

**SOFTWARE PRODUCT QUALITY ASSESSMENT
USING SCOPED CLASS COHESION METRIC (SCCM)**

RAPHAEL NGIGI WANJIKU

MASTER OF SCIENCE

(Software Engineering)

**JOMO KENYATTA UNIVERSITY OF
AGRICULTURE AND TECHNOLOGY.**

2017

**Software Product Quality Assessment Using Scoped Class Cohesion
Metric (SCCM)**

Raphael Ngigi Wanjiku

**A Thesis Submitted in Partial Fulfillment for the Degree of Master
of Science in Software Engineering in the Jomo Kenyatta University
of Agriculture and Technology.**

2017

DECLARATION

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

Signature: _____ Date: _____

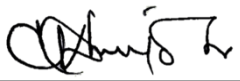
Raphael Ngigi Wanjiku

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

Signature: _____ Date: _____

Dr. George Okeyo

JKUAT, Kenya

Signature:  _____ Date: _____

Dr. Wilson Cheruiyot

JKUAT, Kenya

DEDICATION

I dedicate this thesis to my wife Ann and son Caleb. You were the inspiration behind this research.

ACKNOWLEDGEMENTS

I would like to thank all those who supported me in any way during the writing of this thesis. Special acknowledgement goes to Dr. George Okeyo and Dr. Wilson Cheruiyot for their insightful opinions, criticism and guidance. Special thank you to Dr. Michael Kimwele for his efforts and encouragement during the research process.

TABLE OF CONTENTS

DECLARATION.....	ii
DEDICATION.....	iii
ACKNOWLEDGEMENTS	iv
TABLE OF CONTENTS	v
LIST OF TABLES	x
LIST OF FIGURES	xi
LIST OF APPENDICES	xiv
LIST OF ABBREVIATIONS	xv
ABSTRACT	xvii
CHAPTER ONE	1
INTRODUCTION.....	1
1.1 Background Information	1
1.2 Problem Statement.....	5
1.3 Justification.....	5
1.4 Research Objectives.....	6
1.4.1 General Objective	6
1.4.2 Specific Objectives	6
1.5 Research Questions.....	7
1.6 Scope of the Study	7
1.7 Structure of Thesis	7

CHAPTER TWO	9
LITERATURE REVIEW	9
2.1 Software Quality	9
2.1.1 Maintainability.....	10
2.1.1.1 Scalability	10
2.1.1.2 Modifiability	12
2.1.1.3 Testability	12
2.1.2 Performance	13
2.1.3 Security	14
2.1.4 Availability	14
2.1.5 Integration.....	16
2.1.6 Understandability.....	16
2.1.7 Usability.....	17
2.2 Class Members.....	19
2.2.1 Attribute Members	19
2.2.2 Member Functions	20
2.3 Software Metrics.....	22
2.4 Class Cohesion	24
2.3.1 Functional perspective	24
2.3.2 Sequential perspective	25
2.3.3 Communicational perspective	26
2.3.4 Procedural perspective.....	27
2.3.5 Temporal perspective.....	28

2.3.6 Logical perspective	29
2.3.7 Coincidental perspective.....	30
2.4 Existing Metrics.....	31
2.4.1 Lack of Cohesion Metrics-LCOM 1 and LCOM 2	31
2.4.2 Lack of Cohesion 3 Metric (LCOM3)	34
2.4.3 Lack of Cohesion 4 Metric (LCOM4).....	36
2.4.4 Lack of Cohesion Metric 5(LCOM5)	38
2.4.5 Tight Class Cohesion and Loose Class Cohesion Metrics	40
2.4.6 Relative lack of Cohesion in Methods (RLCOM).....	42
2.4.7 Coh Metric	44
2.4.8 DCD and DCI Metrics	45
2.4.9 Class Cohesion (CC) Metric.....	47
2.4.10 Distance Design-based Direct Class Cohesion (D3C2) Metric.....	49
2.5 Research Gaps	51
2.6 SCCM Conceptual Framework.....	53
2.7 Conclusion	54
CHAPTER THREE	55
METHODOLOGY	55
3.1 Introduction.....	55
3.2 Research Design	55
3.2.1 Sample Population	55
3.2.2 Sample Size	56
3.2.3 Data Collection Methods	56

3.2.4 Scoped Class Cohesion Metric	57
3.2.5 SCCM Algorithm	58
3.2.6 Data Collection Instrument.....	59
3.2.6.1 SCCM Tool Development Process.....	59
3.2.6.1.1 SCCM Tool Specification.....	60
3.2.6.1.2 SCCM Tool Design	63
3.2.6.1.3 SCCM Tool Implementation and Testing.....	67
3.2.6.1.4 SCCM Tool Operation and Maintenance	69
CHAPTER FOUR.....	70
EXPERIMENT, RESULTS AND DISCUSSION	70
4.1 Introduction.....	70
4.2 Experiment Design	70
4.3 SCCM Raw Collected Data.....	71
4.4 Experiment Processed Data	76
4.4.1 Geometric Means on SCCM/COH	77
4.4.2 Pearson’s Coefficient on SCCM/COH	79
4.5 Results	83
4.5.1 Effects of Public Methods on SCCM Value.....	83
4.5.2 Effects of Total Class Variables on SCCM Values.....	87
4.5.3 Effects of Public Variables on SCCM Values.....	90
4.5.4 Effects of Private Variables on SCCM Values.....	94
4.5.5 Effects of Local Variables on SCCM Values	96
4.6 Discussion.....	99

4.6.1 Discussion of Results of the Effects of Public Methods.....	99
4.6.2 Discussion of Results of the Effects of Total Class Variables	102
4.6.3 Discussion of Results of the Effects of Public Variables	103
4.6.4 Discussion of Results of the Effects of Private Variables	107
4.6.5 Discussion of Results of the Effects of Local Variables	107
CHAPTER FIVE	109
SUMMARY, CONCLUSIONS AND RECOMMENDATIONS	109
5.1 Summary and Conclusion.....	109
5.2 Recommendations for Future Work	111
REFERENCES.....	112
APPENDICES	119

LIST OF TABLES

Table 2.1: AccountDialog Class matrix.	50
Table 4.1: SCCM and COH values from the Java Cluster	74
Table 4.2: SCCM and COH values from the C++ Cluster	74
Table 4.3: SCCM and COH values from the JavaScript Cluster	75
Table 4.4: SCCM and COH values from the PHP Cluster	75
Table 4.5: SCCM and COH geometric mean values from the Java Cluster	77
Table 4.6: SCCM and COH geometric mean values from the C++ Cluster	78
Table 4.7: SCCM and COH geometric mean values from the JavaScript Cluster	78
Table 4.8: SCCM and COH geometric mean values from the PHP Cluster	79
Table 4.9: Pearson coefficient values on SCCM matrix of C++ systems	80
Table 4.10: Pearson coefficient values on SCCM matrix of Java systems	81
Table 4.11: Pearson coefficient values on SCCM matrix of C++ systems	82
Table 4.12: Pearson coefficient values on SCCM matrix of PHP systems	82

LIST OF FIGURES

Figure 2.1: An illustration of functional cohesion view	25
Figure 2.2: An illustration of sequential cohesion view.	26
Figure 2.3: A diagram of communicational cohesion perspective	27
Figure 2.4: A diagram showing procedural cohesion perspective	28
Figure 2.5: A diagram showing temporal cohesion perspective.....	28
Figure 2.6: A diagram showing logical cohesion perspective.	29
Figure 2.7: Code sections showing three unrelated methods.....	30
Figure 2.8: An illustration of a class components interaction	32
Figure 2.9: A code snippet used in calculating LCOM1	33
Figure 2.10: An illustration of the LCOM 3 Metric components interaction.....	35
Figure 2.11: An illustration of the LCOM 4 Metric components interaction.....	37
Figure 2.12: An illustration of LCOM 5 metric components	39
Figure 2.13: An illustration of TCC and LCC metrics.	41
Figure 2.14: Illustration of RLCOM.....	43
Figure 2.15: An illustration of the COH Metric components interaction.....	44
Figure 2.16: An illustration of the DCD and DCI Metric components interaction ..	46
Figure 2.17: An illustration of the CC Metric components interaction	48
Figure 2.18: SCCM conceptual framework	53
Figure 3.1: SCCM algorithm	59

Figure 3.2: Incremental iterative development (Adapted from Sommerville, 2015)	60
Figure 3.3: Activity diagram for the SCCM software	62
Figure 3.4: An Abstract model for the SCCM software	63
Figure 3.5: Pipe and Filter architecture pattern SCCM software	65
Figure 3.6: High-level architecture of the SCCM software	66
Figure 3.7: Web interface for the SCCM software	68
Figure 3.8: The output console of SCCM calculated values	69
Figure 4.1: A screen shot of the interface of the SCCM software tool	72
Figure 4.2: A screen shot showing the uploaded class, the compressed code (minified class) and the console output	73
Figure 4.3: Influence of public methods on the SCCM/COH values; a case of Java classes	83
Figure 4.4: Influence of public methods on the SCCM/COH values; a case of JavaScript classes	84
Figure 4.5: Influence of public methods on the SCCM/COH values; a case of PHP classes	85
Figure 4.6: Influence of public methods on the SCCM/COH values; a case of C++	86
Figure 4.7: Influence of total class variables on the SCCM/COH values; a case of PHP	87
Figure 4.8: Influence of total class variables on the SCCM/COH values; a case of C++	88

Figure 4.9: Influence of total class variables on the SCCM/COH values; a case of Java.....	89
Figure 4.10: Influence of total class variables on the SCCM/COH values; a case of JavaScript.	90
Figure 4.11: Influence of public variables on the SCCM/COH values; a case of C++.....	91
Figure 4.12: Influence of public variables on the SCCM/COH values; a case of Java.....	92
Figure 4.13: Influence of public variables on the SCCM/COH values; a case of JavaScript.	93
Figure 4.14: Influence of public variables on the SCCM/COH values; a case of PHP.....	94
Figure 4.15: Influence of private variables on the SCCM/COH values; a case of JavaScript.	95
Figure 4.16: Influence Of Private Variables On The SCCM/COH Values; A Case Of PHP.....	96
Figure 4.17: Influence of local variables on the SCCM values; a case of Java.....	97
Figure 4.18: Influence of local variables on the SCCM values; a case of JavaScript.....	98
Figure 4.19: Influence of local variables on the SCCM values; a case of PHP.....	99
Figure 4.20: A diagram illustrating the Law of Demeter.-Adapted from Lostechies.....	102
Figure 4.21: An illustration of singleton pattern in OO class design-Adapted from tutorialspoint.com.....	105

LIST OF APPENDICES

Appendix A: Abstract of publication on IJCSI.....	119
--	-----

LIST OF ABBREVIATIONS

SCCM	Scoped Class Cohesion Metric
LCOM	Lack of Cohesion Metric
PO	Parameter Occurrence
TCC	Tight Class Cohesion
LCC	Loose Class Cohesion
RLCOM	Relative Lack of Cohesion in Methods
DC_D	Degree of Cohesion Direct
DC_I	Degree of Cohesion Indirect
D₃C₂	Distance Design-based Direct Class Cohesion
PM	Public Method
PRM	Private Method
PA	Public Attribute
PRA	Private Attribute
PO	Public Occurrence
PRO	Private Occurrence
TPC	Total Public Cohesion
TPRC	Total Private Cohesion
PC	Public Cohesion
PRC	Private Cohesion
TC	Total Cohesion
LV	Local Variable
COH	Class Cohesion
HTML5	Hypertext Markup Language 5

CSS3	Cascading Style sheet 3
OOP	Object Oriented Programming
IDE	Integrated Development Environment
PHP	Hypertext Preprocessor

ABSTRACT

Class Cohesion is an important software quality that can be used to improve software development process and assess the software product: process merit assessment and dependable software product. Many Class cohesion metrics measuring the relationship between methods and attributes have been developed and extensively researched. However, the use of relationships among attributes in measuring class cohesion from class scopes has been ignored and the effects of local variables on class cohesion need to be factored in the measurements. This thesis presents a new class cohesion metric that uses attributes relationships within class scopes. The data was collected from JavaScript, PHP, C++ and Java cluster classes using the Scoped Class Cohesion Metric (SCCM) software tool. The browser accessible JavaScript tool allows the user to select any cluster valid class, scans for the methods and attributes and output a metric value on the browser console. The analysed values of Scoped Class Cohesion Metric (SCCM) and Cohesion Metric (COH) showed that development of large classes with many attributes and methods possess low class cohesion compared to the small classes. Moreover, as the number of local variables increase within a class, the value of cohesion decreases and they should therefore be introduced or used only and only when necessary. This makes the software product more understandable, it improves class testing as well as easier maintenance consequently leading to an overall good quality software product.

CHAPTER ONE

INTRODUCTION

1.1 Background Information

The quality of a software product can be traced from its process and the set software metrics that measures its effectiveness in fulfilling customers' requirements and adherence to acceptable development standards. One of the software metrics is cohesion. Cohesion refers to the degree of relatedness among modules of a software product (Kaur & Kaur, 2013). Cohesion measures the usage of a module and its elements within another module in terms of imported or exported functionality. According to Briand, Daly and Wust (1997), high cohesion within a module makes it easier to develop, facilitates comprehension (Dasari & Vasanthakumari, 2011), enhances maintenance, testing (Badri et al., 2011), components reusability (Rosenberg and Hyatt, 2012), improved process merit assessment (Patidar, 2013) and reduces fault-proneness ensuring independent components with less complexities (Pena, 2006).

Cohesion has been a subject of study for almost four decades with Yourdon and Constantine (1978) classifying measures on an ordinal scale for component cohesion to normalized Hamming Distance cohesion metric (Counsell *et al.*, 2006). Cohesion is measured in terms of the degree to which methods and attributes (fields or variables) of a class belong together through their interactions. Interaction between class elements can occur in three ways as described by Dallal (2010):

- (i) Method to attribute interaction –this interaction occurs when an attribute type presented in a class matches its value in the method parameter or the return of a method. For example given a class with method m1,m2 and m3 with two attributes a1 and a2, the presence of attribute a1 may be detected in method m1 while that of a2 may be noted in method m2.This would then give an interaction of 2 attributes in two methods.
- (ii) Method to method interaction –this interaction occurs when two or more methods and their parameters or returns share the same attribute type. For example, if a class has method m1 and method m2 and there is an attribute a1 that is present in both methods, then by the fact that this attribute appears in both methods then that is considered as a method to method interaction via the attribute type of a1.
- (iii) Attribute to attribute interaction – this interaction occurs when two or more attributes share the same type in a method. For example, if a class has two string attributes a1 and a2 and they happen to appear within one method m1 or in another method m2 of the same class, then by the appearance of a1 or a2 within the same method m1 or m2, then it is considered as an attribute to attribute interaction of that class.

High class cohesion manifests a well-designed class (Scott, 2009). A theoretically validated cohesion metric is characterised by four properties (Briand et al., 2006):

- (i) Non-negativity and normalization- cohesion measure should be confined within a range that involves a non-negative value and a given maximum range i.e. a range [0,max]

- (ii) Null value and maximum value- cohesion of a class should hold a value of zero (0) if the class is non-cohesive and a maximum value if the class has all possible interactions between the class elements (Dallal, 2010).
- (iii) Monotonicity- cohesion within a class cannot be reduced by adding cohesive interactions to a module. For example if a class has five cohesive modules and two extra modules are added to the class that contains the five modules then the resultant cohesion should be the same as that of the five despite the addition of the final class modules.
- (iv) Cohesive modules- when two or more modules are combined through a union then the two or more unrelated modules should not decrease the new formed module cohesion.

Lack of Cohesion Metrics (LCOM) 1&2 (Chidamber & Kemerer, 1994) introduces calculation of lack of cohesion within a class using pairs of methods that normally calculates the difference in method pairs for lack of cohesion metric 1 (LCOM1) and a difference of lack of cohesion metric 1 (LCOM1) to the number of similar method pairs in calculating lack of cohesion metric 2 (LCOM2). Lack of cohesion metric 3 (LCOM3) (Li & Henry, 1993) uses undirected graph and its value calculation is based on connected components of the edges and vertices adding up class cohesion. Lack of cohesion metric 4 (LCOM4) extends lack of cohesion metric 3 (LCOM3) by adding an edge between pairs of methods while lack of cohesion metric 5 (LCOM5) (Henderson-Sellers, 1996) sums up definite attributes in its calculation. Other metrics include tight class cohesion (TCC) and loose class cohesion (LCC) by Bieman and

Kang (1995) which uses directly and transitively connected pairs of methods respectively.

Relative lack of cohesion metric (RLCOM) calculates the ratio of non-similar method pairs to the total number of similar pairs. Class cohesion (COH) calculates the ratio of total number of attributes within a class to the product of the total number of attributes and methods of that given class. Direct class direct (DC_D) and Direct cohesion indirect (DC_I) metrics calculate the fraction of directly connected pairs of methods direct cohesion direct (DC_D) while the direct cohesion indirect (DC_I) calculates the fraction of both directly and indirectly connected method pairs.

Class cohesion metric (CC) calculates a fraction of the number of shared attributes to the number of distinct reference attributes by the total number of method pairs identified within a class. Distance design-based direct class cohesion (D3C2) normally calculates the total cohesion of a class by adding up the value of cohesion from attributes to attributes interaction, method to attributes interaction and method to methods interaction.

Despite the development of the many class cohesion metrics discussed in this chapter, the interaction of various attributes (within global and local scopes) needs to be factored in measuring cohesion within a class. In a close attempt to address the interaction between attributes within a class Rajnish (2014) conducted a study that gave a clear interaction measure but failed to address the scoping issue within a class during the interaction.

A class scope refers to a region within a class text where a class member can be referenced without necessarily qualifying its name. (Msdn, 2016). Scope of the class elements is done to give visibility of a given member element. In this research, a review of the class cohesion metrics that use attributes-attributes and attributes-methods classification is done and a metric is developed that addresses the use of attributes in the global and local scopes of a class.

1.2 Problem Statement

The use of scopes and locally defined class attributes is an overlooked facet of consideration in the measurement of class cohesiveness. Majority of class cohesion metrics focus on class methods and attributes in general, for example Chidamber and Kemerer (CK) metrics that measure lack of class cohesion and Class Cohesion (COH) metric that measure cohesion within methods for unique types. The interaction of locally defined attributes with other attributes and scoping of class components has not been addressed by the current existing class cohesion metrics, an element that is addressed in this research.

1.3 Justification

Class cohesion is an important consideration in the design of a class. It describes the binding of the elements defined within a class (Chandrika et al., 2011). The use of class attributes in different scopes is an important consideration in measuring effective relatedness among attributes and methods of a class (Bonja and Kidanmariam, 2006). While there are other class cohesion metrics that have been used in determining software structural quality, the use of Scoped Class Cohesion Metric (SCCM) is

important in showing where and how class cohesion is influenced by the class member elements in the various class scopes. This in the long run helps developers in evaluating various software quality attributes among them reusability, testability and understandability of where to increase or reduce data access in the various scopes of the class.

1.4 Research Objectives

1.4.1 General Objective

The general objective of this research was to investigate the use of scopes within a class in measuring class cohesion. It was done with an aim to assess the software product quality using scoped class cohesion metric (SCCM).

1.4.2 Specific Objectives

The specific objectives of this research were to:

- i) Investigate the different class cohesion metrics that use high level design in a class.
- ii) Investigate the class members' effects on class cohesion.
- iii) Formulate the scoped class cohesion metric.
- iv) Evaluate the effectiveness of the scoped class cohesion metric (SCCM) in the software quality assessment.

1.5 Research Questions

- i) What are the various class cohesion metrics that exist at high level design for a class?
- ii) What are the factors behind the axiomatic formalism of the scoped class cohesion metric?
- iii) What are the parameters involved in formulating the scoped class cohesion metric?
- iv) How does scoped class cohesion metric (SCCM) perform in measuring class cohesion compared to the existing class cohesion metrics in software quality assessment?

1.6 Scope of the Study

This research will be investigating class cohesion metrics based on the class scopes. The various class members' relationship will be investigated and how each member's role affects the cohesion values. It also discusses the development of the scoped class cohesion metric (SCCM) tool that will be used for data collection and calculation of the scoped cohesion metric (SCCM) values on the various systems classes.

