Journal of Theoretical and Applied Information Technology
15th December 2018. Vol.96. No 23

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

FUNCTIONAL COHESION METRIC FOR OBJECT-ORIENTED SYSTEMS

[1]OLUBUKOLA D. ADEKOLA, [2]SUNDAY A. IDOWU, [3]SAMUEL O. OKOLIE, [4]JONAH V. JOSHUA, [5]ADEWALE O. ADEBAYO, [6]ADERONKE O. OLUSOGA

[1, 2, 3, 4, 5]Babcock University, Ilisan-Remo, Ogun State, Nigeria

[6]Independent Researcher, 309 Clareview Station Drive Edmonton, Canada

E-mail: [1]adekolao@babcock.edu.ng, [2]idowus@babcock.edu.ng, [3]okolies@babcock.edu.ng, [4]joshuaj@babcock.edu.ng, [5]adebayoa@babcock.edu.ng, [6]aderonkeolusoga@gmail.com

ABSTRACT

Controlling software development process aids ensuring the quality of the output, but this is dependent upon availability of means of measurement. A major concern that makes software difficult to maintain or reuse is the complexity of the internal design and the most common to Object-Oriented Design (OOD) are cohesion and coupling. Traditional metrics could not scale in measuring cohesion in object oriented systems. Existing static cohesion metrics (for OOD) using variable-method and method-method interactions are satisfactory, but neglect semantic aspects of the software. On the other hand, existing semantic and joint static and semantic cohesion metrics fall short by relying on analysis of identifiers and comments, which are unstructured data with known setbacks. This study, therefore, developed a static and semantic functional cohesion metric that employed data hiding and object behaviour as a representation of OOD domain concept.

Keywords: *Attribution, Cohesion, Coupling, Maintainability, Metrics, Reusability*

1. INTRODUCTION

Software desirable characteristics include adaptability, maintainability, reliability, reusability and understandability. Increase in software complexity inadvertently reduces these characteristics [1]. The desire to control and manage complexity leads to the development of metrics. Generally, high cohesion and low coupling enhances good- design ([2, 3]), and coupling can be intuitively reduced by improving cohesion [4]. Cohesion in object-oriented paradigm is how elements or members of a module (or a class) are related or connected to one another. Coupling refers to the dependency among different modules or classes (that is, how much a class knows about or uses the inner elements of another). Cohesion measures connectedness of members of a single class, which indicates extent of relationship within a module (internal strength). It also indicates whether a class represents single abstraction or multiple abstractions. Multiple abstractions result in low cohesion. Low cohesion indicates monolithic (difficult to change single large block) classes that are difficult to maintain, understand, and reuse.

Maintainability of software includes the effort required to use, modify, correct or improve the quality of its design and more. It is estimated that maintenance takes up to 80% of the total cost of producing software applications [5]. Invariably, any engineering efforts or techniques that can improve maintainability should not be an afterthought [6]. The crux of the argument is that it is superior for software maintainers to enhance the product without having to rebuild a major part of it. In this quest, consideration should be given to researches that could enhance internal design such as cohesion [7]. Benefits of maintainable software include (i) having products that are easy to update and enhance, (ii) propensity for reuse, which would reduce the cost of update time, and (iii) ease of correcting faults found in software.

Reusability is the probability of using an existing design, artefact, product or knowledge to build a new system, with the expectation of achieving more reliable, quicker time-to-market, and possibly easier maintainable systems. Extant findings revealed that 40% to 60% of code is reusable, 60% of design and code are reusable in business applications, 75% of program functions are

Journal of Theoretical and Applied Information Technology
15th December 2018. Vol.96. No 23

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

common to more than one program, and only 15% of the code found in most systems is unique and new to a specific application [8]. Notable in systematic reuse environment are program libraries, design patterns, component based development, program generator, and aspect-oriented development paradigms. A known potent weapon in the design of reuse elements or reusable components is how to reduce dependency and increase cohesion [9]. Objects put for reuse should have good cohesion, preferably functional. Figure 1 simply depicts cohesion and coupling. The lines connecting the stared-alphabets indicate cohesion and the broken lines indicate coupling.

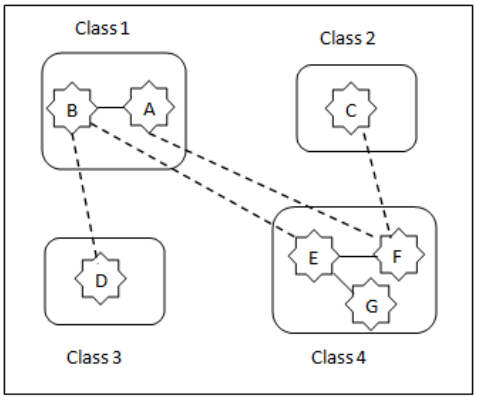

Figure 1: Cohesion and Coupling (adapted from: [10])

Many programs are written to run without considering internal design structure. In object-oriented design, one of the most common internal attributes that reveals a convoluted design (difficult to maintain) is cohesion. Developing metrics to measure cohesion in software is a proactive approach to ensuring software maintainability and reusability properties, in appropriately informing practitioners about design decision they need to make. Traditional metrics could not scale in measuring cohesion in object-oriented systems. ***Existing static OOD cohesion metrics majorly depend on measuring attribute-method interactions which do not appropriately represent what the designers of object-oriented software system often think of a class.*** To a designer, a class is regarded as a set of responsibilities that approximate the concept from the problem domain and also that information hiding has conceptual meaning in a class make-up. Therefore, the extant static metrics neglect semantic aspects of the software. ***Also, existing semantic and joint static and semantic cohesion metrics fall short by relying on analysis of identifiers and comments; codes that falter in commenting rules and attribution would not benefit well from these models of metrics.*** There is tendency for inaccurate computations for cohesion. ***The limitations of available semantic and variable-method interaction based cohesion metrics contribute immensely to code maintenance difficulties and consequently adversely affects the utility of developed software***. This study, therefore, aimed at developing a static and semantic functional cohesion metric that employed data hiding and object behaviour which promotes measuring the degree of single abstraction as a representation of OOD domain concept. The specific objectives were to develop a mathematical model and algorithm of an improved functional cohesion metric that captures varying strength of cohesion and makes efficient use of data hiding and abstraction, to design and develop an automated metric tool of the mathematical model, and to evaluate the behaviour of the developed metric tool.

