditional measures. Due to the complexity of object-oriented software,there is no single, simple measure of software quality for all cases (Michura, Capretz, & Wang, 2013).

The well-known metrics proposed by Chidamber and Kemerer (described as the CK metrics) can be used to measure some object oriented characteristics and can be used to predict defects during maintenance but do not give enough information as regards the difficulty in executing such changes (Michura, Capretz, & Wang, 2013). Also, the proposed metrics of Mal &Rajnish, (2014) emphasized on predicting system reusability.

XIV. Lack of Cohesion Metrics

One of the most common lack of cohesion metrics is that of Chidamber and Kemerer.This has been worked upon over the years to ensure improvement and to see to its application specificity.

XV. Cohesion Presence Metrics

Mal &Rajnish., (2014) proposed a cohesion metrics which shows correlation with Number Line of Code Property, NLOC. It is also said to be a good indicator of reusability. This work principally considered variable-method interactions (Mal &Rajnish., 2014).

XVI. Proposed Metrics

Figure 6 is a model of the proposed metrics which (in addition to variable-method) consider method-method interactions which is an extension or adaptation of Mal &Rajnish, (2014) metrics.Notably, boththe Chidamber and KemererLCOM metrics and Mal and Rajnishshare in common the concept of variable-method interaction but the former measures

the absence of cohesion while the latter measures its presence.

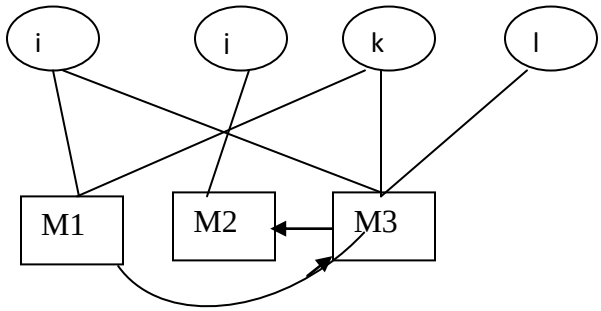

Fig5: Illustration of class cohesion measure indicating variable-method interaction and method-method interaction.

The i, j, k and l represent the variables while M1, M2 and M3 are the methods.
The mathematical model of the model is described as follows:
Let class C consists of a set of instance variables V = {v1, v2,…,vn} and a set of methods M = {m1, m2, …, mn}.
Then Proposed Class Cohesion, PCC is evaluated based on ith instance variable of a class (CVi) for method-data interaction and invoked methods in the class (CMi) for method-to-method interaction.

$$CVi = \frac{\text{Numberof methods sharing the instance variable i of a class}}{\text{Number of all methods in the class}}$$

$$CVi = \frac{n(M(Vi))}{n(M)} \quad (8)$$

$$CMi = \frac{\text{number of methods invoked by other methods in the class}}{\text{Number of all methods in the class}}$$

$$CMi = \frac{n(M(m_i))}{n(M)} \quad (9)$$

$$PCC = CVi + CMi \quad (10)$$

The range of PCC is [0,2]

Then the mean CVi, andCMi cohesion count of a class of n instance variable is computed as follows:

$$\text{Cohesion PCC} = \frac{\sum_{i=1}^{n}(CVi + CMi)}{n} \quad (11) \quad n \geq 1$$

Then, to evaluate a system's cohesion comprising r classes, the following applies:

Such that n = number of instance variables of a class

$$SysCo = \sum_{i=1}^{r} \text{Cohesion PCC} \quad (12)$$

XVII. Discussion

The cohesion measurement discussed and proposed is a predictive approach to designing OO software. This informs the designer of the system status and helps to determine what proactive step to take in ensuring a system with less future problems. This work evaluates existing metrics, design principles, importance of maintainability and reusability properties, cohesion and coupling and most importantly considers how to improve cohesion measures for the benefit of software developers.