1.7 Structure of Thesis

This thesis is presented in chapters; Chapter two introduces software quality and quality attributes discussing class cohesion from its various perspectives ranging from

functional to coincidental cohesion. It further discusses the existing metrics and the research conceptual framework.

Chapter three outlines the methodology; the research design, the metric tool to be created, the tool's development setup and each stage of the development process as well the formulation of the scoped class cohesion metric (SCCM). Chapter four discusses the descriptive statistics to be used in the experiment, the experimental raw and processed data, experimental results and their interpretation which is done through the results discussion.

Lastly, chapter five gives the summary of the research work, deduced conclusion and recommendations for extension of this work.

CHAPTER TWO

LITERATURE REVIEW

2.1 Software Quality

Software quality refers to the desirable attributes of a software product (Houston, 2015) and the degree to which the product is able to meet the specified requirements to fulfil the user's needs (IEEE, 2016). Whenever a software is said to be of high quality then it must reflect how well it is able to conform to the standards of design and development in meeting the needs of the user as per the specification. These specified requirements can be functional or non-functional. Software quality can be evaluated from either a defect management approach or the attributes approach reflecting on the failure to addressing end-user requirements and desirable features of the end product to the stakeholders respectively. Software quality can be categorized into functional, structural and process qualities.

According to Chappell (2012), functional software quality evaluates the correctness of tasks performed by the software as intended to the end-users. Its attributes include meeting the specified requirements, creating software with few defects, good and enough performance and ease of learning and use.

Process software quality refers to the value received by the users, sponsors and the development teams. Some of the attributes include; meeting delivery dates, meeting budgets and repeatable development process that guarantees quality.

Structural software quality refers to the well structuredness of the code. A quality software product's attributes gives it the benchmarks that describes its intended behaviour as well as its operational environment. A well-structured class is said to have the following quality attributes;

2.1.1 Maintainability

According to Advoss (2015), maintainability refers to the ability of a software product to conform to new changes. The changes may be as a result of addition of new features and correction of emerging errors during its maintenance. Maintainability of a class is based on its scalability, testability (Barbacci, 2004) and modifiability (sqa, 2010).As cited by Gueye, Badri and Badri (2008), highly cohesive components have been observed to have high maintainability.

2.1.1.1 Scalability

This refers to the ability of extending the system architecture. A scalable software product is one that can be able to add more functionality to the existing modules of the system. A high quality software should allow extension of its functionality in order to meet the changing needs of the user as well as the expectations (Msdn, 2015). Scalability of a software can be observed from various views;

- i) **Request Load:** In a case where a software is intended to receive a given number of requests for example 5000 requests and ends up getting more requests for example 15000 requests, then the extra number of requests needs to be catered for as well as to ensure that users expectations are met. In the working environment

of a software, it is expected that as the load increases then the throughput of the software product should remain relatively accommodating. Therefore a quality software should be able to handle the extra amount of load of requests that is added to the existing system. This request load handling is normally defined and actualized in the classes and therefore a good design of the system classes should ensure that the proper requests load is addressed.

- ii) **Simultaneous connections:** Most systems in one way or another are designed to support multi user environments. A multi user environment is one that allows multiusers to connect to the software through multiconnections. This functionality is defined and implemented in the software classes. A good example is a software that allows network users by an internet service provider (ISP). In this case, the provider should be able to monitor all the connected clients and the different activities that are consuming the allocated data bandwidth. As the number of users grows, the software should also keep up in supporting these new multi connections. In such a case, the software is said to be of high quality compared to another that cannot support these newly created connections.
- iii) **Data Size:** The amount of data that a software interacts with affects the quality of software depending on the data to be used in the processing. A software product that is tasked with processing huge amounts of data require lots of processing memory from the housing computer system. It is therefore important to consider proper design of the system classes and ensure that they use the right data structures and is supplemented with the right number of methods and variables. A software that is highly scalable should be able to work efficiently as the data grows without compromising on the performance of the system.

2.1.1.2 Modifiability

Due to the changing needs of the customers, a high quality software should be able to accommodate changes whenever they have been presented for implementation on the system. A software classes should allow modification of its members with ease and it is then said to be of high quality. If a class is flexible to accommodate changes then it becomes less expensive to make those changes. Modifiability gives a measure of how easy it may be in changing a software to cater for new requirements (Gorton, 2011).

2.1.1.3 Testability

This refers to a measure of how easy it is to create some test criteria for a given system and its composing modules (Msdn, 2016). Testing is normally done with a goal of finding out if the system meets the criteria of execution as per the user's specifications. Classes that have been well designed should be highly testable and high class cohesion ensures that this attribute is catered for in a given software product.

When a developer is conducting tests on the class, a couple of issues need to be addressed. Firstly, a proper test plan of the class need to be prepared; this involve coming up with the mock objects to be used in the testing and the elements of the testing process ought to be constructed in a simple and a structured manner. Secondly, the developer should ensure that the testing process is automated as possible on the areas that requires the user's interaction. Thirdly, the developer should understand or improve the understandability of the input elements into the software product and the expected output of the processed data. This understandability helps in minimizing the input/output inconsistencies.

2.1.2 Performance

Performance refers to a measure of the amount of work that an application must perform in a given time and which must be met during the operation (Gorton, 2011). Some softwares are critical in performing some operations for example, if a life support machine software was to cause any delay in executing a specific instruction due to poor performance, then the patient's life would be in danger as a result of real-time constraints. This could also be a case for the space navigation system where if a software was to cause a delay in estimating the right trajectory, a very huge mileage deviation could consequently occur.

Performance in systems is defined within the classes that have implemented it and can be measured through the following facets:

- i) **Throughput:** This refers to a measure of the amount of work performed by an application in a unit time. This is normally measured in transactions per second.
- ii) **Response time:** This refers to a measure of time delay experienced in executing a transaction by a software. If a software takes less time in giving a response to the user then it is said to be highly responsive and one that possesses high quality.
- iii) **Deadlines:** This is normally associated with batch systems where a given batch process is supposed to take a given time interval in order to give a measurable expected result. If a software is supposed to take one minute in executing a payroll batch transaction process for staff and the observed time is two minutes, then this software is said to be having poor performance and it is not able to meet the set deadlines.

2.1.3 Security

Security refers to a software product's capability to reduce the chances of any action that can negatively affect the system. These negative effects can include stealing of information, siege of the system or its components. Factors that affect a systems security are its availability, confidentiality and its data integrity. It is a measure that is based on the following attributes as described by Gorton (2011);

- i) **Authentication:** this verifies the identity of the software users and any other application that is connected to the software.
- ii) **Authorization:** this allows an identified user or any application that accesses the software in accessing resources provided via the software.
- iii) **Encryption:** this codes the messages sent or received by the software.

When the developers are addressing the above software security issues, they should be able to design system classes that allow users to authenticate in the best recommended methods as well as get the right authorization in accessing the system resources.

2.1.4 Availability

This is a software quality attribute that measures the reliability of a software by the users (Msdn, 2016). It represents the proportion of time that a software is said to be functioning and working. Availability of a software system should be 100% which represents an all uptime availability. A high quality software should have minimal failures as much as possible and the length of time of unavailability is normally measured as the interval between the failure detection and when the system restarts. This time interval is known as mean to recover.

When developers are designing and implementing a software product they need to cater for system errors, malicious attacks that may be directed by unauthorized users, infrastructural problems that may arise when the software is operational and increase in the software load in the class designs. When designing the classes the following can be done as described on Msdn (2016);

- i) A failover mechanism can be implemented to ensure availability of an alternate way of accomplish the intended task. For example in a case of a software that is connected to a network with two differentiated links, there should be a failover so that in case one fails then the unaffected link picks up automatically. For example in a class that connects a mobile phone application to the internet or even a locally connected server, several methods can be defined by the developer to cater for the several available links and during the connection if one method does not return the connection the other existing connection methods can be polled until a connection is established.
- ii) The classes can be designed to handle unexpected or unknown exceptions. If the unknown exceptions are used within a class, they can be customized to detect any unexperienced behaviours by the system. This is in the long run can assist in detecting anomalies such as denial of service attacks as well as any malicious activities that may be directed to a system through the software.
- iii) The classes can also be designed in a way that bugs and faults are detected before the software enters into an unrecoverable state. This design would involve proper error handling that detect application failures.

2.1.5 Integration

This is concerned with the ease in merging or connecting two or more software systems (Gorton, 2011). This can be done through data integration or through the use of application programming interface (API). A software product that is able to work well with other software makes it easier for data to be exchange internally (within the software product between its modules) or externally (with the other softwares). When developers are developing interoperable classes they should ensure that proper standards and protocols have been taken into consideration. These protocols may include data formats and interfaces.

For proper interoperability there exist several key issues that may need to be observed (Msdn, 2016); firstly, the difference in data formats. This can be handled by data translation as well as use of canonical data model for handling large number of different data formats. Secondly, the use of services interoperability need to be used to diffused between the different systems as well making the classes as cohesive as possible in order to maximize the flexibility of the system. This ultimately facilitates replacement and reusability of the software product classes.

2.1.6 Understandability

This refers to the easiness of the user in understanding the software product logical concept and applicability (Belander etal, 20015). In a class, every module's purpose should be easily understandable in terms of how it accomplishes the tasks which are defined as the class methods and conditions that have to be followed in order for the behavior to be seen. The code should be well commented to make it easier for any

developer who may later be charged to conduct changes on the code. The developer should ensure that proper naming of the class, attributes and the methods has been adopted in line with the set standards of the implementation language. If these standards are followed and it becomes easy for a developer to follow what has been coded then that class is said to be of high quality

2.1.7 Usability

Every high quality software is said to be highly usable. According to Msdn (2015) usability refers to how well a software product is able to meet the needs and requirements of the user as specified. The software product should at the same time be as intuitive as possible during the user's interaction. The software product should be easy to localize and globalize, providing the ease of use to the disabled users with an overall good overall user experience.

When a developer is designing a class, they should ensure that for an effective usability; user interaction is kept at an acceptable level and the use of multistep operations is implemented. Multistep operation involves a long process that is broken down into several sub process that don't disorient the user during interaction. The developer ought to take into consideration the use of proper and appropriate feedback to the user. This should be implemented in the design of the class methods to ensure that each action that is performed by a given task has the appropriate inputs and the expected outputs.

2.1.10 Reusability

In object oriented programming (OOP) the development process is normally characterised with reusability of the class modules for any given system. Reusability refers to the chance that a given module will be used in other modules or even a scenario to give more functionality with little or no change to the module being reused (Msdn, 2015). Highly cohesive classes allow maximum probability of class reuse in other classes and their modules within them and in other classes. When a class and its components are highly reusable, it minimizes the modules and attributes duplication which can otherwise add unwanted redundancy therefore increasing coupling. Class and modules reuse also reduces the amount of time that a given software product takes to be implemented since some of the modules or components can be easily extended or called for use in the system. This in the long run translates to reduced development efforts which lower the maintenance costs of the software since a required part of the system does not have to be developed or implemented from scratch.

When designing a class in favour of high cohesion, a developer should be careful with reusability implementation in order to avoid duplication of code and especially when the functionality is not well layered. It is also important for developers to reuse the classes and components through the use of services instead of mere traditional reusability. The use of services in reusability ensures that the system or class functionality is not entirely reused but a service component acts as a broker for the system classes and modules. The use of service brokering has been reported to have positive effects on the performance of the software products in terms of system loading and response whenever a specific method is invoked or called by another class module.

This research looks into the structural quality of the software at the class level; class cohesion.

2.2 Class Members

A class refers to a blueprint of creating objects (Oracle, 2015) which is normally designed to accomplish a specific task. A class is normally declared using a keyword *class* (Msdn, 2015) and by default every class is publicly accessed. Apart from the use of the *class* keyword in its definition, a class also contains other members; fields or attributes, properties, events and methods.

2.2.1 Attribute Members

An attribute member of a class is also known as a variable, an instance (in object oriented programming) or a field of a class. Variables within a class or program can exist within two scopes: global and local scopes. In the global scopes, these storage locations are accessible through the entire class for usage and are therefore within the class overall environment (Msdn, 2015). This accessibility could result to unwanted mutual dependencies within the code which ultimately leads to unnecessary complexity. The use of global attributes in a class can change the assigned values throughout the class hence their value is not known when they are declared and can be overridden by local definitions (PHP, 2017). For high class cohesion, the class global variables must be used within its methods otherwise, if it has not used any field then its cohesion is said to be zero (Okike, 2010). Furthermore the more the interaction of these variables in class methods the higher the cohesiveness of the class as pointed out by (Briand, 98). In a proposal by Dallal (2010), he noted that the higher the interaction

between variables of a class the higher the cohesion as a result of attribute to attribute interaction (AAC). Global variables are also declared within the top section of the class or code.

Local attributes are those variables that are normally visible within the local scopes of a class or program code. Local variables reference the block or the function where they are declared (PHP, 2017). Just like the global variables, the more the number of local attribute reference within a method, the higher the module cohesion (Lustman, Keller & Kabaili, 2001). These attributes are also unknown to the rest of the class or even other functions of the class. The values assigned to the local variables are often recreated whenever the function is called or even executed.

2.2.2 Member Functions

A method refers to a section of a class that performs a specific task (Msdn, 2015). A class must have at least one method. In most programming languages, a valid class normally contains a main method which acts as a starting point of its execution. There are two classifications of methods;

- i) System defined method – these types of functions are inbuilt within a given programming language and are available for use when needed. If a programmer wants to use a system defined method, all they do is to import that function within their class and the importation is also language specific.
- ii) User defined methods – these types of functions are normally designed by a programmer intended to accomplish a specific task that cannot be done by the inbuilt function.

Methods can be further classified on the basis of their return type;

- i) Value return methods – these group of methods have a specific type of return for example string returned value or even an integer returned value.
- ii) Void or a null value returning methods – these group of methods are not expected to return any value within their definition.

Methods are also known as functions and they normally accept some data input, does the process and return some data usable within the class or within another module of the class or to another class through inheritance.

Methods have different access modifiers that allow them to be scope (visible) from within the sections of a class or even through inheritance to and from another class. These modifiers are; public, protected and private. Public methods do not have any restrictions in passing their data from one module of the class to the other and are therefore globally available to the entire class and its inherited class. A class is said to be more cohesive when attributes are referenced more within it methods (Okike, 2010).

Protected methods are normally limited to their class of definition and any other derived class from the defined class. Private methods are only available to their class of definition and cannot be inherited by other classes. The effect of private and protected methods in measuring class cohesion is similar to that of public methods and it only changes when and if the class is to be used through inheritance since private and protected methods cannot be used in the derived classes (Lustman, Keller & Kabaili, 2001). The higher the number of public methods and any other method usable within a class leads to lower cohesion (Briand, 98). As pointed out by Chae et al

(2004), the use of special methods for example constructors and destructors increase cohesion values due to the indirect methods links. The use of method calls in measuring class cohesion also increases its value as noted by Badri and Badri (2004).

2.3 Software Metrics

Software metric refer to a property of a software, its documentation or its development process that can be objectively measured (Sommerville, 2015). Measurements involve acquiring a value from one product or part of the product and comparing it with another similarly developed product in regard to the standards that should be used so that conclusions can be made on the effectiveness of the process activities, tools and methods employed in delivering the final software product.

As stated by Sommerville (2015), there are two types of software metrics; control metrics and predictor metrics. Control metrics deal with managing the software process and are normally used in assisting software project managers in deciding if the software process used in the development should be changed or not whereas predictor metrics deal with predicting available characteristics of a given piece of software. Predictor metrics are normally used in deciding and determining the required efforts that are needed in making the changes identified on the software process.

The use of measurements on software systems is very important since they;

- i) Allow the development team and the users of the product to assign a value to quality attributes. For example, a software can be said to have low scalability on the basis of how it is designed and its sub systems. It can also be said to be highly secure on the basis of authorization and authentication of its users. The use of low

as a scalability value and high as a security value have been assigned to the two attributes.

- ii) Can be used to identify attributes that do not conform to the set software product standards. For example in the previous case of a system having low scalability and it is expected to have high scalability; low scalability would call for a review of the low scalable components of the system and a subsequent redesign or reengineering until they conform to what is expected before its intended use. The same case would happen for the security if it was noted as poor.

In this research, the product metrics which form part of predictor metrics are investigated. Product metrics are used to measure the internal characteristics of the product. For example number of methods associated with a given class can be calculated. There exists two types of product metrics (Sommerville, 2015):

- i) Static product metrics – this metrics deal with measurements that are normally collected from the representations of the system. In this study the attributes and methods that make up the class will be measured and used as part of cohesion values for the scoped class cohesion metric (SCCM).
- ii) Dynamic product metrics – this metrics deal with measurements that are normally collected during the program's execution. Examples include bugs detection or execution time which could mostly or only occur when the programming is in its running state.

In this research static product measurements are collected and used in coming up with the scoped class cohesion metric (SCCM). The scoped class cohesion metric (SCCM)

measures the cohesion of the class by calculating the number of methods and attributes and their interaction.

2.4 Class Cohesion

Class cohesion is one of the software product metrics. Many class cohesion metrics have been developed that look into the use of attributes and methods in a class. Meyers and Binkley (2007) point out that, a good cohesion metric helps in identification of modules that require reconstruction. This research study has introduced a new class cohesion metric that accounts for class cohesion from a scoping perspective based on the interaction of the class methods and attributes.

Singh and Kaur (2012) outline that class level cohesion metrics are based on two assumption views;

- i) Type of method parameter match accessing attributes types.
- ii) Set of attribute types accessed by a method as an intersection of set of attribute types and methods parameters types.

According to Badri and Gueye (2008), class cohesion can be evaluated from different views namely;

2.3.1 Functional perspective

This is the most desirable cohesion view. As described by Perepletchikov et al (2007) it involves every element's or part of a module contribution to a single unit of a well-defined task. Normally the module or the function performs only one operation.

As shown in the figure 2.1, the elements A, B and C are input elements to be used in the method 1 whose sole functionality is to give out a specific output.

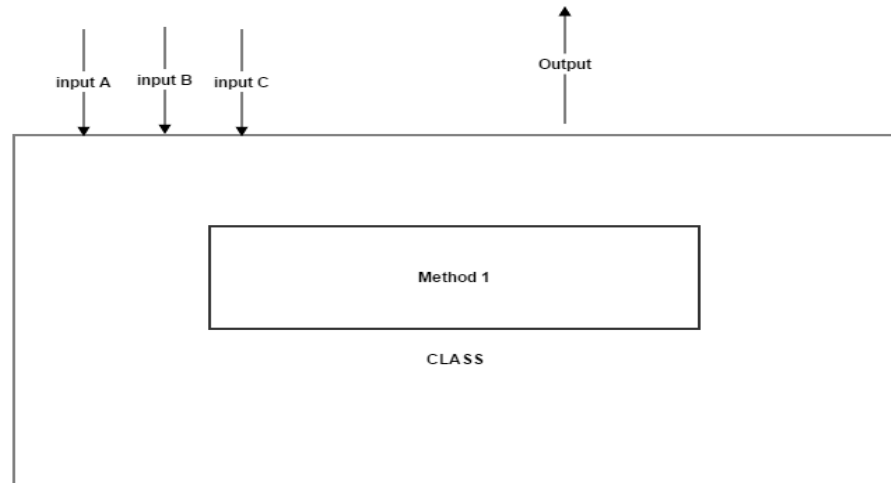


Figure 2.1: An illustration of functional cohesion view

2.3.2 Sequential perspective

This type of cohesion involves one module whose functions are related such that the outputs of one function are the input of the next module (take a particular order) as described by Daghighzadeh, Dastjerdi and Daghighzadeh (2011).