A mathematical model, capturing variable-method interaction, method-method interaction and hidden data access, adequately representing the domain concept of OOD and indicating varying strength of cohesion, was developed leveraging on Mal and Rajnish metrics for its variable-method static part. The algorithm representing the mathematical model, which used three major factors, namely variable-method interaction, method-method interaction and hidden data access, was subsequently created. An automated cohesion metric tool, StaSem_C, was developed using Java Compiler Compiler (JavaCC), adopting regular expression used in formal language theory to analyse and classify sequence of symbols sought within a source code. It uses Backus-Naur Form production to analyse token sequence and determine the structure of the program. Different source codes, from student projects and open source software projects as data sets, were used to empirically validate the StaSem_C. The behaviour of StaSem_C was evaluated using purposefully selected source codes that were also deliberately re-factored for cohesion. No difficulty of being an ethical researcher was encountered.

2. RESEARCH QUESTIONS

The research questions are:

a) What design attributes are worth giving stern consideration in engineering maintainable and reusable software?
b) What should characterize a cohesion metric model suitable for OOD?

Journal of Theoretical and Applied Information Technology
15th December 2018. Vol.96. No 23

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

c) What metric model holistically captures structure and domain concept of object oriented systems

3. OUTCOMES

3.1 Functional Cohesion Metric Mathematical Model

This subsection describes the developed mathematical model, a functional cohesion metric. Figure 2 illustrates class variable-method and method-method interactions.

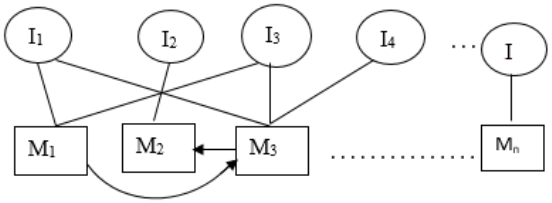

Figure 2 : Class Variable-Method and Method-Method Interactions

The circles represent member variables (I), the rectangles member methods (M), and the lines (edges) joining them, members' interactions. This is complemented with measurement for hidden data access, which semantically represents a class design as a set of responsibilities that approximate the domain concept. This directly represents the concept of the problem and solution domains of OOD. The support for security invariably increases single abstraction. Measuring the degree of single abstraction, therefore, gives an adequate report of cohesion. In addition to consideration for member interactions, hidden data inherently bind a class together and control object behaviours.

Consider a class C having methods M = {M_1, M_2, M_3,…, M_n} and a set of variables I = { I_1, I_2, I_3,…, I_m} accessed or used by M_i. The cohesion metric value α is a ratio scale ranging between 0 and 1 (α ϵ [0, 1]). α = 0 means no cohesion, and α ϵ [0.5, 1] indicates cohesion progressively.

A.

Cohesion value of variable I_i ($CV(I_i)$) is:

$$CV(I_i) = \frac{n(MI_i)}{n(M)} \qquad (1)$$

where $n(MI_i)$ = Number of methods sharing or using variable I_i, and n(M) = Total number of methods in the class.

Mean Cohesion Value (CC) of all variables for a class, therefore, is

$$CC = \frac{\sum_1^n CV(I_i)}{n(I)} \qquad (2)$$

where n(I) = total number of variables present in the class; n(I) > 0.

A class with no relationship between variables and methods gives CC = 0.

Cohesion value of method-method interaction (CMinv) is:

$$CMinv = \frac{n(Minv)}{n(M)} \qquad (3)$$

where n(Minv) = number of method invocations within the class, exclusive of coupling.

Moreover, complementing the structural analysis through capturing the domain concept, a measure of data hiding rate and a report of varying strength of cohesion are done. A computation of the cohesion value for hidden data access yields:

$$CV_h = \frac{n(HA)}{n(I)} \qquad (4)$$

where $n(HA) =$ *number of hidden variable access.*

Combining static and semantic view together, cohesion measure combines equations (2), (3), and (4) to have a normalized metric (for a class):

$$StaSem\_C = \frac{CC + CMinv + CVh}{f} \qquad (5)$$

where f = 2 when CMinv = 0 or
f = 3 when CMinv > 0 in a class.

The divisor, f, is the number of factors in consideration (variable-method interaction, method-method interactions, and number of hidden data access supported).

If $StaSem\_C_i$ symbolizes cohesion value for class **i**, the cohesion value of a program having **k** total number of classes (CoS) is:

$$CoS = \frac{StaSem\_C_1 + StaSem\_C_2 + \ldots + StaSem\_C_k}{k}$$

which is the same as:

$$\sum_{i=1}^{k} StaSem\_C_i$$

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

$$CoS = \frac{(6)}{k}$$

The cohesion metric value of a program comprising a number of classes is the sum of the cohesion metric values of the classes divided by the total number of classes.

3.2 The Cohesion Metric Tool

This subsection presents the software representation of the functional cohesion metric mathematical model, StaSem_C. which acts as a parser and a metric calculator. It takes as input typical Java source code and analyse it using regular expression productions to identify different tokens in the code. The program keeps track of how many methods used a particular variable, how many methods are called within a class which are methods of the class, and how many data are hidden. This is aggregated and normalized as metric value using ratio scale.

A typical source code input is translated into a compilation unit, which is an object representation of an abstract syntax tree (AST). This provides a convenient mechanism to navigate the tree to identify pertinent patterns in source code. The following is an example of how a typical code is analysed to discover useful patterns:

```
int sum(){
  …….. //code(s) to do something
   return 0;
 }
```

The code construct is broken into tokens of the following sequence:

“int”, “ ”, “sum”, “(”, “)”,
“ ”, “{”, “\n”, “\t”, “return”
“ ”, “0”, “ ”, “;”, “\n”,
“}”, “\n”, “”

It identifies the kind of each token as follows:

KWINT, SPACE, ID, OPAR, CPAR,
SPACE, OBRACE, SPACE, SPACE, KWRETURN,
SPACE, VALCONST, SEMICOLON, SPACE,
CBRACE, EOF

This sequence is sent to the parser to determine the structure of the program. Token SPACE is ignored. The analyser employs regular expression production to identify and classify valid tokens. The parser specification also consists of Backus–Naur form (BNF) production which specifies the legitimate sequences of tokens of error-free input.