XVIII. Conclusion and Recommendation

Software engineering is a disciplined approach towards creating quality software products. To get value for effort put into software design some subtle but critical underlying characteristics need to be given serious attention by knowledge engineers, software developers and practitioners. Remarkable quality attributes such as maintainability and reusability are part of external properties of a system. The fulcrum of these is not far from the internal attributes. And two amongthose rated most important are cohesion and coupling attributes. During maintenance for example, these have direct impacts on what the designer of software will go through let alone if such solution were to be maintained by a different developer. This report discussed the relevance of these internal attributes as they relate to reusability and maintainability. Existing works on cohesion metrics were evaluated, design principles were discussed and a narrowed focus was given to cohesion measures. Most works reviewed focused on the important characteristic of cohesion which models the relationship between instancevariables and methods within a class. This work considers additionalbehaviour as method-method interaction. Future recommendation is the consideration or addition of other class characteristics (e.g. discrimination anomaly) that could improve cohesion measurement as exhibited by different designs.

References

1. Ahn, Y., Suh, J., Kim, S., & Kim, H. (2003). The Software Maintenance Project Effort Estimation Model Based on Function Points. *Journal of Software Maintenance Evolution: Research and Practice*, *15*, 71-85.
2. Allen, H. (2005*). Introduction to Object-Oriented Analysis and Design*. Best

Software Canada Ltd. Printed in Canada 8920 Woodbine Ave. Suite 400.

3. Antovski, L., &Florinda, I. F. (2013). Review of Software Reuse Processes. *IJCSI,International Journal of Computer Science Issues, www.IJCSI.org*, *10*(6), 83-88.
4. Badri, L., &Badri, M., (2004). A Proposal of a New Class Cohesion Criterion: An Empirical Study. *Journal of Object Technology, Published by ETH Zurich, Chair of Software Engineering, JOT*, *3*(4).
5. Basili, V. R., Briand, L. C., &Melo, W. L., (1996). How Reuse Influences Productivity in Object-Oriented Systems. *Communication of the ACM*, *39*(10), 104-116.
6. Beck, F., & Diehl, S., (2011). On the Congruence of Modularity and Code Coupling. *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering, ACM*, NY, USA, 354-364
7. Berander, P., Damm, L., Eriksson, J., Gorschek, T., Henningsson, K., Jönsson, P., Kågström, S., Milicic, D., Mårtenssonn, F., Rönkkö, K., &Tomaszewski, P. (2005). *Software quality attributes and trade-offs*. Blekinge Institute of Technology.
8. Bhatnagar, V., & Kumar, A., (2014). Prospective of Software Reusability. *International Journal of Application or Innovation in Engineering & Management (IJAIEM)*,*3*(1), 411-414.
9. Bunge, M. (1972). *Treatise on Basic Philosophy: Ontology II: The World of Systems*. Riedel, Boston, USA.
10. Chawla, J. (n.d.). *Cohesion and Coupling*. Retrieved from https://www.slideshare.net/jagneshchawla/cohesion-coupling.
11. Chidamber, S. R., &Kemerer, C. F., (1994). A Metrics suite for object Oriented Design. *IEEE Transactions on Software Engineering*,*20*(6), 476-493.
12. Computerworld, Software Reuse Plans Bring Pay backs, Computer world, *27*(49), 73-76. Anthes, Gary I.
13. Crnkovic, I., & Larsson, M. (2002). *Building Reliable Component-Based Software Systems*. Boston, London: ArtechHouse.
14. Dallal, J. A., & Briand, L., (2009). A Precise Method-Method Interaction-Based Cohesion Metric for object-oriented classes. *ACM Transaction on Software Engineering and Methodology (TOSEM)*. TR, Simula Research Laboratory.
15. Dallal, J. A. (2011). Measuring the Discriminative Power of Object-Oriented Class Cohesion Metrics. *IEEE Transactions on Software Engineering*,*37*(6), 788-804.
16. Dhanvani, J. (2013). *Difference between Cohesion and Coupling*. Retrieved from http://freefeast.info/difference-between/difference-between-cohesion-and-coupling-cohesion-vs-coupling/
17. Ezran, M., Morisio, M., & Tully, C. (2002), *Practical Software Reuse*. Springer, 374.
18. Fenton, N., &Pfleeger, S. (2010). *Software metrics: A rigorous and practical approach* (2nd ed.). Boston, MA: PSW Publishing.
19. Frakes, W. B., & Kang, K. (2005). Software Reuse Research: Status and Future. *Journal IEEE Transactions on Software Engineering*, *31*(7), 529-536.
20. Garcia, A. (2014). *Framework for Software Measurement Validation*. Departamento de Informática PUS research group.
21. Girish, K. K. (2014). Conceptual Cohesion of Classes (C3) Metrics. *International Journal of Science and Research (IJSR)* ISSN (Online) 2319-7064.
22. Goldberg, A., & Rubin, K. S. (1995). *Succeeding with objects: Decision frameworks for project management*. Boston, MA, USA: Addison-Wesley.
23. Kaur, M., & Kaur, R. (2015). Improving the Design of Cohesion and Coupling Metrics for Aspect Oriented Software Development. *International Journal of Computer Science and Mobile Computing, IJCSMC*, *4*(5), 99 – 106.
24. Liskov, B. &Guttag, J. (2000). *Program development in Java: Abstraction, specification, and object-Oriented design* (1st ed.). Addison-Wesley Professional.
25. Mal, S., &Rajnish, K. (2014). New Class Cohesion Metric: An Empirical View. *International Journal of Multimedia and Ubiquitous Engineering*,*9*(6), 367-376.
26. Martin, R. C. (2012). *Clean code: A handbook of agile software craftsmanship* (1st ed.). Upper Saddle River, NJ, Boston: Prentice Hall.
27. McCabe T. J. (1976). "A Complexity Measure". IEEE Transactions on Software Engineering: 308–320.
28. Michura, J., Capretz, M, & Wang, S. (2013). Extension of Object-Oriented Metrics Suite for Software Maintenance.