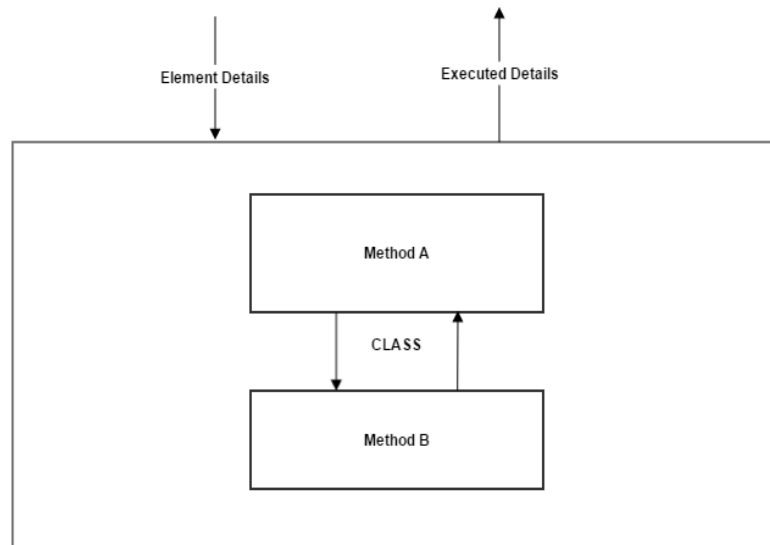


Figure 2.2: An illustration of sequential cohesion view.

The class accepts element details which act as its inputs; they are consumed by method A whose outputs become the inputs of method B which ultimately gives a single output of the module.

2.3.3 Communicational perspective

This type of cohesion involves each module of a class performing its own function (different function from another) but referencing the same data input or information output (Daghaghzadeh, Dastjerdi, & Daghighzadeh, 2011).

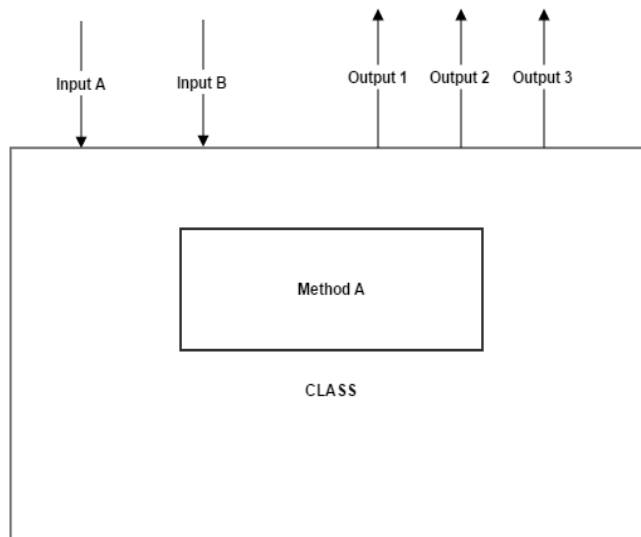


Figure 2.3: A diagram of communicational cohesion perspective

In figure 2.3, a method A consumes data from input A and B which can also be consumed by any other additional function within the same class. Outputs 1, 2 and 3 can also be targeted towards one functionality outside the class which could be targets from more than one methods in the class relayed as these outputs.

2.3.4 Procedural perspective

This type of cohesion involves a module whose elements perform different functionalities, but the activities are implemented in a sequential way (Perepletchikov, Ryan, & Frampton, 2007). As shown in figure 2.4, an action performed in method B can only be executed when all the actions in method A or a required action in method A is executed. It is sequential in nature with strict emphasis on A happening before B.

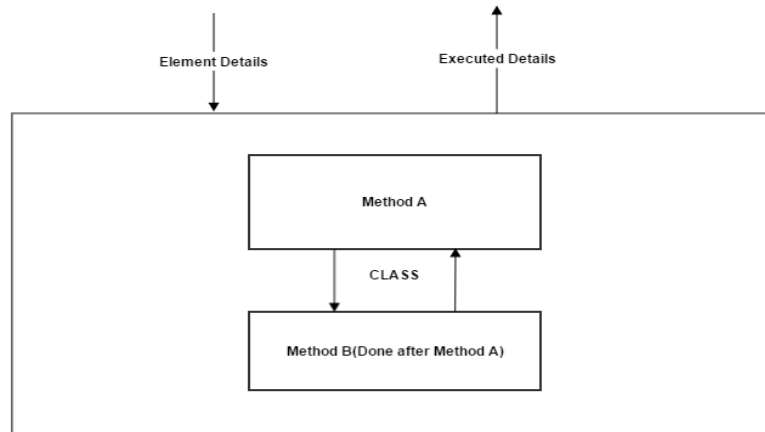


Figure 2.4: A diagram showing procedural cohesion perspective

2.3.5 Temporal perspective

This type of cohesion involves a module or class whose tasks are all related in time (Perepletchikov, Ryan, & Frampton, 2007).

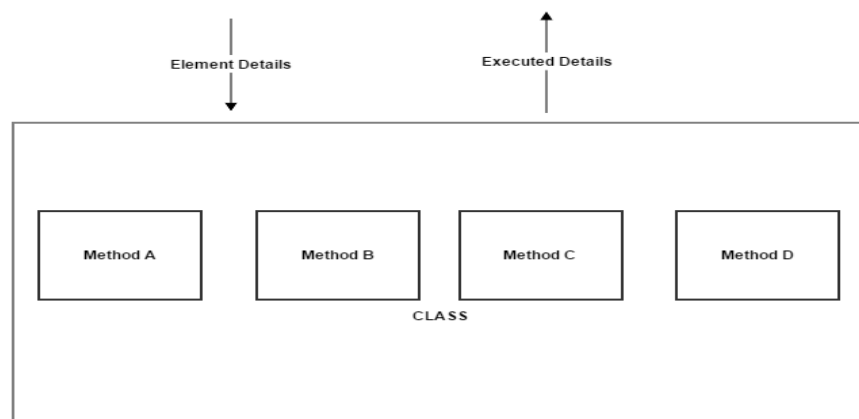


Figure 2.5: A diagram showing temporal cohesion perspective

In this perspective, methods A, B, C and D must happen nearly or at the same time. For example, if the class is an authentication class for a system user then A could be checking if the username and password are as required, B could be verifying the user's account while C and D could be starting a user's session and redirecting them to their customized homepage. All these activities are performed closely and nearly within the same time frame even without the user's awareness combined into one task of user login.

2.3.6 Logical perspective

This type of cohesion involves grouping together class elements that perform similar activities into one module (Pereplechikov, Ryan, & Frampton, 2007). Each action within the module is logically executed ensuring that any incoming data is used for a specific action or invocation.

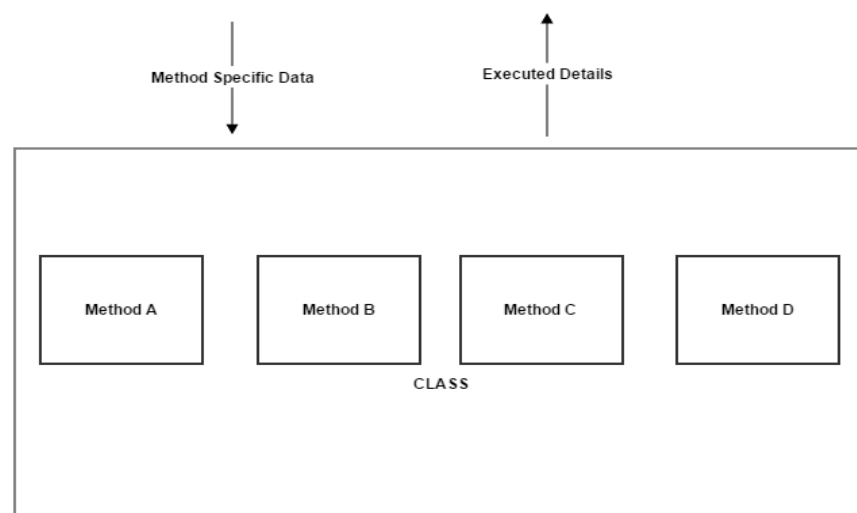


Figure 2.6: A diagram showing logical cohesion perspective.

In figure 2.6, the class module has various methods A, B, C and D. Each of these methods is supposed to accomplish a specific role towards single task completeness. Within the logical perspective data to be used by each method gets into the class (variables of the class) and is consumed specifically by each method.

2.3.7 Coincidental perspective

This type of cohesion involves a module or a class whose elements or functions do not bear any meaning relationships whatsoever, but in the process give ultimate and specific functionality to the module (Perepletchikov, Ryan, & Frampton, 2007). This cohesion should be avoided since any change in the overall functionality may affect more than one element interactions in the different modules.

Coincidental cohesion perspective can be illustrated using the code section below which shows one class that has several methods which are not connected in any way but are within one class (one task, with different modules).

```
class Employee
{
    public void getDetails()
    {
    }
    public int getAge()
    {
    }
    public double getSalary()
    {
    }
}
```

Figure 2.7: Code sections showing three unrelated methods.

The first method gets the details of the employee, while the second gets the age and lastly the employee's salary. The three methods are not called or referenced in each other, but they all work to support the employee object.

2.4 Existing Metrics

2.4.1 Lack of Cohesion Metrics-LCOM 1 and LCOM 2

These are known as the Chidamber and Kemerer (CK) metrics. They were introduced by Chidamber and Kemerer and pioneer the use of metrics in classes. As outlined by Sharma and Srinivasan (2013), they are inverse cohesion measures based on the number of pairs of methods that do not have any common attribute (Uthariaraj et al., 2013).

According to Sridaran and Sreeja (2012), a class with zero value indicates that none of its methods use any of the attributes, therefore lacking cohesion.

Lack of cohesion metric 1 (LCOM1) counts the number of pairs of methods that do not share common attributes as outlined by Chidamber and Kemerer (1991).

Considering a class **C** with methods **Ma, Mb, ..., Mn**, let **{li}** = set of instance variables used by method **MI**; then, there exists **n** sets such that **{li}, ..., {ln}** and LCOM is the measure of disjoint sets formed from the intersection of the sets.

When a class is noted to have high value of lack of cohesion metric 1 (LCOM1) then that class indicates the functionality disparity of the class. This disparity could be as a result of many objectives that a class is trying to implement and should be broken down into smaller classes with small number of methods.

This metric has been criticized to having zero values when used on different classes and is also based on the method to data interaction. It is also not well suited for class that accesses data through the class properties.

Lack of cohesion metric 2 (LCOM2) was the second class cohesion metric proposed by Chidamber and Kemerer (1994). The total class cohesion is measured by subtracting the number of pairs of methods that share common attributes from Lack of cohesion metric 1 (LCOM1). When the value of Lack of cohesion metric 2 (LCOM2) is low then the class is said to highly cohesive and also indicates that encapsulation is decreased, but also increases complexity consequently increasing chances of errors.

$$\mathbf{LCOM2(C) = P - Q.} \quad \mathbf{(2.1)}$$

Where, **P (LCOM1)** is the number of pairs of methods that do not share common attributes and **Q** is the number of pairs of methods that share common attributes.

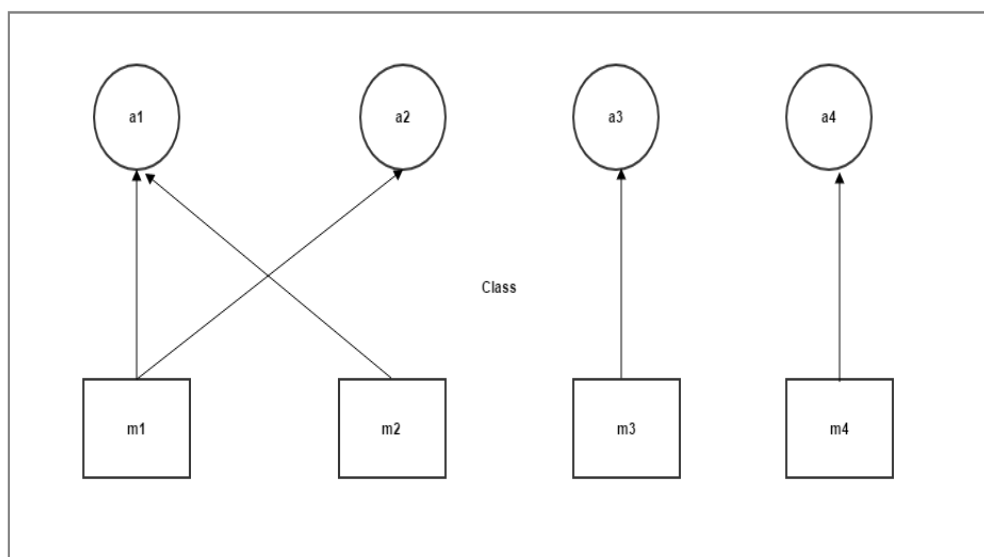


Figure 2.8: An illustration of a class components interaction

In figure 2.8, the represented class is made up of four methods (m1, m2, m3 and m4) and four attributes (a1, a2, a3 and a4) and the value of Lack of cohesion metric 1 (LCOM1) is calculated as shown below in equation 2.2:

$$\text{LCOM1} = P - NP - Q \quad (2.2)$$

Where NP is the number of method pairs. The value of NP in this case is 6 while the value of Q is 1. Therefore, the value of Lack of cohesion metric 1 (LCOM1) is 5 while that of Lack of cohesion metric 2 (LCOM 2) is 4 (following Equation 2.1).

The above illustration can also be illustrated with the code snippet shown in the figure 2.9;

```
class LCOM12
{
    //declare the attributes
    int a1,a2,a3,a4;

    //methods
    public void m1()
    {
        System.out.println(a1); System.out.println(a2);
    }
    public void m2()
    {
        System.out.println(a1);
    }
    public void m3()
    {
        System.out.println(a3);
    }
    public void m4()
    {
        System.out.println(a4);
    }
}
```

Figure 2.9: A code snippet used in calculating LCOM1

These two metrics use pairs of methods that do not share common attributes while the scoped class cohesion metric (SCCM) will use all the methods of a class to identify the use of attributes(direct and indirect) within local and global scopes.

The two metrics also do not cater for the scope factor within their calculations, but focus solely on the attributes commonality. The scoped class cohesion metric (SCCM) will primarily focus on the scope factor in the measurement.

Both Chidamber and Kemerer (CK) metrics are anchored on the use of attributes within their methods much the same as what will be factored in the scoped class cohesion metric (SCCM). This will account for the usage of attributes within local (within methods) and global (class-wide) scopes.

Both the Chidamber and Kemerer (CK) metrics use a parameter occurrence matrix in the class cohesion measure.

The scoped class cohesion metric (SCCM) will measure class cohesion from attributes usage view within a class. However; the Chidamber and Kemerer (CK) metrics looks at the absence of these attributes and does not include the aspect of scopes within their calculations.

2.4.2 Lack of Cohesion 3 Metric (LCOM3)

This metric measure is based on the use of undirected graph (Dallal and Briand, 2010) and was proposed by Li and Henry (1993).Each class method is represented as a graph node (vertice) and any shared instance attribute(s) is represented as an edge. The total class cohesion (LCOM3) is the number of connected graph components.

Lack of cohesion metric 3 (LCOM3) uses a graph concept to represent class methods (vertices) and attributes (edges) while the scoped class cohesion metric (SCCM) will use a parameter occurrence (PO) matrix with rows to represent class methods and while the matrix columns will represent the class attributes in their scopes.

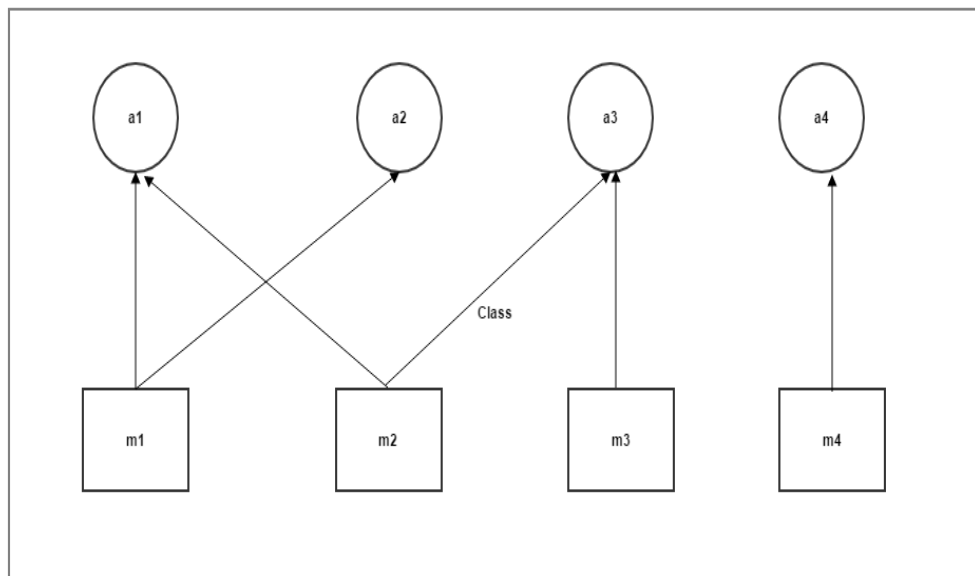


Figure 2.10: An illustration of the LCOM 3 Metric components interaction

In figure 2.10, there are four attributes a1, a2, a3 and a4 and also contains methods m1, m2, m3 and m4. Lack of cohesion metric 3 (LCOM3) calculates the total cohesion as the total number of connections of the graph and the value of lack of cohesion metric 3 (LCOM 3) is 1; the number of disjoint (attribute 4 and method 4) methods is one (1) which is not connected to the other graph nodes. The scoped class cohesion metric (SCCM) will calculate the total cohesion from the total number of occurrences of direct and indirect usage of attributes in the scopes.

Both the scoped class cohesion metric (SCCM) and the lack of cohesion metric 3 (LCOM3) metric will account for class cohesion from the interaction of the attributes and the methods.

The scoped class cohesion metric (SCCM) will use both class members (methods, attributes) just like the lack of cohesion metric 3 (LCOM3), but it will also add scope view measure which will give a better interpretation of inner and outer sections of class members' interactions.

2.4.3 Lack of Cohesion 4 Metric (LCOM4)

This cohesion metric extends Chidamber and Kemerer (CK's) work where a class **X** has a set of instance attributes **I (x)** and a set of methods **M (x)**. An undirected graph **G (v, e)** is used where **M(x)** represents vertices. The graph edges are formed when two vertices access the same instance attribute (Chandrika et al., 2011). Lack of cohesion metric 4 (LCOM4) is measured as the number of connected components of **G(x)**. It further proposes that large classes should be divided into smaller, more cohesive classes if $LCOM4 > 1$.

An illustration of lack of cohesion metric 4 (LCOM 4) is shown in figure 2.11 where two disjoint are seen. Disjoint 1 comprises of attributes a1, a2 and a3 while disjoint 2 is made up of attribute a4. Therefore the value of lack of cohesion metric 4 (LCOM 4) is 2. If a class has two or components then it should be broken down into smaller classes with each hiding the connected component (Ducasse, Anquetil, Bhatti, & Hora, 2011)

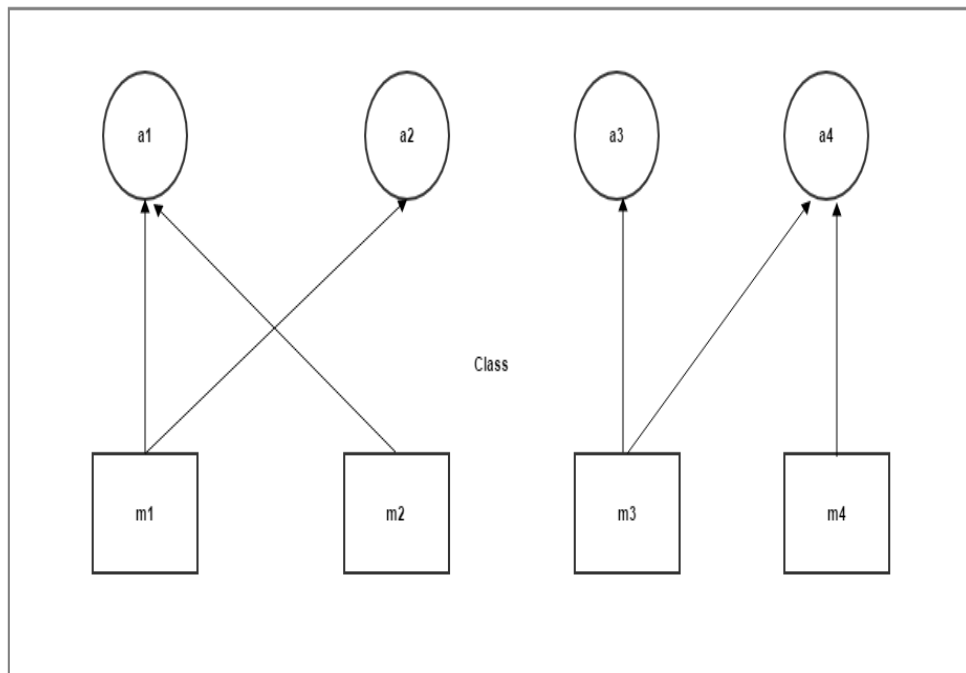


Figure 2.11: An illustration of the LCOM 4 Metric components interaction

Unlike the SCCM which will use a PO matrix, LCOM4 uses the graph concept in its cohesion calculation. It extends Li and Henry's work by adding an edge between pair of methods (Yadav, 2014) if and only if any of the methods invokes the other. Lack of cohesion metric 4 (LCOM4) also does not include the aspect of scope in calculating class cohesion.