Figure 3 is an example of object representation of a typical code viewed as an AST. The nodes on the tree are the elements, such as methods, in the program. As the tree is traversed, it becomes easy to collect methods that use a particular variable, methods that interact with another and the proportion of hidden data access. The developed tool, StaSem_C, uses those to compute the metric.

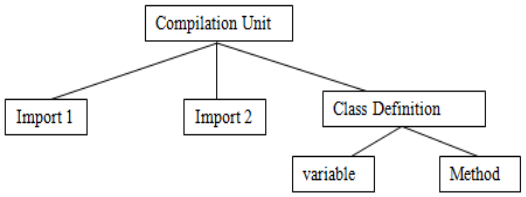

Figure 3: Typical Source Code Translation into AST

3.2.1 Architecture of major system component

Figure 4 represents major components of the StaSem_C design.

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

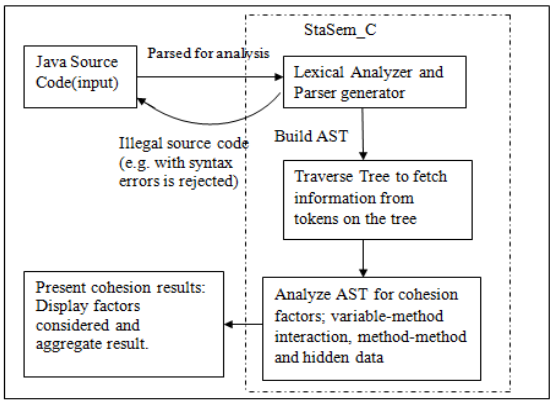

Figure 4: Architecture of major components of StaSem_C Design

3.2.2 Algorithm development

Algorithms of StaSem_C, performing source code parsing and metric calculations, are presented in this subsection. Major tasks are to: (i) identify each method in a class, keep track of variables used, and report the number of methods that use a particular variable divided by the total methods in the class, (ii) find the intersection of all the methods in the class, and the entire method calls to compute method-method interactions within the class (ignoring coupling), and (iii) account for hidden data in a class compared to the total data present. The results of the three factors are summed up, and then averaged as the functional cohesion metric of a class. The following is the presentation of step-by-step flow of the program:

Algorithm 1 Identifying and Collecting Variable to method interaction in the code:

Step 1: Get java file path location
Step 2: Get root folder
Step 3: Get current level
Step 4: Parse files with ".Java" extension
Step 5: Convert each java file into Abstract Syntax Tree
Step 6: Get global variables and store in ArrayList
Step 7: Visit Class Node from each file
Step 8: Get and store Class Name
Step 9: Visit Method nodes from each class
Step 10: Store method names in ArrayList
Step 11: Store method names and method in HashMap
Step 12: Define Compilation Unit
Step 13: For each method in Hashmap
13.1: Get Map Value
13.2: Convert Method to Abstract Syntax Tree
13.3: Iterate over tree nodes
13.4: Store each variable referenced in Method in HashMap(String, List)
Step 14: Remove duplicate variable references
Step 15: Store New list in HashMap(String, List) with method name

Algorithm 2 Identifying and Collecting Method to method interaction:

Step 1: Get java file path location
Step 2: Get root folder
Step 3: Get current level
Step 4: Parse files with ".Java" extension
Step 5: Convert each java file into Abstract Syntax Tree
Step 6: Visit Class Node from each file
Step 7: Get and store Class Name
Step 8: Visit Method nodes from each class
Step 9: Store Method Names in List
Step 10: For each Method
10.1: Get Method calls
10.2: Store Each call in a HashMap(String, List) and the corresponding method referenced
Step 11: Remove duplicate method calls form List
Step 12: Find and Retain the intersection of calls and all methods existing in the class
Step 13: Store New list in HashMap(String, List)

Algorithm 3 Identifying and Collecting Hidden Data Access:

Step 1: Get java file path location
Step 2: Get root folder
Step 3: Get current level
Step 4: Parse files with ".Java" extension
Step 5: Convert each java file into Abstract Syntax Tree
Step 6: Get Global variables and store in ArrayList
Step 7: Get variables with private modifier
Step 8: Store variables with private modifiers in a list

Algorithm 4 Getting Metric Computation for Variable to Method Interaction:

Step 1: If no methods or global variables exist, cohesion metric ← 0
Step 2: Parse each method select variables referenced in method
2.1 Remove Identical variables that appear more than once in a method
2.2 Total number of methods that referenced a variable divided by the total number of methods
2.3 Add result of each method to a list
Step 3: Sum result of each method in the list
Step 4: Divide the sum (in Step 3) by the total number of global variable
Step 5: Final output multiplied by 100 is variable to method metrics (represented in %)

Algorithm 5 Getting Metric Computation for Method to Method Interaction:

Journal of Theoretical and Applied Information Technology
15th December 2018. Vol.96. No 23

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

Step 1: If no methods or global variables exist, cohesion metric ← 0
Step 2: Get all methods in class in a list
Step 3: Remove duplicate methods
Step 4: Remove duplicate method calls or references in class
Step 5: Get total number of methods in class
Step 6: Get total number of method calls or references in class
Step 7: Do an intersect between method calls in class and the methods in class, output is a list method calls in found in that class (to eliminate computing for coupling)
Step 8: Divide the total number of method calls(without duplicates) in class by the total Number of methods in that class
Step 9: Final output multiplied by 100 is method to method cohesion metrics (in %)

Algorithm 6 Getting Metric Computation for Hidden Data Access:

Step 1: If no methods or global variables exist, cohesion metric ← 0
Step 2: Get and Count number of hidden variables in class
Step 3: Get and count total number of variables in class
Step 4: Divide total number of hidden variables in class by the total number of global variables in class
Step 5: Final output multiplied by 100
Hidden data cohesion metrics

Algorithm 7 Getting Metric Computation for Total Cohesion:

Step 1: Add Final output of Variable to method, method to method and Hidden Data Access in a list as Total cohesion cumulative and set f ←3
Step 2: If Variable to method result is equal to 0, Then Total cohesion ← 0
Else If method to method is equal to 0, Then Total cohesion divided by f-1
Else Total cohesion divided by all the factors, f
EndIf
EndIf
Step 3: Display Total cohesion (StaSem_C)
Step 4: Stop

3.3 Evaluation of the Behaviour of StaSem_C

The results of validation and evaluation of the behaviour of StaSem_C are presented in this subsection. Section 3.3.1 presents sample code analysis results and section 3.3.2 the dataset requiring maintenance efforts (candidates for re-factoring) results.