Hindawi Publishing Corporation ISRN Software Engineering, 2013(276105), 14.

29. Okike, E. U., &Osofisan, A. (2008). An Evaluation of Chidamber and Kermerer's Lack of Cohesion in Methods Metric Using Different Normalization Approaches. *Afr. J. comp. & ICT,1*(2), 35- 43.
30. Okike, E. U. (2010a). A Pedagogical Evaluation and Discussion about the Lack of Cohesion in methods (LCOM) Metric Using field Experiment. *International Journal of Computer Science Issues, 7(*2), 36-43.
31. Okike, E. U. (2010b). A Proposal for Normalized Lack of Cohesion in Method (LCOM) Metric Using Field Experiment. *IJCSI International Journal of Computer Science Issues*, *7*(4), 5.
32. Okike, E. U., &Rapo, M., (2015). Using Cohesion and Capability Maturity Model Integration (CMMI) as Software Product and Process Quality criteria: A case study of Software Engineering practice in Botswana. *International Journal of Computer Science and Information Security (IJCSIS),13*(12), 140-149.
33. Pressman, R. S., & Maxim, B. R. (2015). *Software engineering: A practitioner's approach* (8th ed.). McGraw-Hil
34. Sommerville, I. (2011). *Software engineering* (9thed.). New York: Addison Wesley.
35. Shumway, M. F. (1997). Measuring Class Cohesion in Java. (Masters dissertation, Computer Science Department, Colorado State University, Technical Report CS-97-113.
36. Suresh, G. R., (2011). Strategies for Deploying Reusable Software Components. *International Journal of Graphics & Image Processing*, www.ifrsa.org, *2*(4), 264-273.
37. Suresh, Y., Pati, J., & Ku, R. S. (2012). Effectiveness of Software Metrics for Object-Oriented System. *SecondInternational Conference on Communication, Computing & Security [ICCCS-2012], Department of Computer Science and Engineering, National Institute of Technology, Rourkela, India, Procedia Technology*, *6*(2012), 420–427.
38. Wang, J. A. (2000). *Towards Component-Based Software Engineering*. Department of Computer Science and Information Systems Univ

39. Ersity of Nebraska, Kearney Kearney, NE 68849,USA