When the value of lack of cohesion metric 4 (LCOM4) is equal to 1, then the class is said to have a perfect cohesion and represents a high quality class. When the value is 0, it shows that the class has no cohesion and should be reviewed and redesigned.

Both metrics will use attributes and methods in calculation of class cohesion. This metric also uses an undirected graph like lack of cohesion metric 3 (LCOM3). Lack of cohesion metric 4 (LCOM4) uses the concept of a graph to calculate the class cohesion.

This gives a general view of class cohesiveness unlike the scoped class cohesion metric (SCCM) which will give an inner and outer view of the class cohesiveness through the interaction of direct and indirect attributes.

2.4.4 Lack of Cohesion Metric 5(LCOM5)

This metric was proposed by Henderson-Sellers (1996).It outlines that a given class has a cohesion measure (LCOM5) zero (0) if its every method references all its attributes (perfect cohesion). A one (1) class cohesion value is given, if every class method references only one attribute. This metric uses a normalized range of 0 to 1 and the measure varies as a percentage of the perfect cohesion.

The total cohesion(C) is given by;

$$C = \frac{m - a}{m - h} \quad (2.3)$$

Where **m**=number of methods, **h**=number of attributes and **a**=summation of the definite attributes accessed by each class method.

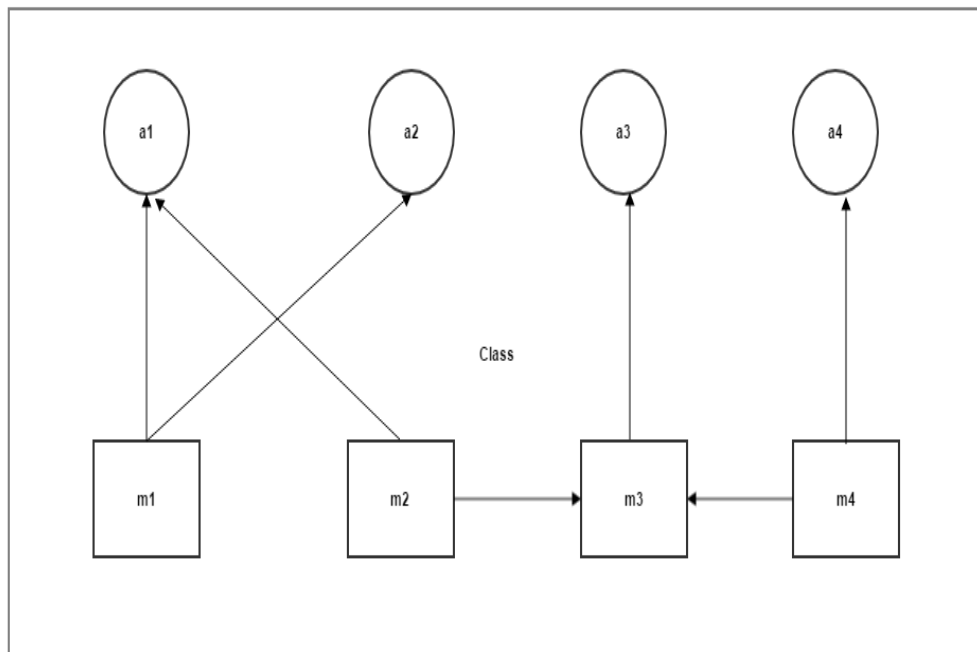


Figure 2.12: An illustration of LCOM 5 metric components

In the illustration shown in figure 2.12, the value of lack of cohesion metric 5 (LCOM 5) is given by;

$a = (2 \text{ methods accessing } a1) + (2 \text{ methods accessing } m3) + (1 \text{ method accessing } a3) + (1 \text{ method accessing } a4)$, $k=4$ and $l=4$; Therefore the value of LCOM5 is 0.833.

Lack of cohesion metric 5 (LCOM5) uses summation of definite attributes in its calculation and does not mention the use of indirect attributes and their scope usage, a concept which will be introduced in the scoped class cohesion metric (SCCM). The lack of cohesion metric 5 (LCOM5) metric only checks the usage of global attributes within the local scope ignoring the impact of local attributes.

The scoped class cohesion metric (SCCM) will measure the presence of class cohesiveness while the lack of cohesion metrics (LCOM) measures the absence of class cohesion (Baig, 2005).

Both metrics will involve the use of attributes within methods and summation of methods in calculating total class cohesion.

The scoped class cohesion metric (SCCM) will observe all the core properties of a good cohesion metric identified by Briand (1996). However, lack of cohesion metric 5 (LCOM5) does not normalize its values in the range of 0 and 1 (Eladawy et al., 2012). The scoped class cohesion metric (SCCM) will also cater for scopes (global and local) as well as the usage of indirect attributes, an ignored factor in the lack of cohesion metric 5 (LCOM5).

2.4.5 Tight Class Cohesion and Loose Class Cohesion Metrics

These two metrics were proposed based on the lack of cohesion metrics (LCOM) by Bieman and Kang (1995).

Tight class cohesion (TCC) – it measures the percentage of pairs of public methods in a class with no common attribute usage. It calculates the percentage of the relative number of directly connected methods (those sharing at least one attribute) (Dallal, 2010).

Loose class cohesion (LCC) –it measures the percentage of pairs of public methods in a class with transitive closure of common attribute usage. LCC calculates the percentage of relative number of indirectly connected methods (two methods that share at least one attribute directly or transitively).

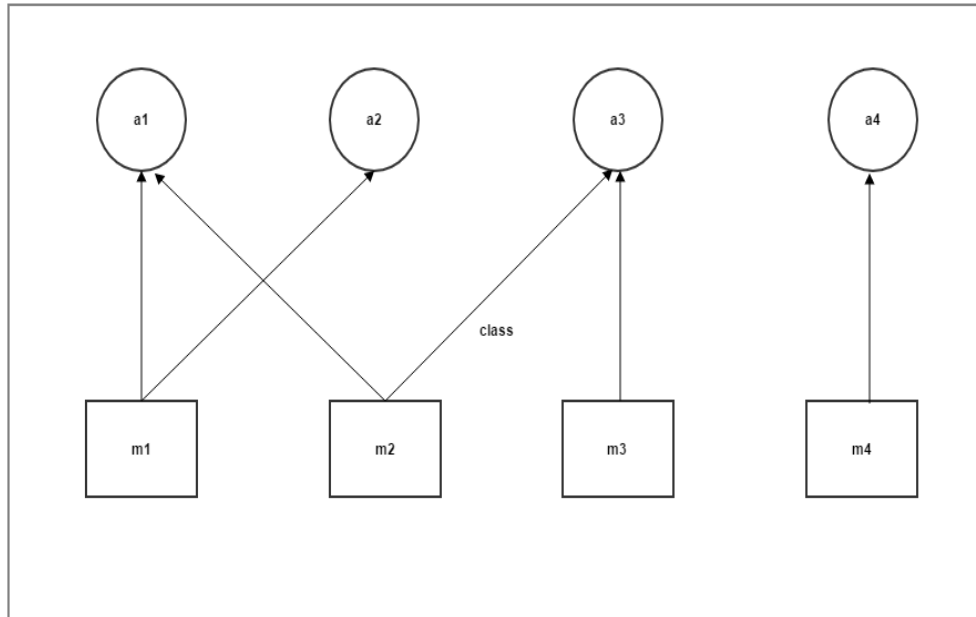


Figure 2.13: An illustration of TCC and LCC metrics.

In figure 2.13, the value of tight class cohesion (TCC) is given by the number of disjoint sets of method pairs whose value is 2 divided by the total possible maximum method pairs whose value is 6 giving a tight class cohesion (TCC) value of 0.333. The value of loose class cohesion (LCC) is given by the number of transitively joint method sets whose value is 3 divided by the total possible maximum method pairs whose value is 6 giving an LCC value of 0.5.

As surveyed by Baig (2005), the cohesion of a class is measured as the relative number of connected pairs of methods to the maximum possible number of pairs.

When the value of TCC and LCC is equal to 1 then the class is said to be highly cohesive which means that all the methods are connected to each other. When the value of LCC is less than one, then all the existing connections are direct even though not all

of the concerned class methods are connected. Since not all methods are directly connected in a class in many instances then the value of LCC is normally lower than that of TCC although there may exist some indirect methods connection. When both LCC and TCC are 0, then it means that none of the class method is connected either directly or even indirectly.

The TCC and LCC metrics are based on the pair of methods sharing at least one attribute while the SCCM will be based on the usage of attributes(direct and indirect) within the local and global scopes. The SCCM will not compare pair of methods, but will work with all the methods of a class.

While the TCC works on the absence of attributes within pairs of methods, the SCCM will calculate the presence of class cohesion through attributes interactions.

All the three metrics integrate the use of attributes and methods in their cohesion measure. The metrics also use direct and indirect attributes within their measures.

The SCCM will integrate scope level calculation unlike the TCC and LCC which only accounts for global scope despite the use of direct and indirect attributes.

2.4.6 Relative lack of Cohesion in Methods (RLCOM)

This was proposed by Li (2006) and modifies LCOM where the total cohesion (C) (Gui & Scott, 2009) is calculated by the formula below.

$$C = \frac{\text{total number of non-similar method pairs}}{\text{total number of method pairs}} \quad (2.4)$$

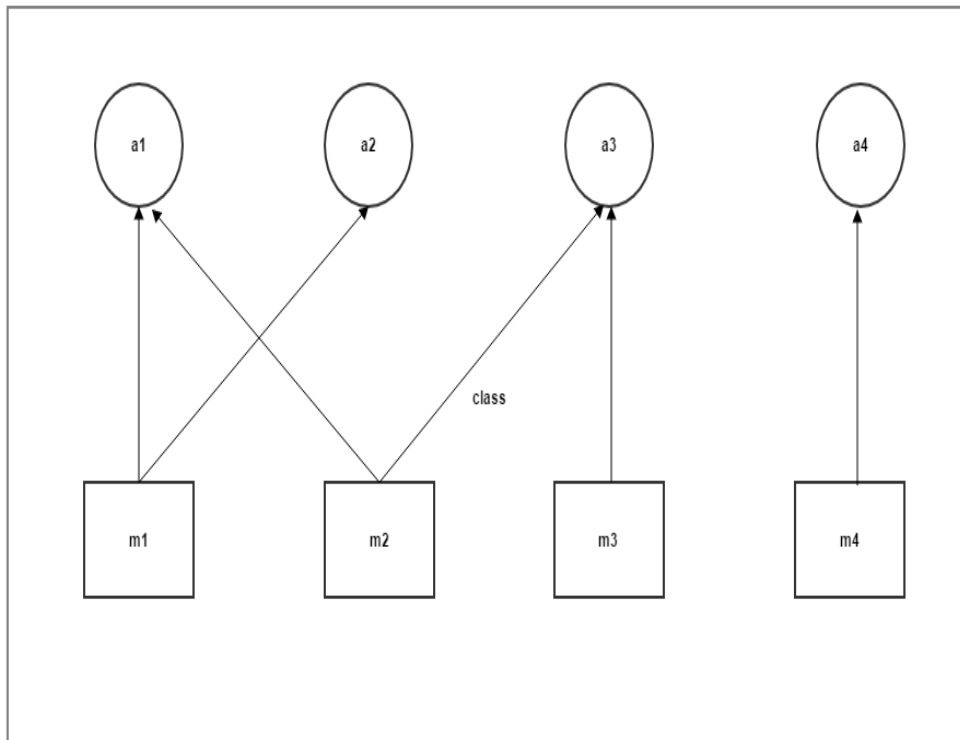


Figure 2.14: Illustration of RLCOM

In figure 2.14, the value of RLCOM is given by the number of disjoint sets of method pairs whose value is 3 divided by the total possible maximum method pairs (following equation 2.4) whose value is 6 giving a RLCOM value of 0.5.

The RLCOM metric works with the use of pairs of methods borrowing the LCOM concept of measuring the absence of class cohesion. The SCCM will use all the methods without pairing them. It will also measure the presence of class cohesion and not its absence. The RLCOM also does not integrate the aspect of scopes in measuring class cohesion, which will be the basis of the SCCM. Both metrics will also cater for methods and attributes in calculation of class cohesion.

2.4.7 Coh Metric

This class cohesion metric enhances the LCOM3 metric by normalizing it in the range of 0 and 1(Ibrahim et al., 2012).It was proposed by Briand et al (1997);

$$\mathbf{Coh} = \frac{\mathbf{a}}{\mathbf{k} \mathbf{l}} \quad (2.5)$$

Where **a** = summation of the number of distinct types accessible by each class method,

k =number of methods and **l** =number of attributes.

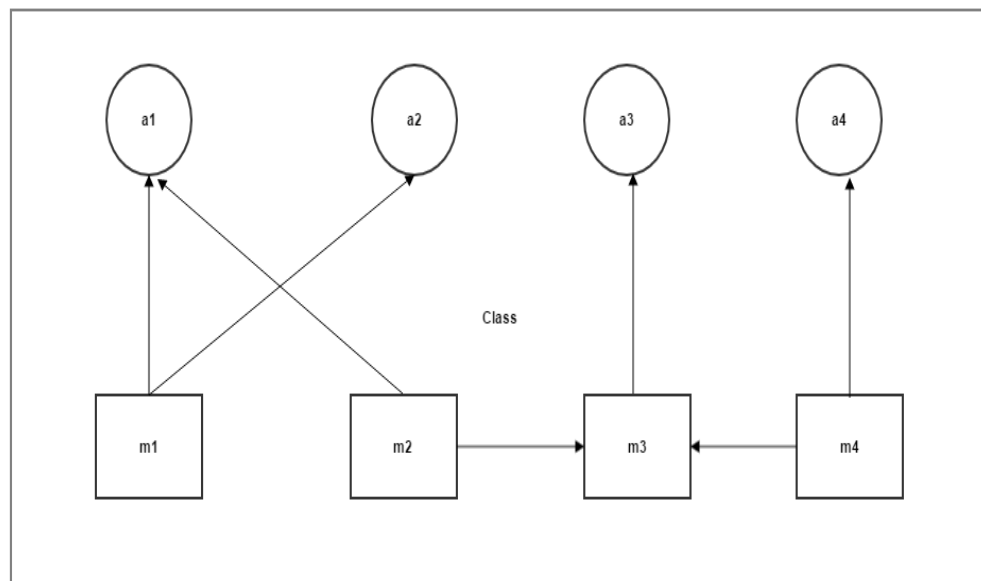


Figure 2.15: An illustration of the COH Metric components interaction

Using figure 2.15, the Coh value is given by;

a= 6, k=4 and l=4

Therefore, Coh value is 0.375 (following equation 2.5). The Coh approaches the measurement of class cohesion from the usage of distinct types for each method in a class while the SCCM will approach the attributes from the scope level and interaction (direct and indirect).

Both metrics will use the interactions of the class members in measuring the class cohesion. They will also cater for attributes usage by each method despite the measure scale (to global view) to be used in the SCCM.

Both metrics will also use a normalized range of 0 to 1(perfect cohesion).

The SCCM will involve attributes interactions (direct and indirect attributes) and their contribution to class cohesion measure in local and global scopes an issue which is not addressed by Coh.

2.4.8 DCD and DCI Metrics

They were proposed by Badri and Badri (2004) and enhance TCC and LCC metrics by including the method invocations; when one method invokes the other (Eladawy et al., 2012):

DC_D (Degree of Cohesion Direct) – it measures the fraction of the directly connected pairs of methods where two methods are directly connected if they are directly connected to an attribute or if they directly or transitively invoke the same method.

DC_I (Degree of Cohesion Indirect) – it measures the fraction of the directly and transitively connected pairs of methods where the two methods are transitively connected if they are directly or indirectly connected to an attribute or if the two methods directly or transitively invoke the same method (Marsic, 2013).

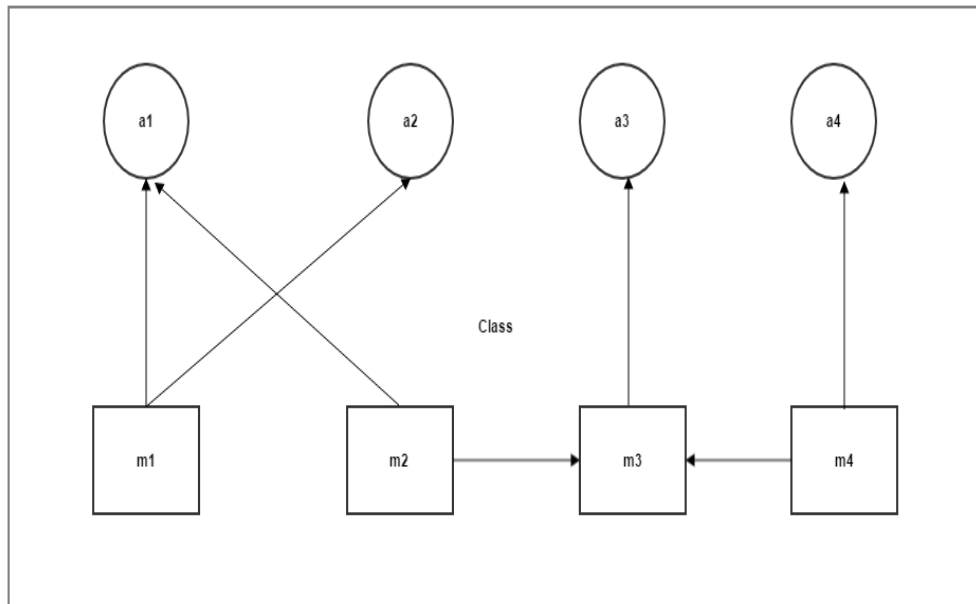


Figure 2.16: An illustration of the DCD and DCI Metric components interaction

Using the illustration in figure 2.16, the value of DCD is given by;

Directly connected pairs=4, total number of methods=6; the value of $DCD=4/6=0.667$

The value of DCI is given by;

Directly or transitively connected pairs=4, total number of methods=6; the value of $DCI=4/6=0.667$.

These two metrics work with pairs of methods for direct (DC_D) connection and indirect or transitive connection of methods (DC_I) while the SCCM will consider all the methods of a class.

The two metrics also do not factor in the aspect scope within their calculations, a key component which will be employed in the SCCM. The two metrics will utilise both class members in the class cohesion measure, a similar approach that will be implemented in the SCCM.

The two metrics also use direct and indirectness between methods, while the SCCM will use direct and indirect usage of attributes within the various scopes.

The SCCM will be based on scoping as a key component in measuring class cohesion. This will ensure that cohesion within a class is holistically calculated from the inner and outer views of the class members.