3.3.1 Sample code analysis results

Some datasets, described as category A data (classes carefully chosen for exhibiting high cohesion), were used to test StaSem_C. They are (i) Calculator.java, (ii) Rectangle.java, (iii) Circle.java, (iv) Clock.java, and (v) BankAccount.java. Figure 5 presents an output of StaSem_C identifying code attributes and behaviours during cohesion analysis. Table 1 shows category A Mal and Rajnish static and semantic cohesion characteristics results.

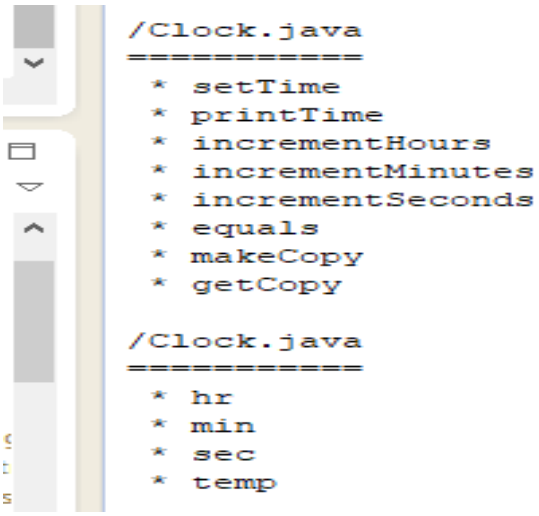

Figure 5: Screen shot of tool identifying class members such as methods and identifier in source code

Table 1: Reflection of Category A Static and Semantic Cohesion Characteristics

S/N	Carefully written class reflective of static and semantic cohesion	Mal and Rajnish Static cohesion experiment (Range: 0..1)	StaSem_C; static and semantic cohesion metric (Range: 0..1)
1	Calculator.java	1.00	1.00
2	Rectangle.java	0.67	0.83
3	Circle.java	0.67	1.00
4	Clock.java	0.47	0.75
5	BankAccount.java	0.22	0.50

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

The results show that Mal and Rajnish static metric gives less importance to semantic and domain idea for OOD idiosyncrasy.

Figure 6 presents an output of StaSem_C cohesion metric tool on a particular source code. Table 2 shows cohesion reports of Mal and Rajnish static and StaSem_C metrics using open source project (AccountingSoftware folder containing an entire project downloaded as open source software from www.github.com) as data set.

```
==========Calculating Metrics Variable to Method Interacti
Print Variable Used/method result in List : [0.75, 0.75, 0.75,
Number of Variables in Class : 4
Sum of Mi/M : 2.375
CC Metric For Class is : 59.375%
==========Calculating Metrics Method to Method Interaction
Methods in Class :[makeCopy, getCopy, equals, incrementMinutes
Methods invoked :[print, incrementMinutes, incrementHours]Size
[print, incrementMinutes, incrementHours]methods : [makeCopy,
Intersection[incrementMinutes, incrementHours]
Methods Invoked that are existing in class :[incrementMinutes,
Methods to Method Cohesion :25.0%
==========Calculating Metrics Private Variables to Variabl
Private Variables in Class :[[hr], [min], [sec]]Size :3
Private Variable to Normal Variable Cohesion :75.0%
[0.59375, 0.25, 0.75]
After Removing Zeros :[0.59375, 0.25, 0.75]
Total Cohesion : 0.53125
```

Figure 6: Screen Shot of StaSem_C Result of Cohesion Analysis of a Typical Java Source Code

Table 2: Cohesion Reports of Static and StaSem_C Metrics Using Open Source Project as Data Set

S/N	Carefully written class reflective of static and semantic cohesion	Mal and Rajnish Static cohesion Experiment (Ratio used: 0..1; 0-worst, 1-best)	StaSem_C; static and semantic cohesion metric (ratio; 0..1)
1	BankAccountDAO.java	0.20634922	0.3650794
2	BankTransactionDAO.java	0.20634922	0.3650794
3	CapitalAccountDAO.java	0.20634922	0.3650794
4	CapitalTransactionDAO.java	0.20634922	0.3650794
5	CashTransactionDAO.java	0.20634922	0.3650794
6	ConnectionManager.java	0.25	0.38888893
7	CustomerDAO.java	0.20634922	0.3650794
8	InventoryItemDAO.java	0.0	0.0
9	ProductCategoryDAO.java	0.16666667	0.24722223
10	ProductDAO.java	0.20634922	0.3650794
11	PurchaseOrderDAO.java	0.20634922	0.3650794
12	PurchaseOrderDetailDAO.java	0.22916666	0.41805553
13	SalesOrderDAO.java	0.20634922	0.5555556
14	SalesOrderDetailDAO.java	0.1964286	0.38492063
15	SupplierDAO.java	0.20634924	0.5555556
16	ProductCategoryDAO.java	0.16666667	0.24722223

The AccountingSoftware, comprising 16 files, is an arbitrary open source project collected from github.com to further evaluate the behaviour of the developed metric tool (StaSem_C) and a typical static metric. Note that the tool has earlier been tested with deliberate dataset to ensure the output tallies with expected result.