2.4.9 Class Cohesion (CC) Metric

This metric was proposed by Bonja and Kidanmariam (2006). Cohesion is measured as the degree of similarity between methods.

$$\text{Similarity} = \frac{\text{number of shared attributes}}{\text{number of distinct reference attributes by the methods pair}} \quad (2.6)$$

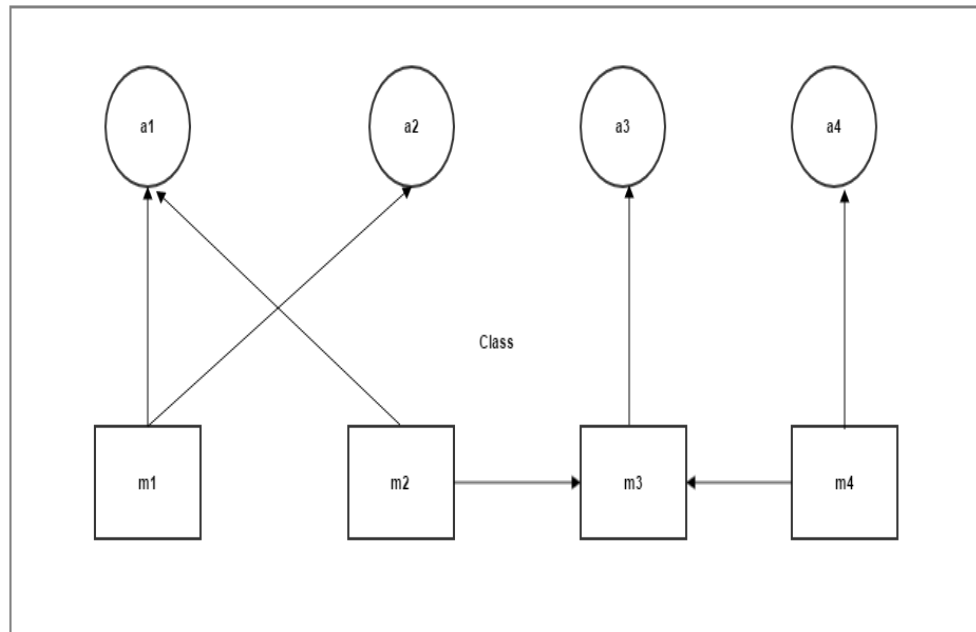


Figure 2.17: An illustration of the CC Metric components interaction

Using the illustration in figure 2.17, the value of CC is given by;

Number of shared attributes=1, number of distinct referenced attributes by shared=2;
the value of $CC=1/2=0.5$.

This metric measures the similarities between pairs of methods while the SCCM will focus on the use of attributes in a class and method scopes without pairing them. The SCCM will also focus on the attributes usage in the various scopes as well.

Both metrics focus on the use of attributes despite the pairing of methods by the CC and general usage of attributes by the CC metric.

Both metrics will also consider all the class methods despite the pairing by the CC metric.

The SCCM will not only considers the use of common attributes by the methods, but also the use of attributes within the whole class in order to understand the usage within the methods and their distribution.

2.4.10 Distance Design-based Direct Class Cohesion (D3C2) Metric

This cohesion metric was proposed by Dallal(2010).It uses a direct attribute type (DAT) matrix that measures the interaction between methods caused by sharing attributes; attribute to attribute interactions(AAC) ,method to method through attributes interactions(MMAC) and attribute to method interactions(AMC). The DAT matrix is a k (number of methods) by l (number of distinct attributes).

The metric is not defined if it has zero methods and zero attributes and it uses the distance between pairs of methods and pairs of attributes to compute the percentage of similarity .The total cohesion (C) of the class is the summation of the AAC, MMAC and AMC.

$$C = AAC + MMAC + AMC \quad (2.7)$$

Table 2.1: AccountDialog Class matrix.

	String	int	Date	Address
showInfo	1	1	1	0
showAddress	0	0	0	1
showExtraInfo	0	0	0	0
readName	1	0	0	0

From table 2.1, the class AccountDialog shows the distribution of attribute usage by the four methods; showInfo () accesses three of the four available attributes, showAddress () uses one and readName () also accesses one attribute.

The value of total cohesion is then calculated as the total of method to method (MMAC) interaction, method to attribute (AAC) interaction and attribute to method (AMC) interactions (following equation 2.7) as shown below;

$$MMAC = \frac{2(1)+1(0)+1(0)+1(0)}{4(4)(3)} = 0.042$$

$$AAC = \frac{3(2)+1(0)+0(-1)+1(0)}{4(4)(3)} = 0.125$$

$$AMC = \frac{5}{16} = 0.313$$

$$\text{Total } D_3C_2 = \frac{4(3)(0.042)+4(3)(0.125)+2(4)(4)(0.313)}{3(3)+4(3)+2(4)(4)} = 0.227$$

This metric is based on the Hamming distance and measures the attribute differences between method pairs while the SCCM will be based on the scoping aspect and will not use pairs of methods, but utilize all the class methods.

Both metrics use a matrix that is used as occurrence entry. They also use all their class members (methods and attributes) in the measurement.

The D_3D_2 metric combines interactions from three views (AAC, MMAC and AMC) while the SCCM will combine the interactions from the global and scope view.

This metric gives more emphasis of pairs of methods in measuring cohesion while the SCCM will give a more specified approach of capturing class cohesion from the attributes(direct and indirect) interaction within methods and outside the methods.

2.5 Research Gaps

The LCOM1 and LCOM2 use pairs of methods that do not share common attributes. The two metrics also do not cater for the scope factor within their calculations, but focus solely on the attributes commonality. LCOM3 uses a graph concept to represent class methods (vertices) and attributes (edges). LCOM3 calculates the total cohesion as the total number of connections of the graph. LCOM4 uses the graph concept in its cohesion calculation and does not include the aspect of scope in calculating class cohesion.

The LCOM5 uses summation of definite attributes in its calculation. The LCOM5 metric only checks the usage of global attributes within the local scope ignoring the impact of local attributes. LCOM5 also does not normalize its values in the range of 0 and 1 (Eladawy et al., 2012).

The TCC and LCC metrics are based on the pair of methods sharing at least one attribute. The TCC works on the absence of attributes within pairs of methods.

The RLCOM metric works with the use of pairs of methods borrowing the LCOM concept of measuring the absence of class cohesion.

The Coh approaches the measurement of class cohesion from the usage of distinct types for each method in a class and their contribution to class cohesion measure in local and global scopes which has not been address by Coh.

The DC_D and the DC_I metrics work with pairs of methods for direct connection and indirect or transitive connection of methods.

The CC metric measures the similarities between pairs of methods.

The D3D2 metric measures the attribute differences between pairs of methods. The D3D2 metric combines interactions from three views (AAC, MMAC and AMC).

The SCCM will calculate the total cohesion from the total number of occurrences of direct and indirect usage of attributes in the scopes. SCCM will primarily focus on the scope factor in the measurement. The SCCM will use both class members (methods, attributes) just like the LCOM3, and adds scope view measure which gives a better interpretation of inner and outer sections of class members' interactions.

The SCCM will use all the methods without pairing them. It also measures the presence of class cohesion and not its absence. The SCCM will not only focus on the shared attributes, but also on their usage and distribution in the various scopes. The SCCM will combine the interactions from the scopes view.

2.6 SCCM Conceptual Framework

Figure 2.18 shows the conceptual framework of the metric.

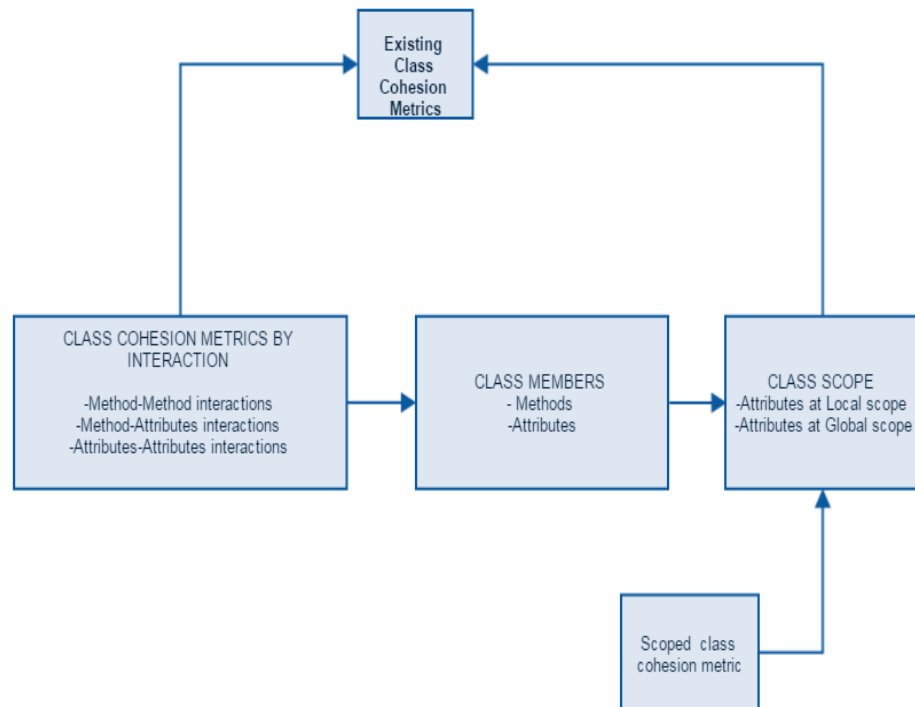


Figure 2.18: SCCM conceptual framework

There exists many class cohesion metrics as discussed in this work; most of the metrics work on the basis of methods-attributes interaction while others are based on attributes-attributes interaction and the SCCM metric is one of the class metrics that will be based on class members interaction (Attributes-Attributes interaction). The SCCM calculations will be based on the occurrence of attributes (global and local scopes) and their interactions with the methods of a class.

2.7 Conclusion

From the above literature it is clear that the aspect of attributes scope and their direct and indirect interactions has not been addressed: The use of pairs of methods (D3C2) is not a conclusive measure of class cohesion. Even with the use of the LCOM (1-5) metrics, they do not give a clear interpretation of class cohesion due to the various interpretations of the original CK metric (LCOM1) which result to a measure of absence rather than presence of class cohesion.

Some of the methods that almost will come close to those of the SCCM; those that use of direct and indirect attributes, combine pairs of methods, but fail to capture the attributes interactions from global and local scopes of a class.

CHAPTER THREE

METHODOLOGY

3.1 Introduction

In conducting this research, quantitative process was followed in coming up with the scoped class cohesion metric. This involved data collection from Github.com which was the main source of data. Github is an open source web platform that allows software developers to develop their software and host them for free on the site for use by other developers as well allowing other contributors to extend the functionality of the softwares through documentation and other software activities that are involved in software engineering.

3.2 Research Design

3.2.1 Sample Population

The sample population refers to the number of units collected for use in the experiment. In this research, the population consisted of sixty software systems classes that were extracted from sixty software systems.

Github was used as the source of data since it hosts a huge collection of open source systems and data sources that record the various software activities which have been used in many systems (Yang, Bai & Zhang, 2016). For example, Ruby on Rails (web scripting language source code), AngularJS (Google developed web frontend framework) and Bootstrap (Twitter developed frontend framework) are some of the

biggest and highly used open source systems. In addition, the data source has huge number of best software engineering practitioners who use and make contributions to the software systems.

The population was then clustered into four groups with each cluster containing fifteen classes representing four of the several object oriented programming languages. The four languages selected were; PHP, Java, Java Script and C++ and selected on the basis of being the most widely used object oriented programming (OOP) languages by developers (Hiscott, 2014).

3.2.2 Sample Size

A sample refers to a proportion of the research population that can be used to generalize certain characteristics on the research population after a study. In order to gather enough data that could be used for conclusive analysis, a total of forty classes were selected from the four clustered programming languages and each cluster was represented by ten classes. The selected classes were also verified to comply with object oriented development of classes to assist in generalization of the results.

3.2.3 Data Collection Methods

The data collection method that was adopted in this research was experimental and used a software to automate the collection process. The various classes were scanned for the methods and attributes and their counts were used in calculating the metric values.

3.2.4 Scoped Class Cohesion Metric

The SCCM is based on a rational scale with a minimum value of natural 0 and a maximum value of 1(Eladawy, 2012). Since a class defines data which is stored in variables (attributes) and behaviours (behaviours) which are expressed as methods (Msdn, 2015) the two elements can be public, private or protected. Most metrics in the high level class design make use of methods and attributes as evident in the LCOM1-LCOM5, COH and others and SCCM is not an exception.

The following parameters were used in the development of the metric;

PM - public methods

PRM - private methods

PA - public attributes

PRA - private and local attributes

PO - public occurrences (of both PA and PRA in public methods and the invocations of any class methods)

PRO - private occurrences (both PA and PRA in private methods and the invocations of any class methods).

TPC - Expected total public cohesion

$$\mathbf{TPC = (PA + PRA) * PM} \quad \mathbf{(8)}$$

TPRC - Expected total private cohesion

$$\mathbf{TPRC} = (\mathbf{PA} + \mathbf{PRA}) * \mathbf{PRM} \quad (9)$$

PC - Observed total public cohesion

$$\mathbf{PC} = \frac{\mathbf{PO}}{\mathbf{TPC}} \quad (10)$$

PRC - Observed total private cohesion

$$\mathbf{PRC} = \frac{\mathbf{PRO}}{\mathbf{TPRC}} \quad (11)$$

SCCM- Total class cohesion

$$\mathbf{SCCM} = \mathbf{PC} + \mathbf{PRC} \quad (12)$$

In the development of the scoped class cohesion metric (SCCM) metric, a close derivative of lack of cohesion metric 5 (LCOM5): Class cohesion (COH) was identified as a good reliability tool for the acquired results. The class cohesion (COH) metric calculates the occurrences of class attributes within its methods and a summation is done. However, the class cohesion (COH) metric omits the use of scopes and use of local attributes, a key aspect in the development of the metric.

3.2.5 SCCM Algorithm

The SCCM algorithm is illustrated using figure 3.3.

```

Step 1: Start
Step 2: Declare variables PM, PRM, PA, PRA, PO, PRO, TPC, TPRC, PC, PRC and SCCM.
Step 3: Assign the new variables length of PM, PRM, PA, PO, PRO and PRA.
Step 4: Add PA and PRA and multiply their total with PM and assign the result to TPC.
      TPC= (PA+PRA)*PM
Step 5: Add PA and PRA and multiply their total with PRM and assign the result to TPRC.
      TPRC= (PA+PRA)*PRM
Step 6: Divide PO with TPC and assign the result to PC.
      PC=PO/TPC
Step 7: Divide PRO with TPRC and assign the result to PRC.
      PRC=PRO/TPRC
Step 8: Add PC and PRC and assign the result to SCCM.
      SCCM=PC + PRC
Step 9: Stop.

```

Figure 3.1: SCCM algorithm

3.2.6 Data Collection Instrument

In this research, a JavaScript tool was designed and developed to collect and analyze the data from various files: the scoped class cohesion (SCCM) tool.

3.2.6.1 SCCM Tool Development Process

The SCCM software tool developed for this research study used an incremental development process model. This methodology was chosen on the basis of having to deliver the application that involves four different implementations due to syntactical differences, faster delivery and useful software to a user through the acquired feedback. Incremental development involves coming up with an initial software product that is released to a user (Sommerville, 2015) and through their feedback

iterations are done taking into account of their views and suggestions until a final product is developed as shown in figure 3.2.

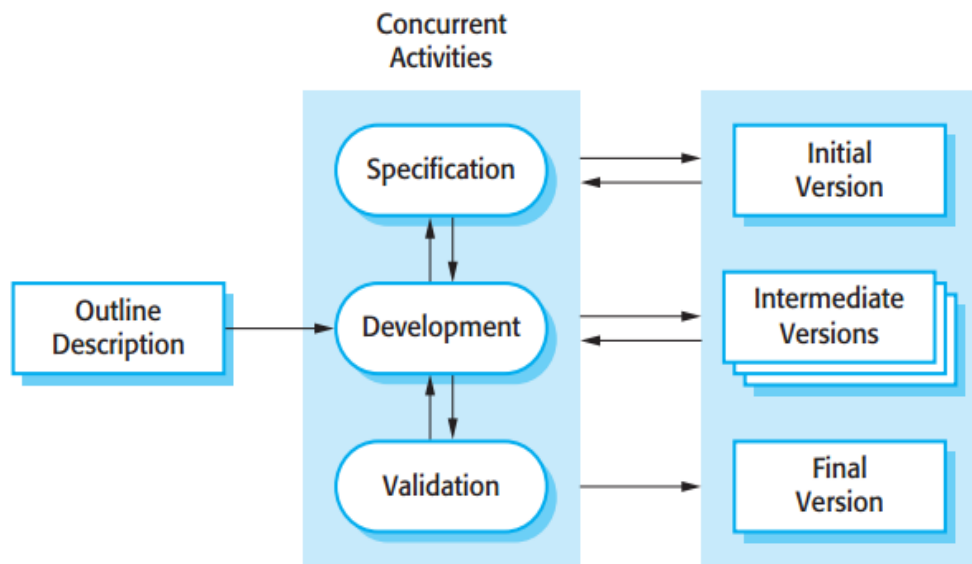


Figure 3.2: Incremental iterative development (Adapted from Sommerville, 2015)

This development was plan driven in nature since the components were already known. With the adoption of this process model, the iterated process activities were mainly; software specification, development and validation.

3.2.6.1.1 SCCM Tool Specification

This is one of the interleaved activities that is iterated during the development process. The stage involves understanding the services that are provided by the software and identification of the constraints during its development (Sommerville, 2015). It takes into account the satisfiability of the users from the software product to be developed (feasibility studies) and elicitation (acquiring information from the users) in order to understand the required system specifications. This then provides a thorough

understanding of the user requirements from the system and the functionalities that the system must have (system requirements).

The following requirements were considered necessary in the iterative development process of the SCCM software:

3.4.1.1 Functional Requirements

- i) The system should be able to only recognize files developed in PHP, Java, JavaScript and C++ languages.
- ii) The user should be able to select files from a saved location.
- iii) The system should automatically calculate the metric value for a given selected class.
- iv) Third party plugins used (i.e. Bootstrap front-end framework) should support mobile interfaces.

3.4.1.2 Non-functional Requirements

- i) The system should be able to process any file size uploaded for metric calculation.
- ii) The system should allow the user to copy and paste file content provided it is in C++, Java, PHP or JavaScript format.
- iii) The software should be accessible on any browser, although it is primarily tested on Google Chrome and Mozilla Firefox browsers.
- iv) The hosting service should be done on a local machine and a copy of the file hosted on an online repository: Github to be accessed by other users.
- v) All system responses to the user should be handled on the console.

- vi) The system user should be provided with a simple manual for reference (support) that describes how the system works as well as any other technical support that may arise.

3.4.1.3 Activity diagram for the SCCM

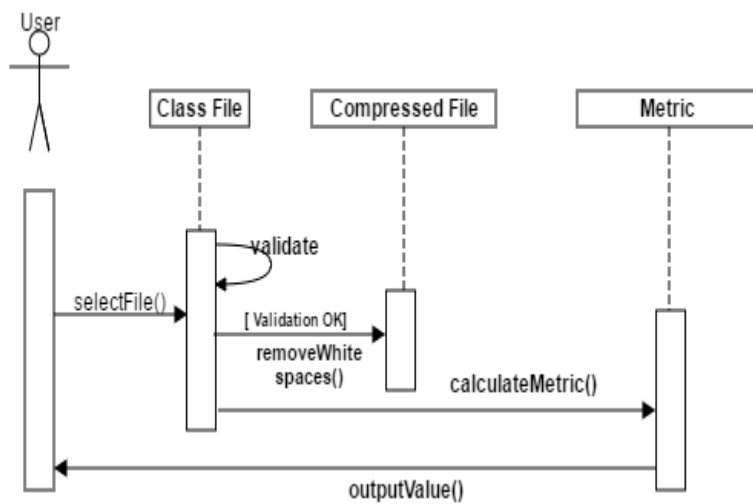


Figure 3.3: Activity diagram for the SCCM software

In figure 3.3, the following activities are performed on and by the system:

- i) The user selects the file from a storage location in the computer.
- ii) The class file is validated removing the white space characters giving a compressed class as an output.
- iii) The metric is then calculated and an output is returned to the user.

3.2.6.1.2 SCCM Tool Design

Software design gives software's structural description, system's structures and data models, components interfaces and algorithms. (Sommerville, 2015).

The following diagram shows an abstract architectural model of the SCCM software.