3.3.2 Dataset requiring maintenance efforts - Candidates for re-factoring

A set of poorly cohesive classes were considered. These classes were re-factored to increase their potential maintainability and reusability properties. Effort expended, described as maintenance effort, is measured by the number of changed lines per class (that is addition, deletion or modification made). The following is an example sample code (Person.java class) that was subsequently re-factored and analysed for cohesion:

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

```
class Person {
   public String name;
   public String surname;
   public String email;
   public int yearOfBirth;

   public Person (String name, String surname, String
email, int yearOfBirth)
  {
      this.surname = surname;
      this.name = name;
      this.yearOfBirth = yearOfBirth;
      if(this.validateEmail(email)) {
        this.email = email;
      }
      else {
         throw new Error("Invalid email!");
      }
   }
   public boolean validateEmail(String email) {
//validate email lines
      return test;
   }

   public int calculateAge(int currentYear) {
            return currentYear - this.yearOfBirth;
   }
}// end of Person class
```

In Person.java, the responsibilities appear logical, but have different nature. The Person class is better split into two, which make it more reusable and maintainable. Email validation is not connected with Person behaviour. It is better to have email validation class separated to achieve single responsibility. This will make future modification or reuse easy. The critical question is 'how do we reuse Validate email feature without visiting Person class?' The following is the Maintenance Efforts on Person.java: removing the responsibility of email validation from the Person class and creating a new Email class:

```
// set or initialize effort  =  0
class Person {                  //effort  = 0;
   public String name;
   public String surname;
   public Email email;    //declare Email reference
type;(++effort); effort=1;
   public int yearOfBirth;
   public Person (String name, String surname, Email
email, int yearOfBirth)  // effort  = 2
  {
       this.email = email;

       this.name = name;
       this.surname = surname;
       this.yearOfBirth = yearOfBirth;
      // delete validation test;    effort  = effort + 5;
//effort  = 7
   }
  public int calculateAge(int currentYear) {
            return currentYear - this.yearOfBirth;
   }
   }
```

A new Email class is created to handle validation:

```
class Email {  //create a new class called Email;
//effort = 8
   public String email; // move instance variable
//here;   effort = 9
   public Email(String email){  //  move  the
//validation here  effort = 10
      if(this.validateEmail(email)) {
        this.email = email;
      }
      else {
         throw new Error("Invalid email!");
      }
   }
       public boolean validateEmail(String email) {
//move validate functionality here effort =11
   //validate email lines
      return test;
   }
}
```

The summary of efforts expended is counted as: effort =11. Increasing cohesion results into a system that is easy to create, maintain and reuse. Table 3 presents StaSem_C's behaviour with five re-factored classes.

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

Table 3: StaSem_C's Behaviour with Re-factored Classes

S/N	Carefully selected low cohesion classes	StaSem_C cohesion Experiment	StaSem_C cohesion After Refactoring	Maintenance Effort; working with StaSem_C (count variable: effort)
1	UserSettingService.java	0.45	0.54	5
2	Person.java	0.68	0.83	11
3	FooBar.java	0.67	1.00	9
4	RectangleClass.java	0.83	1.00	6
5	CGPA.java	0.45	0.89	4

It is revealed that each of the listed sample programs costs notable maintenance efforts to enhance their potentials for maintainability and reuse. The metric tool indicates their cohesion status, before and after re-factoring.

4 CONCEPTUAL FRAMEWORK

Quality implies the inherent characteristics of an object, which may set it apart from others. Quality may also mean some degree of excellence. External attributes such as maintainability and reusability are described as developer-oriented quality attributes [11]. The relationship between internal essential attributes (e.g. cohesion and coupling) and external quality attributes is intuitive; for instance, a more complex system would be more difficult to maintain [12]. This section discusses characteristics, types of cohesion, strength and weaknesses of existing cohesion metrics, direction for improvement and relevance to software community.

4.1 Cohesion

Cohesion, from illustration of a class, indicates the degree to which a class has a single, well-focused purpose [13]. Cohesion implies that a component or class encapsulates only attributes and operations that are closely related to one another and to the class or component. Figure 7 and 8 are unified modelling language (UML) class diagram examples of cohesion. Figure 7 depicts low cohesion in that these functionalities appear logical but do not particularly belong together. The Staff class is not the appropriate class to include checkMail or validate emails. These functionalities should be separated into Email class to improve cohesion.

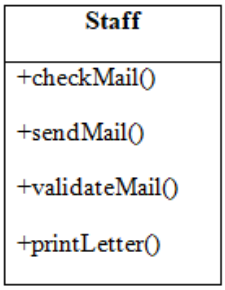

Figure 7: Low cohesion

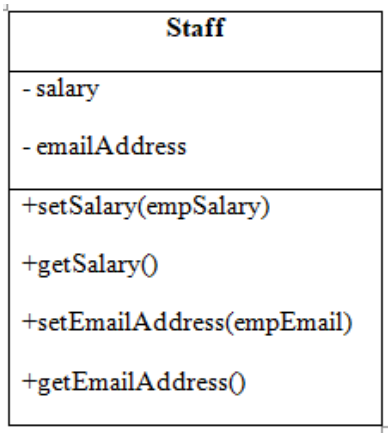

Figure 8: High cohesion

Conversely, the **Staff** class in Figure 8 contains only proper information for setting and getting Staff related data. It does not perform actions that should be managed by another class.

4.1.1 Cohesion views

There are a number of perspectives to cohesion in software. Static and Semantic perspectives are paramount.

(a) Static View

The task of static metrics is to measure or predict what happens when execution of program takes place, and the quantity and complexity of different features of the source code. This is based on the structure, appearance or organization of code elements.

(b) Semantic View

This means externally notable concept that assesses whether the abstraction represented by the module (class in object-oriented concept) can be

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

considered to be semantically whole. The advantage is that it makes cohesion measures more meaningful.

4.1.2 Types of cohesion

The following are the different types of cohesion:

i. Functional Cohesion: This means parts of the module or component are grouped because they all contribute to the module's single well-focused task. On an ordinal scale, this is the best type of cohesion because it fully supports the principle of locality.

ii. Sequential Cohesion: This is when the parts of modules are grouped because the output from one part is the input to the other - X output → Y input; X, Y € same Module.

iii. Communication Cohesion: In this case, parts of the module are grouped because they operate on the same data or contribute to the same data. Sequence is not important in this case.

iv. Procedural Cohesion: This occurs when parts of the module are grouped because a certain sequence of execution is followed by them. The elements of methods are connected by some control flow.

v. Temporal Cohesion: Here, instructions that are executed during the same time span are grouped together.

vi. Logical Cohesion: This is when the module's parts are grouped because they are categorized logically to do the same work, even though they all have different nature.

vii. Coincidental Cohesion: This type is seen in a component whose parts are unrelated to one another. The entity is responsible for a set of tasks which have no good reason for being together except for something like convenience. This is more or less the worst degree of cohesion. It is an indication of poor design [14].