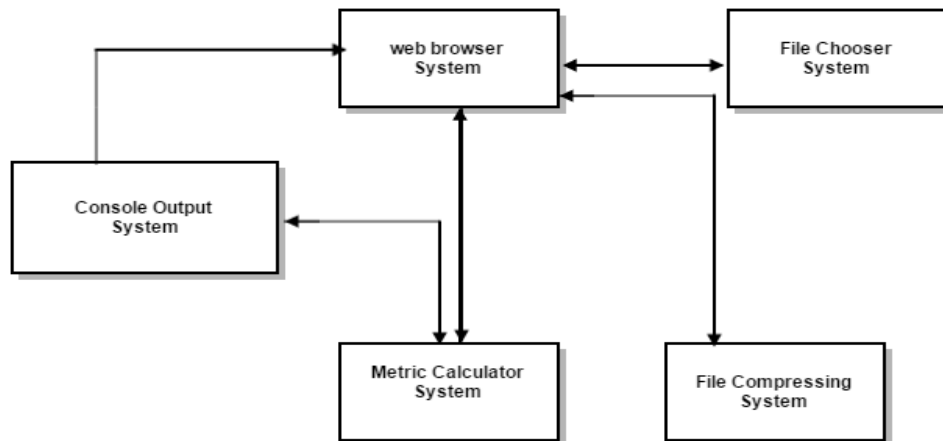


Figure 3.4: An Abstract model for the SCCM software.

In the diagram, the system (SCCM software tool) is subdivided into five sub-systems namely:

(i) Web browser system

This subsystem is responsible for the user interface where the user is able to interact with the system. It has been implemented with the Bootstrap framework.

(ii) File chooser system

This subsystem is responsible for allowing the user to upload a file and decode the contents of the file. The result of this subsystem is the raw code from the file which is later compressed into minified code.

(iii) File compressing system

This subsystem involves minimization of the decoded code. The main aim of this subsystem is to remove whitespace and any untokenized characters from the raw code. The result of this entire process is the minified tokenizable code that can be placed under a parser for analysis.

(iv) Metric calculator system

This subsystem involves the actual metric mathematical operations. The set components that makeup the metric are measured and used in the formula to produce the SCCM metric value which is fed into the output subsystem.

(v) Console output system

This subsystem involves giving the actual metric value to the user. This is done by the web console of the browser.

The SCCM system design used a pipe and filter architectural pattern as shown in figure 3.5.

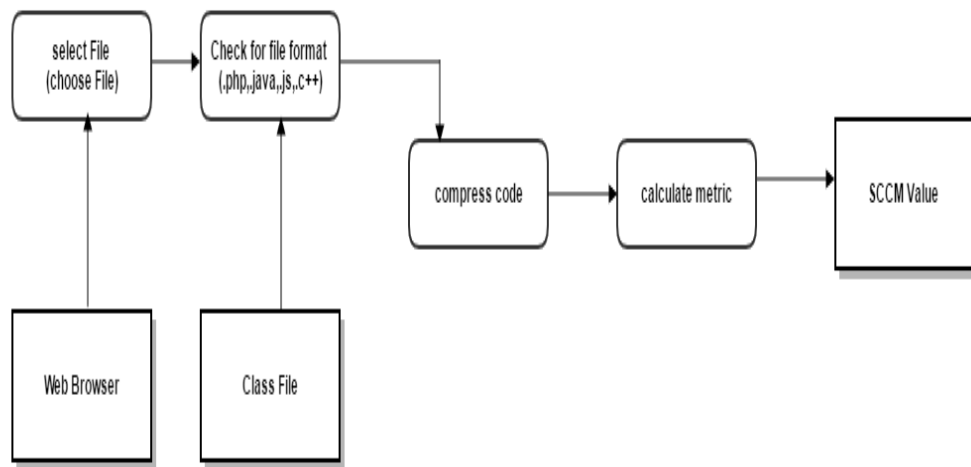


Figure 3.5: Pipe and Filter architecture pattern SCCM software

- i) The user selects a file through the web browser and passed on to the next stage.
- ii) The file is validated through class file checkers and passed on to the next stage.
- iii) The validated file is compressed to remove the white spaces and untokenizable characters and passed on to the next stage
- iv) The compressed code is scanned for the various components that make up the metric, mathematical calculations are done and passed to be used as the SCCM value.

Having identified the architectural pattern, the following high level design was developed:

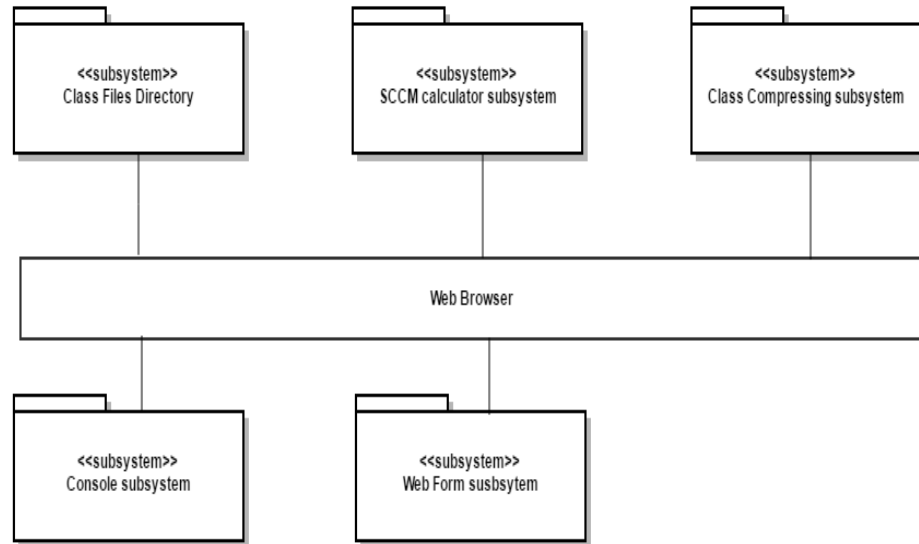


Figure 3.6: High-level architecture of the SCCM software

In figure 3.6, all the five subsystem are interconnected by the web browser subsystem which acts as the user interface.

- i) The user through the browser is able to access the form containing the file chooser input which selects a file stored within a directory in the computer.
- ii) The result of that is raw class code which is displayed on the browser as shown on figure 4.2
- iii) The user can then compress the code by clicking a compress button as shown in figure 4.2 which gives compressed code ready for analysis phase.
- iv) The user then clicks a SCCM calculate button also shown in figure 3.7.
- v) The user can finally get the class calculated SCCM value via the console of the browser.

3.2.6.1.3 SCCM Tool Implementation and Testing

Implementation of a computer system means developing of the system using the existing technologies in order to realize the design and the intended system. During the implementation stage an executable version of the software (system) is actualized. The developed SCCM system was made of two parts: the library/scanner and the view/presentation parts.

The Library or Scanner

This system was developed in pure JavaScript that involved regular expressions for scanning and matching text. This module was made up of four separate files that hold each individual implementation. In order to use any of the four files, the user has to import a specified file on the view based on the scanning file.

The View (Presentation)

This was developed using Bootstrap framework. This is a HTML5 and CSS3 developed HTML page style render.

Testing

This was done using Mocha. Mocha is a JavaScript testing library that is used in creating mocks for testing code.

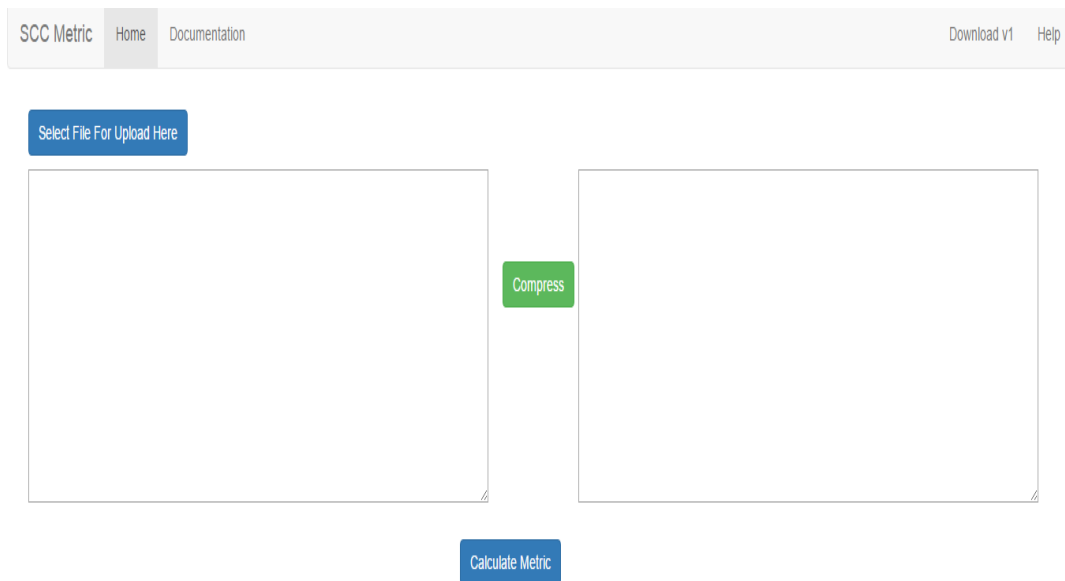


Figure 3.7: Web interface for the SCCM software

The software works by allowing a user to select a valid source code file from a storage location, the user then compresses the code in order to remove white spaces and comments that do not form part of the tokenized source code and then calculates the metric values which are output on the web console.

The implemented system generated the following output on the console.

```

Elements Console Sources Network Timeline Profiles Application Security Audits
) top [x] Preserve log
PI= 13 sccmhp.js:133
PRI= 0 sccmhp.js:145
PA= 5 sccmhp.js:231
PRA= 0 sccmhp.js:278
***** sccmhp.js:889
LCCMS va1ue= 0.8 sccmhp.js:891
***** sccmhp.js:893
***** sccmhp.js:895
COH va1ue= 0.26153846153846155 sccmhp.js:899
***** sccmhp.js:907
SCCM va1ue= 0.5230769230769231 sccmhp.js:932

```

Figure 3.8: The output console of SCCM calculated values.

3.2.6.1.4 SCCM Tool Operation and Maintenance

The system was mainly run on the local machine and a copy also hosted publicly on Github for users utilization. Since the system involved iterations maintenance was done during the several iterations to correct emerging errors that were reported by the users as well as improvements.

CHAPTER FOUR

EXPERIMENT, RESULTS AND DISCUSSION

4.1 Introduction

This chapter discusses the conducted experiment that was involved in calculating the value of SCCM. This involved data collection using the SCCM software tool. With the defined parameters of the metric included in the tool, data was filtered and allocation of counts was done and a value of the metric returned. This section gives the collected filtered data by the tool which was then analysed using graphs and charts and discussed in order to understand the observed chart patterns.

4.2 Experiment Design

During the experiment setup, the following control environments were observed:

- i) A computer with a JavaScript enabled browser since the data collection tool was developed in JavaScript; it was also important for the browser to have a web console that would allow the visibility of the output.
- ii) Hosting of the files was done on the local machine as well as Github to enable the researcher to test the data collection tool on the local computer as well as for other interested tool users to download and test how it works.
- iii) All the data files used (for the four clusters) were acquired from Github; a total of ten standard classes per cluster from ten different object oriented systems.
- iv) A sublime text editor was used as the main software development IDE (as a researcher preference), though any other text editor can be used.

Once the control environments were available, the researcher used the SCCM tool and followed the SCCM algorithm to generate the metric values using the following steps;

- i) The user selects a file from a storage location in the computer. This is done through the web browser and the file chooser subsystems discussed in sections 3.2.6.1.2
- ii) Once a validated file has been selected from the computer directory, the file compressing system removes the white space characters giving a compressed class as an output.
- iii) The compressed class is then analysed by the scoped class cohesion metric (SCCM) tool. The tool calculates the number of each metric parameters (outlined in the algorithm- figure 3.3) and computes a class cohesion value which is then returned as an output to the user on the web browser console.

4.3 SCCM Raw Collected Data

This section shows the data collected using the SCCM tool. Figures 4.1 and 4.2 shows the interface and the console output respectively while tables 4.1, 4.2, 4.3 and 4.4 show the collected raw data.

The data in these tables is described as follows:

- **SCCM** – the calculated value of the research metric.
- **COH** – the calculated value of the cohesion metric (a comparative metric used in the study).
- **PM** – the number of public methods that are contained in the minified class and one of the components used in the calculation of the metric value.

- **PRM** - the number of private methods in the minified class and one of the components used in the calculation of the metric value.
- **PA** – the number of public attributes in the minified class and one of the components used in the calculation of the metric value.
- **PRA** – the number of private attributes in the minified class and one of the components used in the calculation of the metric value.
- **PO** – the number of public attributes occurrence in the minified class and one of the components used in the calculation of the metric value.
- **LV** – the number of local variables or attributes in the minified class. LV is used in making comparison with the other global variables in order to identify the effects of local variables on the metric value.

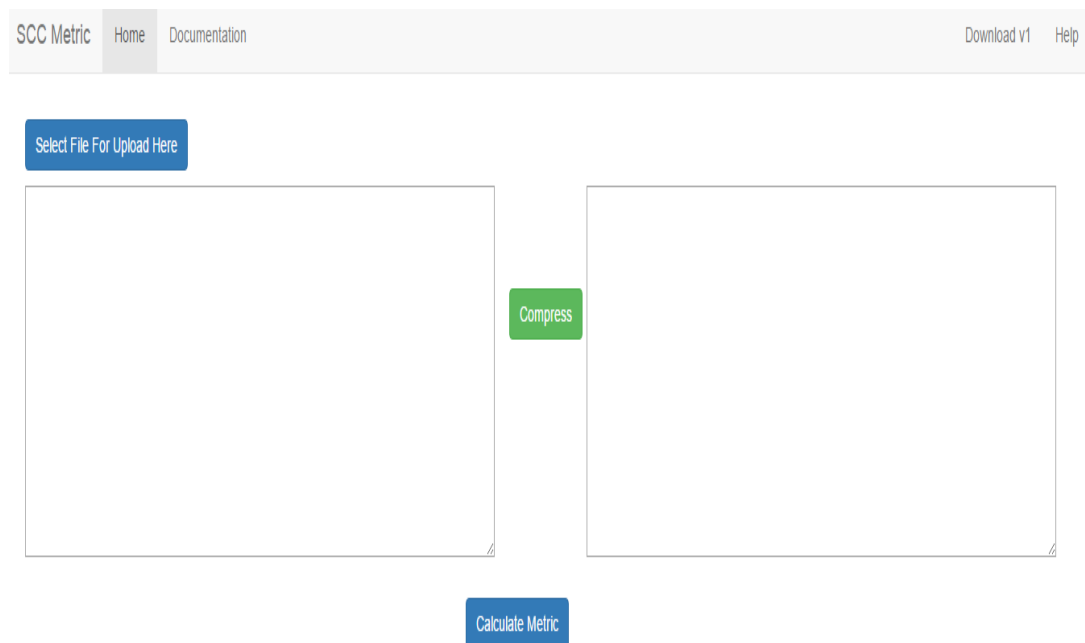


Figure 4.1: A screen shot of the interface of the SCCM software tool.

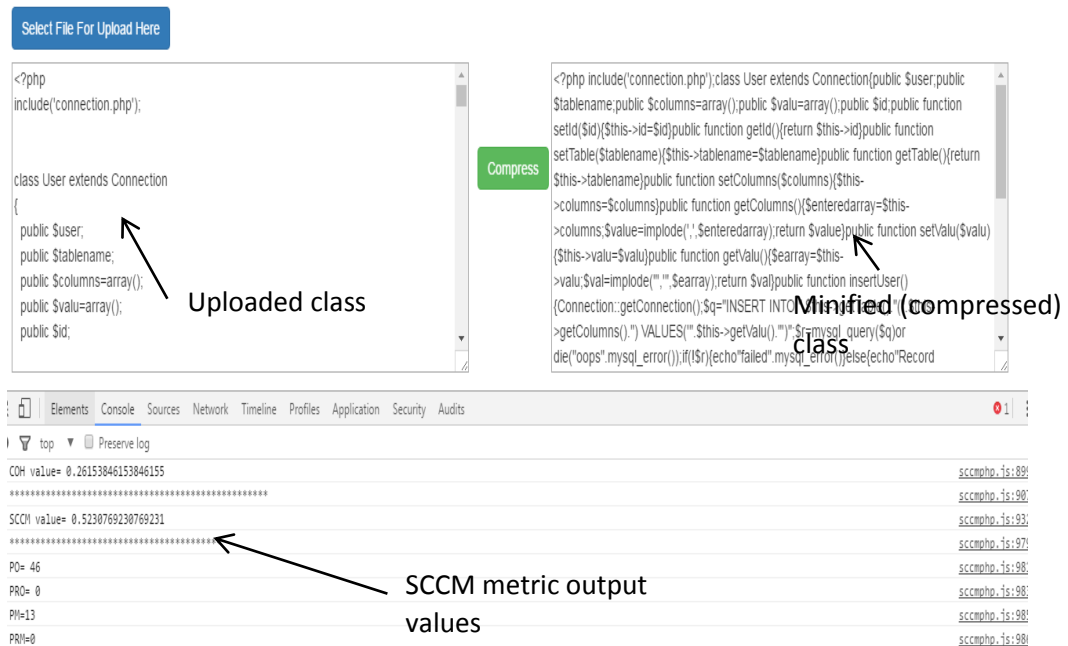


Figure 4.2: A screen shot showing the uploaded class, the compressed code (minified class) and the console output.

Figure 4.1 shows the home interface that the user firstly interacts with when they open the tool. Figure 4.2 shows the interface after the user calculates the metric value as a result of series of events; the user loads a valid file or class, it is compressed and the cohesion is calculated through a triggered event once the user clicks on the calculate metric button.

Table 4.1: SCCM and COH values from the Java Cluster

JAVA SYSTEMS										
SYSTEM	SCCM	COH	PM	PRM	PA	PRA	PO	PRO	LV	LVUSAGE
SOUNDLIBRARY	0.0588	0.0588	17	0	1	2	8	0	2	20
GSM	0.1	0.1	18	0	4	6	49	0	5	110
CANDY SYSTEM	0.75	0.75	2	0	2	2	8	0	1	40
GUMMIES SYSTEM	0.2444	0.2444	9	0	2	3	24	0	2	90
AI	0.0714	0.0714	8	0	2	5	16	0	5	260
CALL4	0.25	0.25	8	0	2	4	18	0	2	70
TRACKENGINE	0.1666	0.1666	6	0	2	2	16	0	7	200
DARK 3 SYSTEM	0.3333	0.3333	5	0	1	2	11	0	1	40
CHOC SYSTEM	0.3	0.2333	10	0	1	2	18	0	4	150
HOTEL SYSTEM	0.125	0.125	18	0	4	4	38	0	7	360
			-0.6967934		0.53099	-0.470630347	0.8852		-0.6	-0.42067

Table 4.2: SCCM and COH values from the C++ Cluster

C++ SYSTEMS										
SYSTEM	SCCM	COH	PM	PRM	PA	PRA	PO	PRO	LV	LVUSAGE
ESCAPEPOD	0.5333	0.3333	5	0	1	0	24	0	0	0
STATIONARY	0.2922	0.2922	27	0	15	0	674	0	4	17
PLAYERGAME	0.419	0.419	7	0	4	0	154	0	5	23
WIDGET	0.325	0.175	8	0	3	0	26	0	1	0
BANKING SYSTEM	0.4474	0.4474	38	0	23	0	3150	0	6	30
HRM SYSTEM	0.6518	0.2024	19	0	3	0	161	0	1	6
SNAKE GAME	0.2286	0.2286	10	0	5	0	145	0	1	8
PERSON	0.5117	0.1839	23	0	3	0	153	0	2	59
LEANCLUBLIB	0.375	0.375	4	0	8	0	320	0	8	30
PROGRAMLIB	0.0893	0.0714	4	0	16	0	5	0	1	0
			0.305		0.19429		0.1176		0.01	0.296514

Table 4.3: SCCM and COH values from the JavaScript Cluster

JAVASCRIPT SYSTEMS										
SYSTEM	SCCM	COH	PM	PRM	PA	PRA	PO	PRO	LV	LVUSAGE
ALERTIFYJS	0.403	0.2113	34	0	5	22	370	0	41	344
ANCESTRYJS	0.875	0.875	4	0	0	4	24	0	7	10
BOARDGAME	0.6155	0.4981	24	0	8	14	325	0	42	888
CLASS12LIB	0.3043	0.1973	23	0	3	10	91	0	16	142
METAClass	0.925	0.125	10	0	4	0	37	0	7	36
JSCLASS9LIB	0.1561	0.0886	82	0	6	50	717	0	108	781
JSLL8LIB	0.189	0.1197	61	0	18	25	496	0	47	2771
PROTOJS	0.3238	0.2285	15	0	0	7	34	0	39	39
PERSONAJS	0.6667	0.3333	2	0	3	3	8	0	12	12
LANGUAGEJS	0.4375	0.25	8	0	1	3	14	0	24	24
			-0.7279191		-0.4001	-0.68625	0.6659443		-0.7	-0.48825

Table 4.4: SCCM and COH values from the PHP Cluster

PHP SYSTEMS										
SYSTEM	SCCM	COH	PM	PRM	PA	PRA	PO	PRO	LV	LVUSAGE
SHOPPING CART	0.5	0.5	5	5	0	1	19	0	6	19244
CONFIGURATION	0.7543	0.7543	10	15	0	7	156	0	17	1462080
CSS	0.2333	0.2333	5	5	6	0	22	12	7	112944
GAME1	0.3333	0.25	4	0	0	3	11	0	1	0
GAME2	0.6	0.6	3	2	2	0	6	4	11	30562
BOARD	0.1667	0.1667	8	0	0	3	21	0	4	1560
CRUD	0.5231	0.2615	13	0	0	5	46	0	12	10610
REGISTRATION-LOGIN	0.4	0.4	5	0	2	0	13	0	8	6940
DSN	0.25	0.2	4	1	0	1	7	0	1	40
TICTACTOE	0.6	0.6	3	2	2	0	6	4	11	30562
			0.19655	0.57127	-0.191	0.33901	0.56	-0.168	0.8434648	0.41363

The above four tables namely table 4.1, 4.2, 4.3 and 4.4 show the distribution of the factors that affect the SCCM and their corresponding values. This is done on ten systems for each cluster of programming language. Table 4.1 shows the values that represent factors in the java systems, table 4.2 shows values that represent factors in the C++ systems, table 4.3 shows values that represent factors in the JavaScript systems and table 4.4 caters for PHP systems factors representative values.

4.4 Experiment Processed Data

In this section, the collected raw data is refined further in order to identify the various relationships that exist within the data. This will then assist in making an informed decision on the effects of the various components that make up the SCCM and the possible explanation of these effects.

In order to identify the effective metric among the two, descriptive statistics have been used for data interpretation. In measuring the central tendency; the geometric mean is used to approve the effective metric among SCCM and COH whereas the relationship between the metric values and its various constituents uses Pearson's coefficient. The COH metric has been considered since its formulation is closely related to the SCCM. The COH metric is also chosen to enhance the study's reliability despite the fact that it does not account for scopes and use of local variables.

4.4.1 Geometric Means on SCCM/COH

A geometric mean is an average that indicates a typical value of a set of numbers through the use of their values product (Geometric Mean, 2011). It is often used to show which item is superior or has a higher merit than the other. For example, if two items A and B have values 1 and 2 and their geometric mean is 1.5, then B is superior than A. The geometric mean has been considered for use in this research in order to identify which metric values are higher between the SCCM and the COH metrics.

Tables 4.5, 4.6, 4.7 and 4.8 shown in this section indicate the various calculated geometric means from the four clusters of the systems classes.

Table 4.5: SCCM and COH geometric mean values from the Java Cluster

JAVA SYSTEMS			
SYSTEM	SCCM	COH	GEOMETRIC MEAN
SOUNDLIBRARY	0.0588	0.0588	0.0588
GSM	0.1	0.1	0.1
CANDY SYSTEM	0.75	0.75	0.75
GUMMIES SYSTEM	0.2444	0.2444	0.2444
AI	0.0714	0.0714	0.0714
CALL4	0.25	0.25	0.25
TRACKENGINE	0.1666	0.1666	0.1666
DARK 3 SYSTEM	0.3333	0.3333	0.3333
CHOC SYSTEM	0.3	0.2333	0.264556232
HOTEL SYSTEM	0.125	0.125	0.125

The geometric values acquired from the Java systems for the SCCM and COH show that both SCCM and COH do not differ in the ten systems. However, CHOC system shows a different value due to availability of extra private variables.

Table 4.6: SCCM and COH geometric mean values from the C++ Cluster

C++ SYSTEMS			
SYSTEM	SCCM	COH	GEOMETRIC MEAN
ESCAPEPOD	0.5333	0.3333	0.4216
STATIONARY	0.2922	0.2922	0.2922
PLAYERGAME	0.419	0.419	0.419
WIDGET	0.325	0.175	0.2385
BANKING SYSTEM	0.4474	0.4474	0.4474
HRM SYSTEM	0.6518	0.2024	0.3632
SNAKE SYSTEM	0.2286	0.2286	0.2286
PERSON	0.5117	0.1839	0.3068
LEANCLUBLIB	0.375	0.375	0.375
PROGRAMLIB	0.0893	0.0714	0.0798

The geometric mean values were lower compared to the values of SCCM in the ten C++ systems. The SCCM values were also noted to be higher than those of the COH metric. This difference shows that the SCCM metric performs better than the COH metric.

Table 4.7: SCCM and COH geometric mean values from the JavaScript Cluster

JAVASCRIPT SYSTEMS			
SYSTEM	SCCM	COH	GEOMETRIC MEAN
ALERTIFYJS	0.403	0.2113	0.2918
ANCESTRYJS	0.875	0.875	0.875
BOARDGAME	0.6155	0.4981	0.5537
CLASS12LIB	0.3043	0.1973	0.245
METAClass	0.925	0.125	0.3401
JSClassLIB9	0.1561	0.0886	0.1176
JSLL8LIB	0.189	0.1197	0.1504
PROTOJS	0.3238	0.2285	0.272
PERSONAJJS	0.6667	0.3333	0.4714
LANGUAGEJS	0.4375	0.25	0.3307

The geometric mean values were lower than the SCCM values in all the ten systems which shows that SCCM compared better than its comparative COH metric.

Table 4.8: SCCM and COH geometric mean values from the PHP Cluster

PHP SYSTEMS			
SYSTEM	SCCM	COH	GEOMETRIC MEAN
SHOPPING CART	0.5	0.5	0.5
CONFIGURATION	0.7543	0.7543	0.7543
CSS	0.2333	0.2333	0.2333
GAME1	0.3333	0.25	0.2887
GAME2	0.6	0.6	0.6
BOARD	0.1667	0.1667	0.1667
CRUD	0.5231	0.2615	0.3698
REGISTRATION-LOGIN	0.4	0.4	0.4
DSN	0.25	0.2	0.2236
TICTACTOE	0.6	0.6	0.6

The values of SCCM were noted to be higher than the geometric mean which reflects that the values acquired are superior to those of COH.

4.4.2 Pearson's Coefficient on SCCM/COH

Pearson Correlation coefficient refers to a measure of the strength of the linear relationship between two variables (Lane, 2016). The coefficient ranges between -1 and 1 with a -1 giving a perfect negative relationship and +1 a positive relationship respectively.

In this research the Pearson correlation values have been used to identify the linear relationship between values of a given component of the SCCM metric as shown in tables 4.9, 4.10, 4.11 and 4.12.

Table 4.9: Pearson coefficient values on SCCM matrix of C++ systems

C++ SYSTEMS								
SYSTEM	SCCM	COH	PA	PRA	PO	PRO	LV	LVUSAGE
ESCAPEPOD	0.5333	0.3333	1	0	24	0	0	0
STATIONARY	0.2922	0.2922	15	0	674	0	4	17
PLAYERGAME	0.419	0.419	4	0	154	0	5	23
WIDGET	0.325	0.175	3	0	26	0	1	0
BANKING SYSTEM	0.4474	0.4474	23	0	3150	0	6	30
HRM SYSTEM	0.6518	0.2024	3	0	161	0	1	6
SNAKE GAME	0.2286	0.2286	5	0	145	0	1	8
PERSON	0.5117	0.1839	3	0	153	0	2	59
LEANCLUBLIB	0.375	0.375	8	0	320	0	8	30
PROGRAMLIB	0.0893	0.0714	16	0	5	0	1	0
			0.19429		0.1176		0.01	0.296514

In table 4.9 above, the correlation coefficient of public variables and the local variables give a very poor positive correlation coefficient value to the values of SCCM.

Table 4.10: Pearson coefficient values on SCCM matrix of Java systems

JAVA SYSTEMS										
SYSTEM	SCCM	COH	PM	PRM	PA	PRA	PO	PRO	LV	LVUSAGE
SOUNDLIBRARY	0.0588	0.0588	17	0	1	2	8	0	2	20
GSM	0.1	0.1	18	0	4	6	49	0	5	110
CANDY SYSTEM	0.75	0.75	2	0	2	2	8	0	1	40
GUMMIES SYSTEM	0.2444	0.2444	9	0	2	3	24	0	2	90
AI	0.0714	0.0714	8	0	2	5	16	0	5	260
CALL4	0.25	0.25	8	0	2	4	18	0	2	70
TRACKENGINE	0.1666	0.1666	6	0	2	2	16	0	7	200
DARK 3 SYSTEM	0.3333	0.3333	5	0	1	2	11	0	1	40
CHOC SYSTEM	0.3	0.2333	10	0	1	2	18	0	4	150
HOTEL SYSTEM	0.125	0.125	18	0	4	4	38	0	7	360
			-0.6967934		0.53099	-0.470630347	0.8852		-0.6	-0.42067

In table 4.10, the number of public methods (PM) and the private variables (PRA) and the local variables gave a negative correlation coefficient values. This is a clear indicator that as the number of PM and LV increase within a class, then the cohesion value of both SCCM and COH decreases and vice-versa.

Table 4.11: Pearson coefficient values on SCCM matrix of C++ systems

JAVASCRIPT SYSTEMS										
SYSTEM	SCCM	COH	PM	PRM	PA	PRA	PO	PRO	LV	LVUSAGE
ALERTIFYJS	0.403	0.2113	34	0	5	22	370	0	41	344
ANCESTRYJS	0.875	0.875	4	0	0	4	24	0	7	10
BOARDGAME	0.6155	0.4981	24	0	8	14	325	0	42	888
CLASS12LIB	0.3043	0.1973	23	0	3	10	91	0	16	142
METAClass	0.925	0.125	10	0	4	0	37	0	7	36
JSCLASS9LIB	0.1561	0.0886	82	0	6	50	717	0	108	781
JSLL8LIB	0.189	0.1197	61	0	18	25	496	0	47	2771
PROTOJS	0.3238	0.2285	15	0	0	7	34	0	39	39
PERSONAJS	0.6667	0.3333	2	0	3	3	8	0	12	12
LANGUAGEJS	0.4375	0.25	8	0	1	3	14	0	24	24
			-0.7279191		-0.4001	-0.68625	0.6659443		-0.7	-0.48825

In table 4.11, the number of public methods (PM), public attributes (PA), private variables (PRA) all indicate a negative correlation coefficient value such that as the number of these three increase then the value of SCCM cohesion decreases significantly.

Table 4.12: Pearson coefficient values on SCCM matrix of PHP systems

PHP SYSTEMS										
SYSTEM	SCCM	COH	PM	PRM	PA	PRA	PO	PRO	LV	LVUSAGE
SHOPPING CART	0.5	0.5	5	5	0	1	19	0	6	19244
CONFIGURATION	0.7543	0.7543	10	15	0	7	156	0	17	1462080
CSS	0.2333	0.2333	5	5	6	0	22	12	7	112944
GAME1	0.3333	0.25	4	0	0	3	11	0	1	0
GAME2	0.6	0.6	3	2	2	0	6	4	11	30562
BOARD	0.1667	0.1667	8	0	0	3	21	0	4	1560
CRUD	0.5231	0.2615	13	0	0	5	46	0	12	10610
REGISTRATION-LOGIN	0.4	0.4	5	0	2	0	13	0	8	6940
DSN	0.25	0.2	4	1	0	1	7	0	1	40
TICTACTOE	0.6	0.6	3	2	2	0	6	4	11	30562
			0.19655	0.57127	-0.191	0.33901	0.56	-0.168	0.8434648	0.41363

4.5 Results

The following section is informed by the data acquired in sections 4.1 and 4.2. The results have been categorized according to the effects of components that make up the SCCM metric: Effects of public methods, effects of total class variables, effects of public variables, effects of private variables and the effects of local variables.

4.5.1 Effects of Public Methods on SCCM Value

Figures 4.3, 4.4, 4.5 and 4.5 show the graphical representation of the number of public methods against the SCCM and COH metric values. The data used in these representations has been acquired from tables 4.1, 4.2, 4.3 and 4.4.

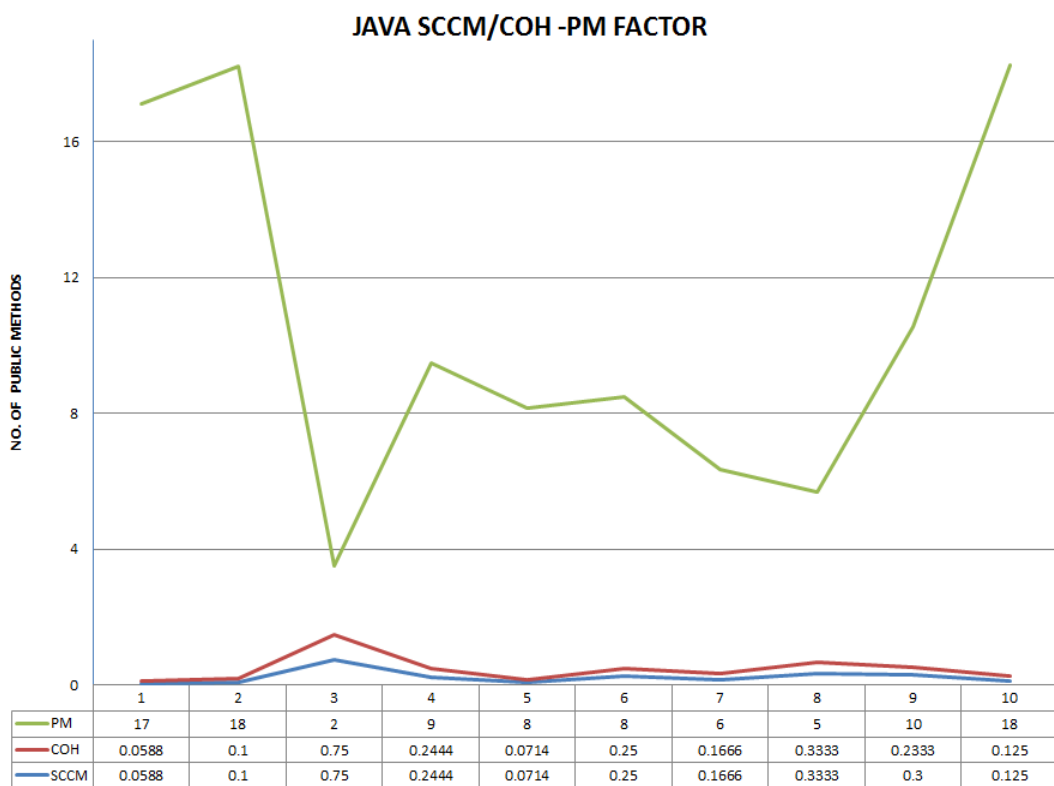


Figure 4.3: Influence of public methods on the SCCM/COH values; a case of Java classes

It is noted that the Java classes with high number of public methods from the ten systems used in the research gave lower values of the SCCM compared to those with low number of public methods. This trend is noted in all the classes with classes having smaller methods being more cohesive. In figure 4.3, systems 1 and 2 with 17 and 18 methods give the least cohesion values compared to systems 3 and 5 that give the highest cohesion values.

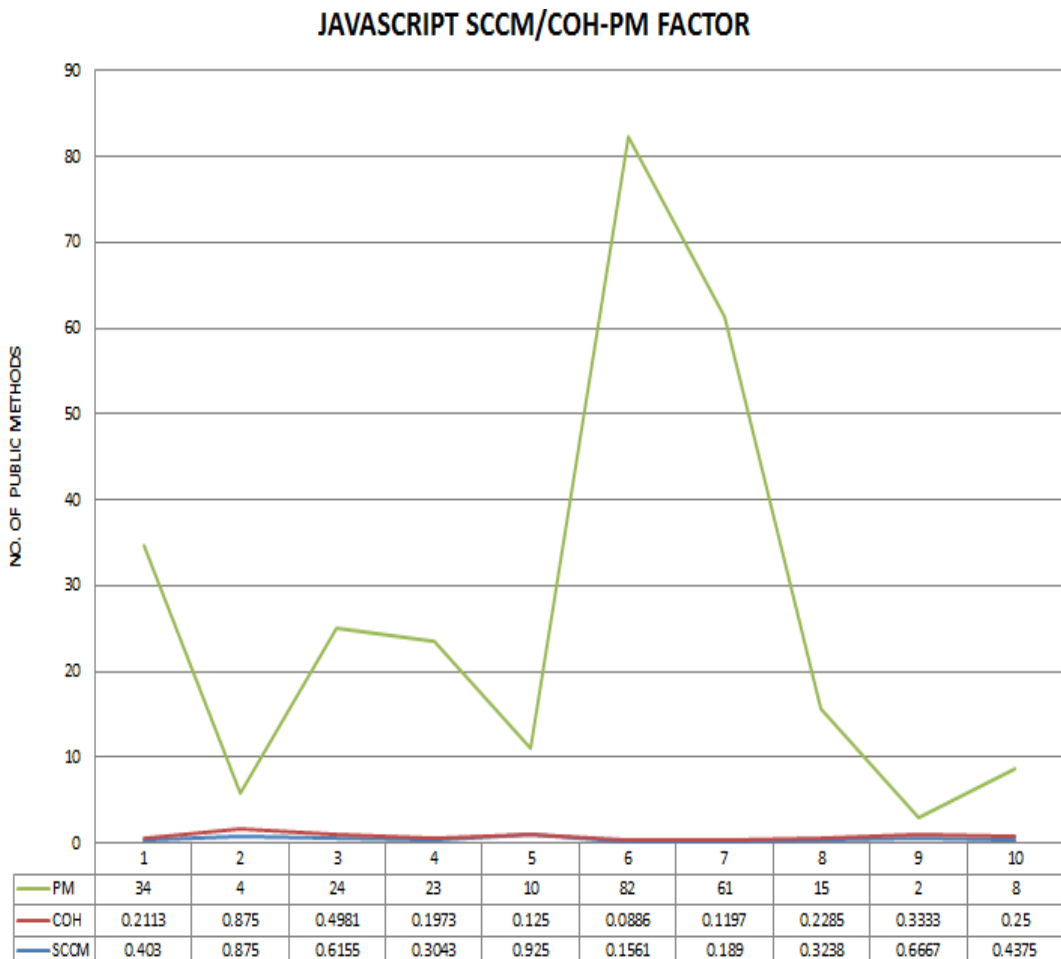


Figure 4.4: Influence of public methods on the SCCM/COH values; a case of JavaScript classes.

As depicted on the figure 4.4, JavaScript classes with lower public methods were found to have higher cohesion values for example, systems number 2 and number 9 have which have 4 and 2 public methods respectively. These two systems classes were also among the classes with high metric values.

Among systems classes in the PHP cluster, the same was evident with classes having higher number of methods giving lower SCCM values as shown in the figure 4.5 below.