Generally for functional cohesion, each of the methods of a class would manipulate one or more variables. When cohesion is high, it means that the methods and variables of the class are co-dependent and form a logical whole. Highly cohesive classes are much easier to maintain and less frequently changed. Such classes are more usable than others as they are designed with a well-focused purpose. Figure 9 is a diagrammatic description of cohesion occurrence flow modelled from cohesion types.

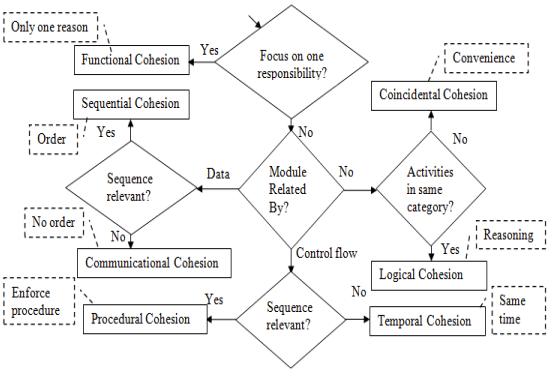

Figure 9: Flow of Occurrence of Cohesion

4.2 Coupling

Coupling is the interaction or relationship between modules. Increase in cohesion intuitively reduces coupling [15]. The more coupled modules are, the harder it is to replace them. For instance, a change in class B breaks class A if class B is tightly coupled to class A, which should not necessarily happen. Refactoring highly coupled design is difficult. 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 highly independent. Loose coupling eases understanding of one class without learning about its neighbours. A class could be changed in isolation with little or no effect on others, thereby improving maintainability. Coupling, also referred to as dependency, has the following important consequences:

a) If a class A depends on a class B, and a system that reuses class A is to be built, class B would be included in the system together with class A, whether or not it serves any purpose.

b) If a class A depends on a class B, and class B is modified. Class A would probably require modification as well. It is indicated that dependencies should be intentionally minimized.

4.3 Software Complexity and Measurements

What determines a software product complexity is the internal attributes. Most internal attributes metrics are development concept dependent (for example traditional or object-oriented concepts).

4.3.1 Traditional measures of complexity

These are metrics used in traditional

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

software development. Notable examples include:

(a) Source Lines of Code (SLOC)

This is typically used to estimate the amount of effort that is required to develop a program. SLOC is a count of non-blank, non-comment lines in the text of the program's source code. This metric is sensitive to logically irrelevant formatting and programming style conventions.

(b) McCabe's Cyclomatic Complexity

This metric measures the number of linearly independent paths through a program module [16]. It indicates complexity of a program and is computed via the control flow graph of a program. Functions with higher Cyclomatic complexity values from 10 and above are hard to understand and maintain. However, this metric could not be used to measure abstractions in OOD. Figure 10 depicts a control flow construct representing independent paths in a typical code.

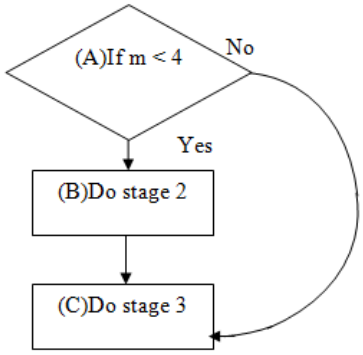

Figure 10 - Control flow construct representing independent paths in a typical code

The complexity of the illustrated code could be computed using the following:

$$V(G) = e - n + 2 \quad (7)$$

Such that:
V(G) = Cylomatic Complexity Graph G
e = number of edges on graph G, and
n = number of nodes on graph G (the nodes represent vertices on a typical graph) ([17]).
In this example,
e = 3 (that is A-C; A-B; B-C)
n = 3 (that is A, B, C), and
V(G) = e – n + 2 = 3 – 3 + 2 = 2

The Cyclomatic complexity is, therefore, 2. In fact, physically tracing the independent paths in the above, we have A-B-C and A-C control flows as the basic paths.

4.3.2 Object-Oriented measures of Complexity (Metrics)

Traditional metrics like cyclomatic complexity (one of the best indicators for system reliability) would not scale well in handling object-oriented software [18]. Traditional approaches emphasize a function-oriented view, where data and procedures are separated. However, modelling the real world in terms of its objects views data and procedure as a single bound unit. However, as object-oriented techniques become more prevalent there is an increasing need for metrics that could correctly evaluate their peculiar properties. Notable metrics are as follows:

(a) Chidamber and Kemerer (CK) metric suite

CK metrics suite is widely known as good indicator of fault proneness [19]. This originally consists of six metrics created to test some specific system characteristics, which are:

(i) Weighted Method per Class (WMC): WMC is useful in predicting maintenance and testing effort. Consider a Class C, with methods $m_1, m_2, \ldots, m_n$ that are defined in the class. Let $c_1, c_2, \ldots, c_n$ be the complexity of the methods such that c_i is the complexity of the method associated in the i^{th} class. The WMC is given as

$$WMC = \sum_{i=1}^{n} c_i, \quad \text{for } i = 1 \text{ to } n \quad (8)$$

(ii) Depth of Inheritance Tree (DIT): DIT is calculated as the maximum length of path from a class to the root class of the inheritance tree. The greater the opportunity of inheriting more methods, the poorer the chance of predicting behaviour.
(iii) Number of Children (NOC): NOC is the number of immediate sub-classes subordinate to a class in the class hierarchy. It is an indicator of the potential influence a class could have on the design.
(iv) Coupling between Objects (CBO): CBO for a class is a count of the number of other classes to which it is coupled. A measure of coupling is useful in determining how complex the testing of various parts of the design would be.
(v) Response for a Class (RFC): RFC is a set of methods that can potentially be executed in response to a message received by an object of that class, and
(vi) Lack of Cohesion in Methods (LCOM): LCOM attempts to find the degree of similarity of methods [19]. Consider a Class C1 with n methods $M_1, M_2, \ldots, M_n$. Let $\{ I_i \}$ be the set of instance variables

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

accessed by method M_i. There are n such sets: $\{I_1, I_2, \ldots, I_n\}$.

Then the following sets are defined:

Let $A = \{ (I_i, I_j) \mid I_i \cap I_j = \phi \}$, (9)

$B = \{ (I_i, I_j) \mid I_i \cap Ij \neq \phi \}$ (10)

If all n sets $\{I_1, I_2, \ldots, I_n\}$ are ϕ then let $A = \phi$

LCOM is defined by:

LCOM = |A| - |B|, if |A| > |B| or 0 otherwise (11)

LCOM = 0 implies that the class is cohesive, and LCOM > 0 implies that the class is not cohesive.

LCOM is an inverse cohesion measure. LCOM metric counts the number of pairs of methods that do not share instance variables. The higher the LCOM, the worse the cohesion of the design, indicating need for refactoring. This means that such a class design should be broken down into two or more classes to promote maintainability or reusability. Chidamber and Kemerer LCOM metric for object-oriented software is effective in identifying the most non-cohesive classes, but is not effective in distinguishing between partially cohesive classes. This means it is not discriminating enough to reveal varying strength of cohesion in classes.

A variation of LCOM by Henderson-Sellers, Constantine, and Graham ([20]) also presented a mathematical model for functional cohesion as follows:

$$LCOM = \frac{\left(\frac{1}{v}\sum_{i=1}^{v} m(Vi)\right) - m}{1 - m} \quad (12)$$

If no variables are accessed, the equation becomes:

$$LCOM = \frac{\left(\frac{1}{v}\sum_{i=1}^{v} m(Vi)\right) - m}{1 - m} = \frac{-m}{1-m}$$

$$= 1 + \frac{1}{m-1} \quad (13)$$

where -

m = number of methods

v = number of variables (attributes)

m(Vi) = number of methods that access variable i (Vi).

LCOM is undefined for m = 1.

(b) Robert Martins Metric Suite

This metrics suite is commonly called package metrics [21]. It attempts to reflect ideal models of dependency and abstraction. It captures some good design principles and also gives a clear description of stable software. It consists of the following:

i) Efferent Coupling (Ce): The number of classes inside the package that depend upon classes outside this package.

ii) Afferent Coupling (Ca): The number of classes outside the package that depend upon classes within the package.

iii) Instability (I): I = Ce / (Ce + Ca) (14)

It implies the package's adaptability to change. The range is [0, 1], I = 0 means absolutely stable package, and I = 1 means absolutely instable package.

iv) Abstractness (A): Abstractness is the ratio of the number of abstract classes (and interfaces) to the total number of classes in the evaluated package.

A = abstract Classes / total Classes (15)

The range is [0, 1]. A = 0 signifies an absolute concrete package, and A = 1 signifies an absolute abstract package.

v) Normalized Distance from Main Sequence (D): Normalized Distance from Main Sequence is the perpendicular distance of a package from the idealized line. It is given as

D = A+ I – 1 (16)

D = 0 depicts a package that is coincident with the main sequence, and

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

D = 1 represents a package which is far away from the main sequence.

4.4 Characterization of Cohesion Metrics in OOD

A summary of extant literature towards improved measures of class cohesion is presented in succeeding subsections.

4.4.1 Basic Static Cohesion metrics and their Scope

The relationship between class cohesion and size was empirically investigated [22]. The metric does not account for connectivity established through user-defined constructors. A class cohesion metric that ignored responsibility assessment of classes was proposed [23]. The inability of Chidamber and Kemerer metric was addressed to yield normalized values [24]. The metric only indicates absence of cohesion but do not present varying strength. A metric that benefited large systems the more was proposed [25]. A static metric that reports the presence of cohesion was also proposed [26]. However, other characteristic interactions e.g. method-method interaction that exist within a class context were not incorporated and also the static metrics do not have capacity to capture the semantics of OO designs.

4.4.2 Basic Semantic Cohesion metrics

A metric that captures domain concepts encoded in comments and identifiers was proposed [14]. A conceptual metric that combined static and semantic views finding textual coherence by analysing textual information expressed in comments and identifiers was implemented [27]. Conceptual Cohesion of Classes (C3) metric analysing comments and identifiers classified to reflect concepts from the domain of the software system was developed [28]. A set of evaluation metrics to measure cohesion for semantic web towards achieving understandability was proposed [29]. Basing cohesion measurement on analysis of comments is insufficient and biased especially for improperly documented software.

4.4.3 Summary of Research Gaps from Literature

It is revealed that static metrics considered majorly variable-method interactions, which do not conceptually represent a class design as a set of responsibilities that approximate the domain concept. The extant semantic metrics and its hybrid leveraged on extracting information from comments and identifiers to represent concepts of the problem and solution domains. However, this is plagued with the assumption that commenting rules are followed in code. Then, if there is problem of comment attribution in codes, the extant metrics would underperform ([28, 29, 27]). Therefore, this study proposed to complement the existing static cohesion metrics by introducing measurement for hidden data attributes (as opposed to studying comments), which semantically represents a test for a conceptually cohesive class that stands as a clear indicator for good abstraction.

5. CONCLUSION AND RECOMMENDATIONS

In conclusion, the developed metric model, which captures structure and the domain concept of object-oriented design, provides a quantitative means to adequately measure and control product quality. Researcher would find the mathematical model as a useful inspiring construct. StaSem_C is recommended for developers to proactively manage design complexity, which would increase software maintainability and reusability.

5.1 Contributions to Knowledge

This work carried out a thorough empirical evaluation of how cohesion affects the maintainability property of software. Static cohesion is considered insufficient and the existing semantic cohesion assumed that comments would always be available for analysis of cohesion in software. This study combined static and semantic views such that the semantic aspect finds the degree of support for data hiding and single abstraction which greatly promotes the idiosyncrasy of object-oriented design. This study contributed to the academic discourse as follows:

i. Development of an improved mathematical model for functional cohesion metric that represents a measurement based on the set of responsibilities and domain concept a class exhibits rather than only variable-method interactions.
ii. Development of an automated cohesion metric tool that could be used by developers to predict software maintainability and reusability properties.

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

5.2 Further Work

Future research efforts could be directed towards confirming and refining coupling and cohesion measures and models. Pattern of interaction among software elements could be further considered.

REFERENCES:

[1] B. Isong and E. Obeten, "A systematic review of the empirical validation of object-oriented metrics towards fault-proneness prediction", *International Journal of Software Engineering and Knowledge Engineering,* Vol. 23, No. 10, 2013, pp. 1513–1540.

[2] E. U. Okike and A. Osofisan, "An evaluation of Chidamber and Kermerer's lack of cohesion in methods metric using different normalization approaches", *Afr. J. comp. & ICT,* Vol. 1, No. 2, 2008, pp. 35- 43.

[3] E. U. Okike and M. Rapo, "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),* Vol. 13, No. 12, 2015, pp. 140-149.

[4] S. Tiwari and S. S. Rathore, "Coupling and cohesion metrics for object-oriented software: a systematic mapping study", ISEC'18: Innovations in Software Engineering Conference, Hyderaba. ACM, New York, NY, USA, 2018, Retrieved from https://doi.org/10.

[5] Y. Ahn, J. Suh, S. Kim, and H. Kim, "The software maintenance project effort estimation model based on function points", *Journal of Software Maintenance Evolution: Research and Practice,* Vol. 15, 2003, pp. 71-85.

[6] J. Viljanen, "Measuring software maintainability", Master's Thesis, *Aalto University School of Science, Es*poo, August10, 2015.

[7] T. Tahir, G. Rasool, and C. Gencel, "A systematic literature review on software measurement programs. *Information and Software Technology,* Vol. 73, 2016, pp. 101–121.

[8] M. Ezran, M. Morisio, and C. Tully, "Practical Software Reuse", Springer, 2002, 374.

[9] J. A. Wang, "Towards component-based software engineering", Department of Computer Science and Information Systems University of Nebraska, Kearney, 2000.

[10] J. Dhanvani, "Difference between cohesion and coupling", 2013, Retrieved from http://freefeast.info/difference-between/difference-between-cohesion-and-coupling-cohesion-vs-coupling/

[11] P. Berander., L. Damm, J. Eriksson,, T. Gorschek, K. Henningsson, P. Jönsson, S. Kågström, D. Milicic, F. Mårtenssonn, K. Rönkkö, and P. Tomaszewski, "Software quality attributes and trade-offs", Blekinge Institute of Technology. 2005.

[12] M. Ribeiro, R. Q. Reis, and A. J. Abelalm, "How to automatically collect object oriented metrics: A study based on systematic review", Latin American Computing Conference (CLEI), 2015, 1–12.

[13] R. C. Martin, "*Clean code: A handbook of agile software craftsmanship* (1st ed.)." Upper Saddle River, NJ, Boston: Prentice Hall, 2012.

[14] S. M. Chandrika, E. S. Babu, and N. Srikanth, "Conceptual cohesion of classes in object oriented systems", *International Journal of Computer Science and Telecommunications*, Vol. *2*, No. 4, 2011.

[15] H. Izadkhah, and M. Hooshyar, "Class cohesion metrics for software engineering: A critical review", *Computer Science Journal of Moldova,* Vol. 25, No. 1, 2017, pp. 44-74.

[16] R. M. Redin, M. F. Oliveira, L. Carro, L. C. Lamb, and F. R. Wagner, "Software quality metrics and their impact on embedded software", Budapest, 2008, pp. 68-77.

[17] A. H. Watson, and T. J. McCabe, "Structured testing: A testing methodology using the cyclomatic complexity metric", NIST Special Publication 500-235 Computer Systems Laboratory National Institute of Standards and Technology, Gaithersburg, 1996, Retrieved from www.mccabe.com /pdf/mccabe-nist235r.pdf.

[18] M. Kaur, and R. Kaur, "Improving the design of cohesion and coupling metrics for aspect oriented software development", *International Journal of Computer Science and Mobile Computing,* IJCSMC, Vol. 4, No. 5, 2015, pp. 99 – 106.

[19] S. R. Chidamber, and C. F. Kemerer, "A metrics suite for object oriented design", *IEEE Transactions on Software Engineering,* Vol. 20, No. 6, 1994, pp. 476-493.

[20] B. Henderson-Sellers, L. Constantine, and I. Graham, "Coupling and cohesion: Towards a

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

valid metrics suite for object-oriented analysis and design", *Object Oriented Systems,* 3, 1996, 143-158.

[21] R. C. Martin, "Object oriented design metrics", 1994, Retrieved from http://www.objectmentor.com/rescources/articles/oodmetric.pdf

[22] M. F. Shumway, "Measuring class cohesion in java, (Masters Dissertation)", Computer Science Department, Colorado State University, Technical Report CS-97-113, 1997.

[23] L. Badri, and M. Badri, "A proposal of a new class cohesion criterion: An empirical study", *Journal of Object Technology, Chair of Software Engineering, JOT*, ETH Zurich Publishing. Vol. 3, No. 4, 2004.

[24] E. U. Okike, "A proposal for normalized Lack of Cohesion in Method (LCOM) metric using field experiment", *IJCSI International Journal of Computer Science Issues*, Vol. 7, No. 4, 5, 2010.

[25] J. Michura, M. Capretz, and S. Wang, "Extension of object-oriented metrics suite for software maintenance", *Hindawi Publishing Corporation ISRN Software Engineering,* (276105), Vol. 14, 2013.

[26] S. Mal, and K. Rajnish, (2014). "New class cohesion metric: An empirical view", *International Journal of Multimedia and Ubiquitous Engineering,* Vol. 9, No. 6, 2014, pp. 367-376.

[27] K. Rakesh, and S. Sushumna, "Implementation of measuring conceptual cohesion", *International Journal of Computer Science and Information Technologies, (IJCSIT),* Vol. 3, No. 3, 2012, pp. 4237-4243.

[28] K. K. Girish, "Conceptual Cohesion of Classes (C3) Metrics", *International Journal of Science and Research (IJSR)*, Vol. 3, No. 4, 2014, pp. 2319-7064.

[29] L. Liao, G. Shen, Z. Huang and F. Wang, "Cohesion metrics for evaluating semantic web ontologies", *International Journal of Hybrid Information technology*, Vol. 9, No. 11, 2016, 369 -380.