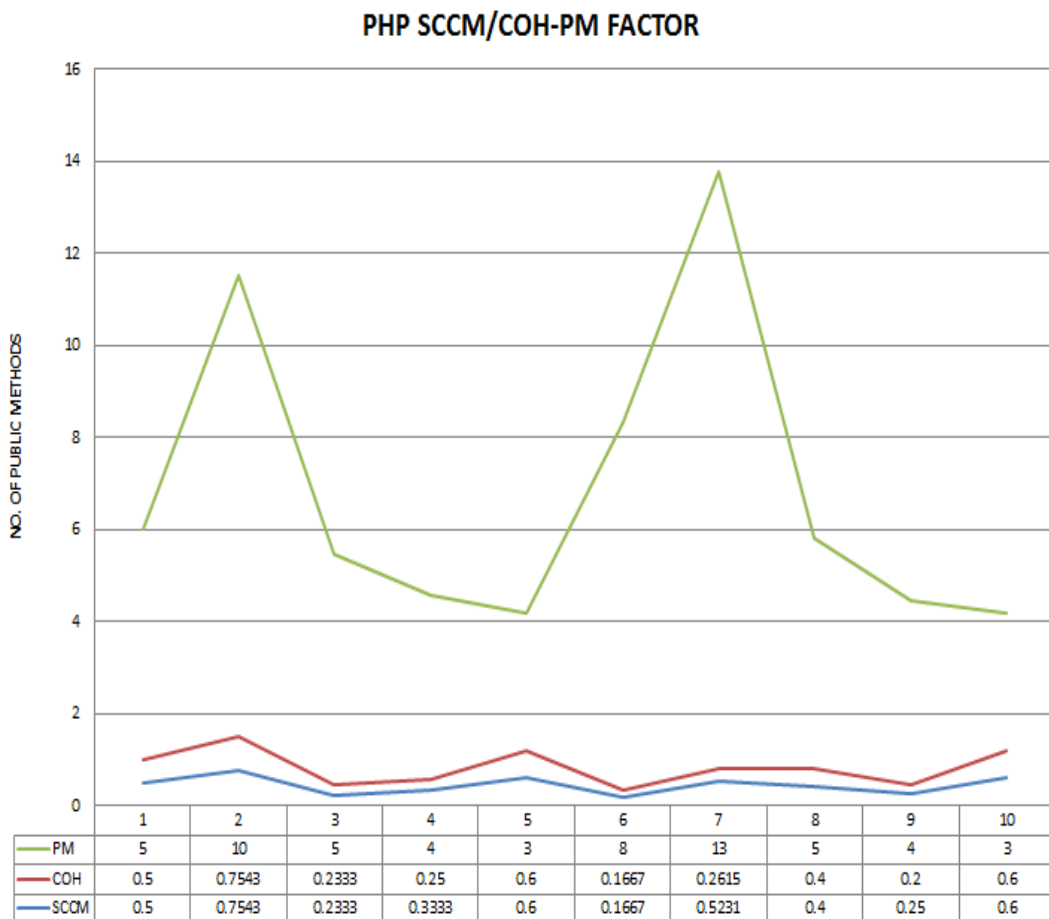


Figure 4.5: Influence of public methods on the SCCM/COH values; a case of PHP classes.

In figure 4.5, it was noted that classes with higher number of public methods still exhibited lower cohesion values compared to those classes with lower numbers of public methods.

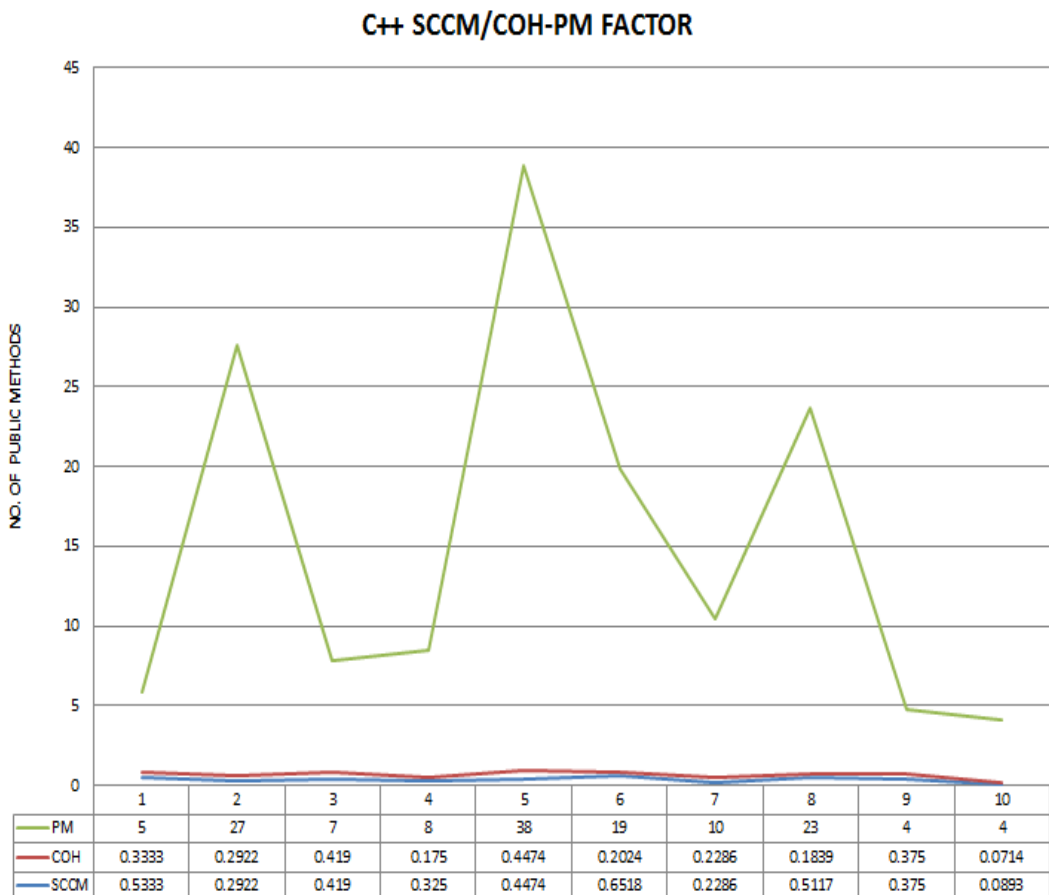


Figure 4.6: Influence of public methods on the SCCM/COH values; a case of C++.

Lastly in the C++ classes, the systems with lower number of public methods also exhibited higher cohesion values compared to those classes with higher number of public methods. It can therefore be stated that lower cohesion was associated with higher number of methods within the four clusters.

4.5.2 Effects of Total Class Variables on SCCM Values

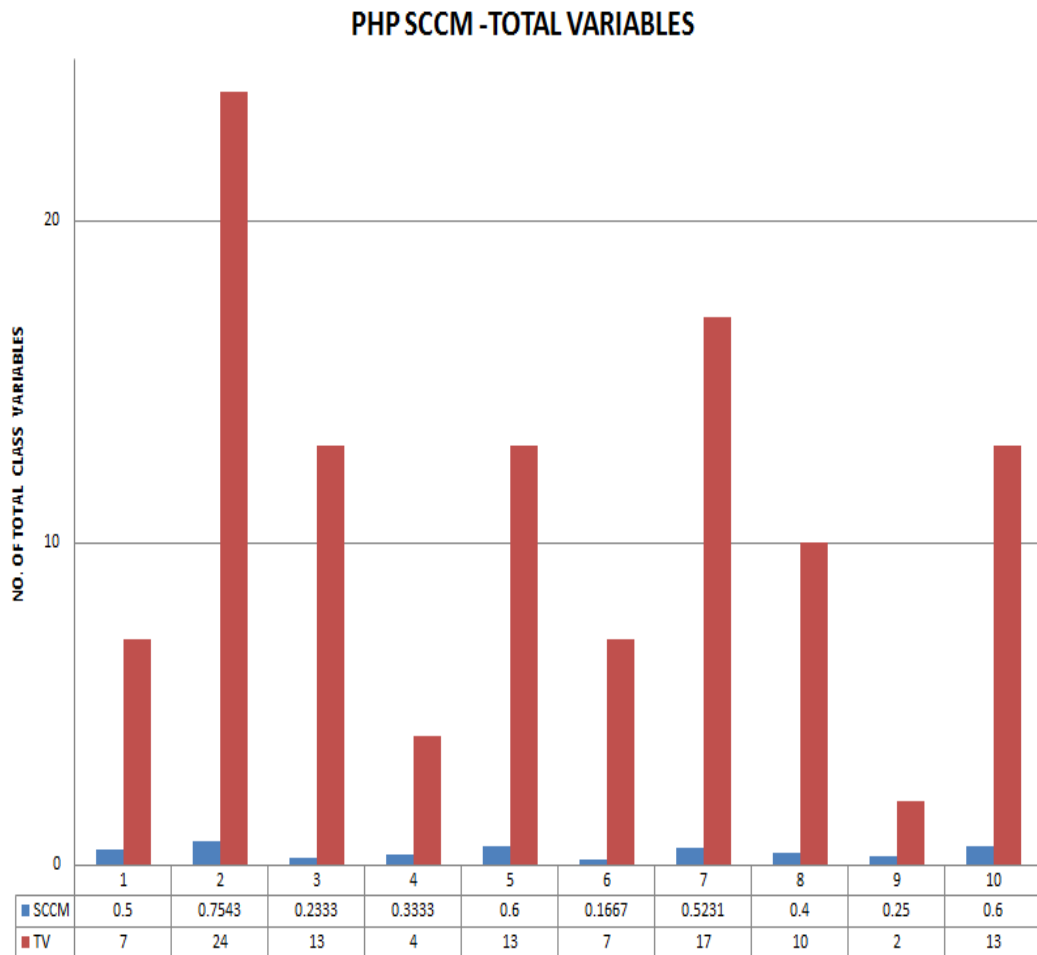


Figure 4.7: Influence of total class variables on the SCCM/COH values; a case of PHP.

As the total number of variables within a class increased, the values of the SCCM decreased as evident in figure 4.7 with systems 2, 5, 7 and 10 (acquired from table 4.4) giving high values of the SCCM. This trend is also noted in figure 4.8 when considering C++ classes (acquired from table 4.2).

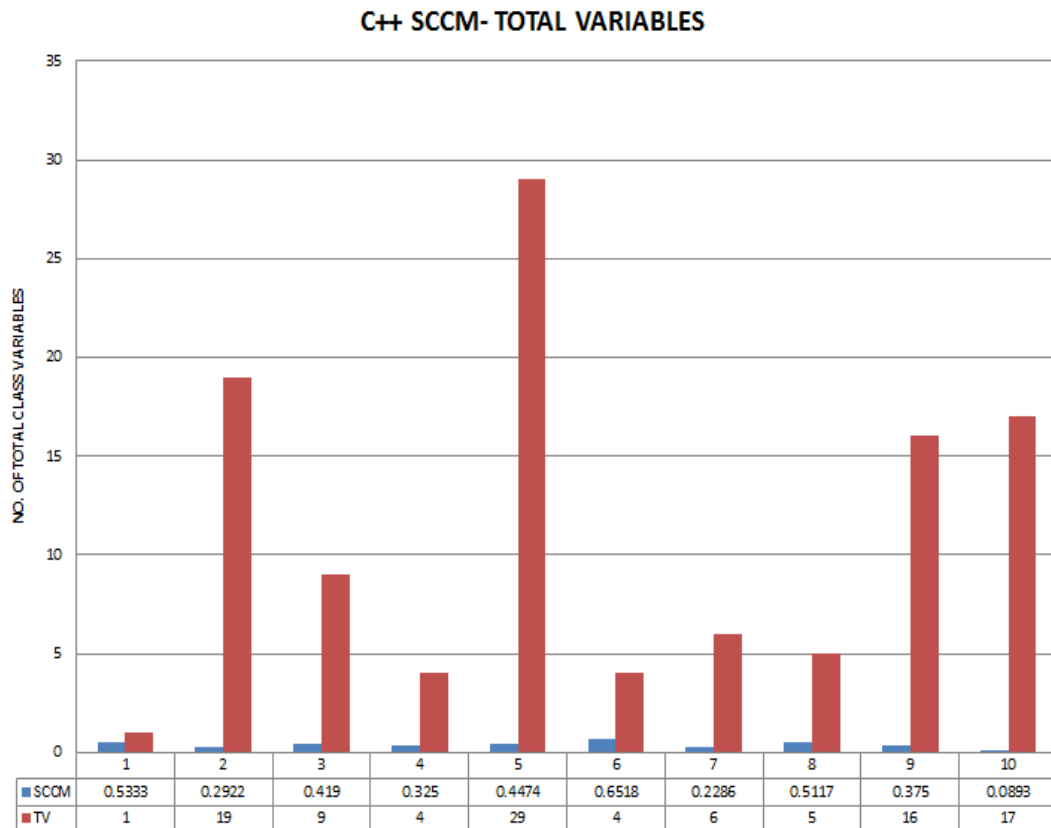


Figure 4.8: Influence of total class variables on the SCCM/COH values; a case of C++.

The same trend is also seen in the Java Systems classes (acquired from table 4.1) as depicted in figure 4.9. One of the reasons why the variables increment could be associated with very low cohesion value is possibly due to the fact that the class members' distribution within the class becomes more therefore increasing communication coupling which ultimately leads to reduced cohesion within the class.

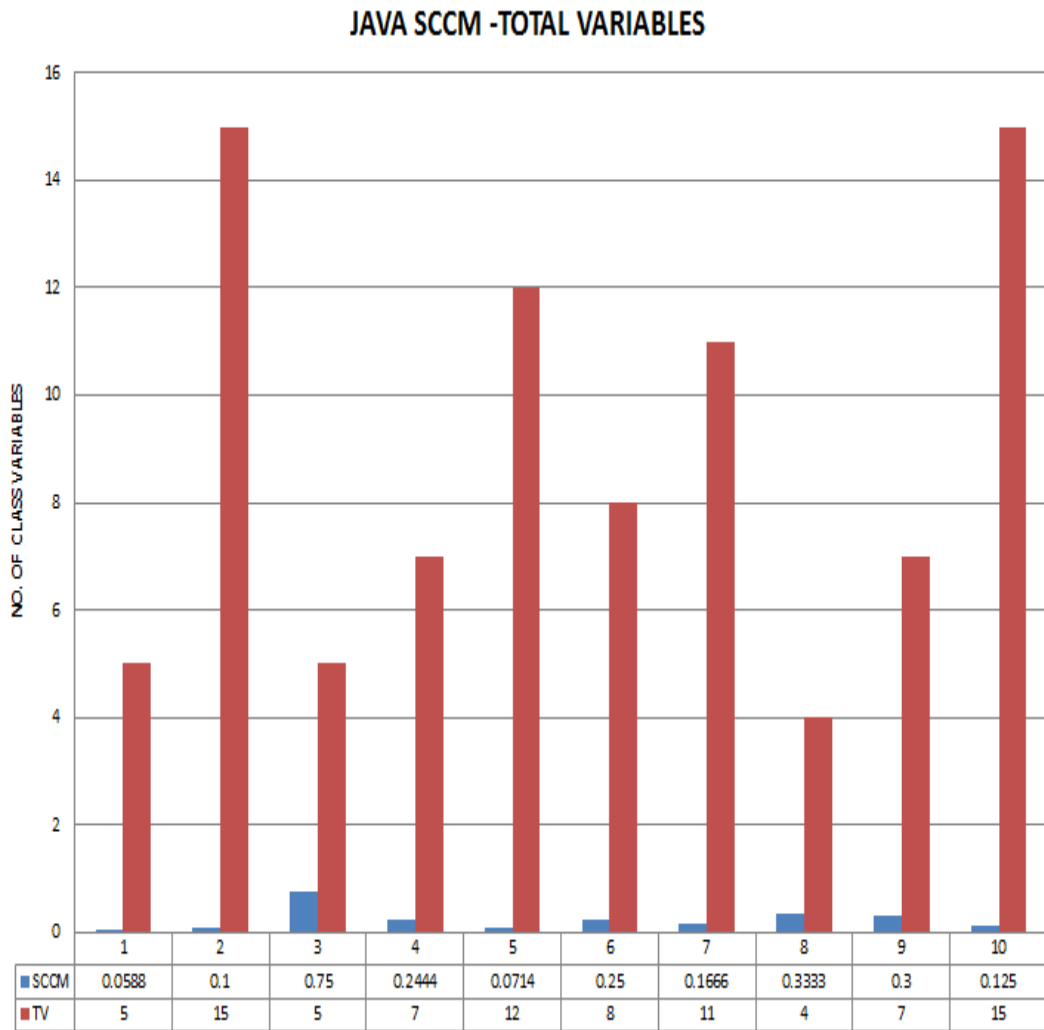


Figure 4.9: Influence of total class variables on the SCCM/COH values; a case of Java.

In the JavaScript classes high total number of variables was also noted to have the least cohesion with system classes 6 and 7(acquired from table 4.3) in figure 4.10 having the highest number of variables but also the least cohesion among the ten studied JavaScript systems. This trend is a similar trend like the one seen among the

public variables.

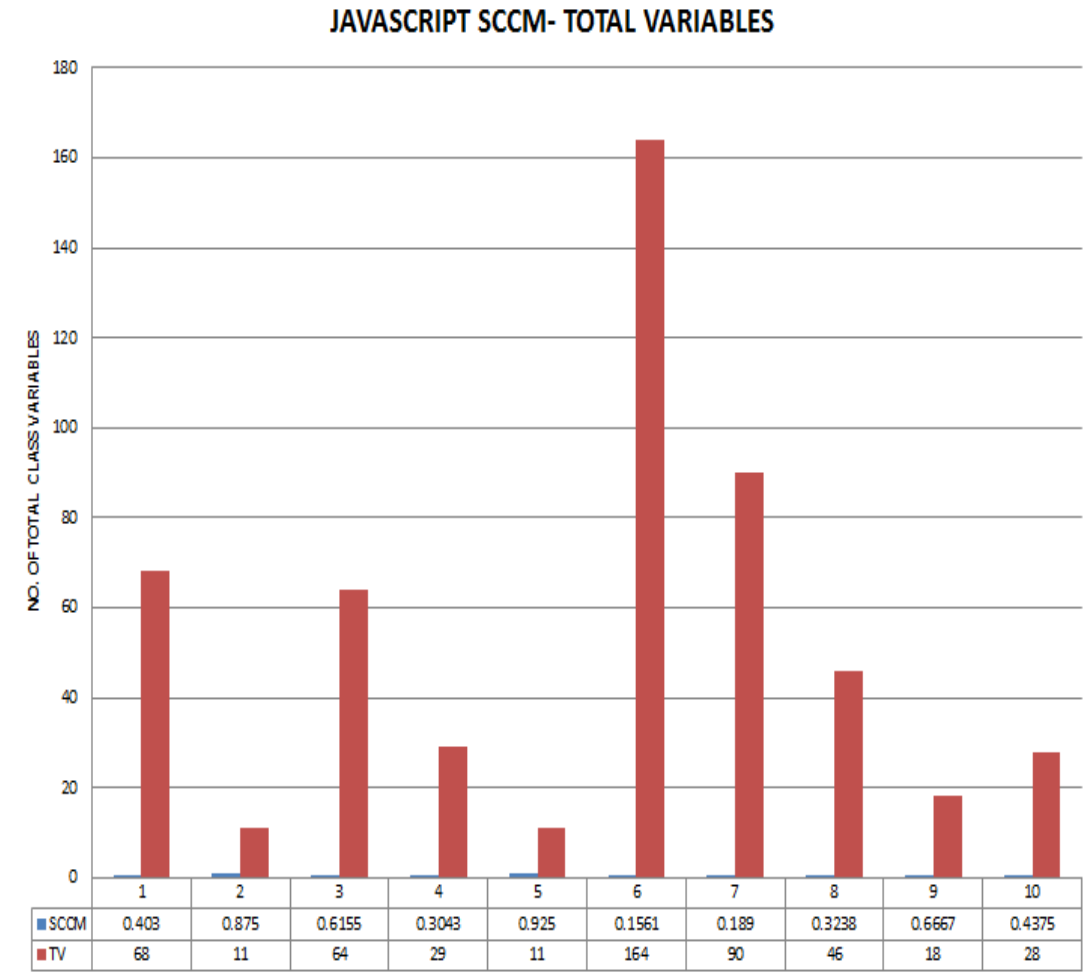


Figure 4.10: Influence of total class variables on the SCCM/COH values; a case of JavaScript.

4.5.3 Effects of Public Variables on SCCM Values

The number of public variables within a class also displayed a similar trend like the effects of public methods: As the number of variables got lower, the values of the SCCM also decreased. This was evident in many systems (acquired from table 4.2) in the C++ cluster of systems as shown in figure 4.11.

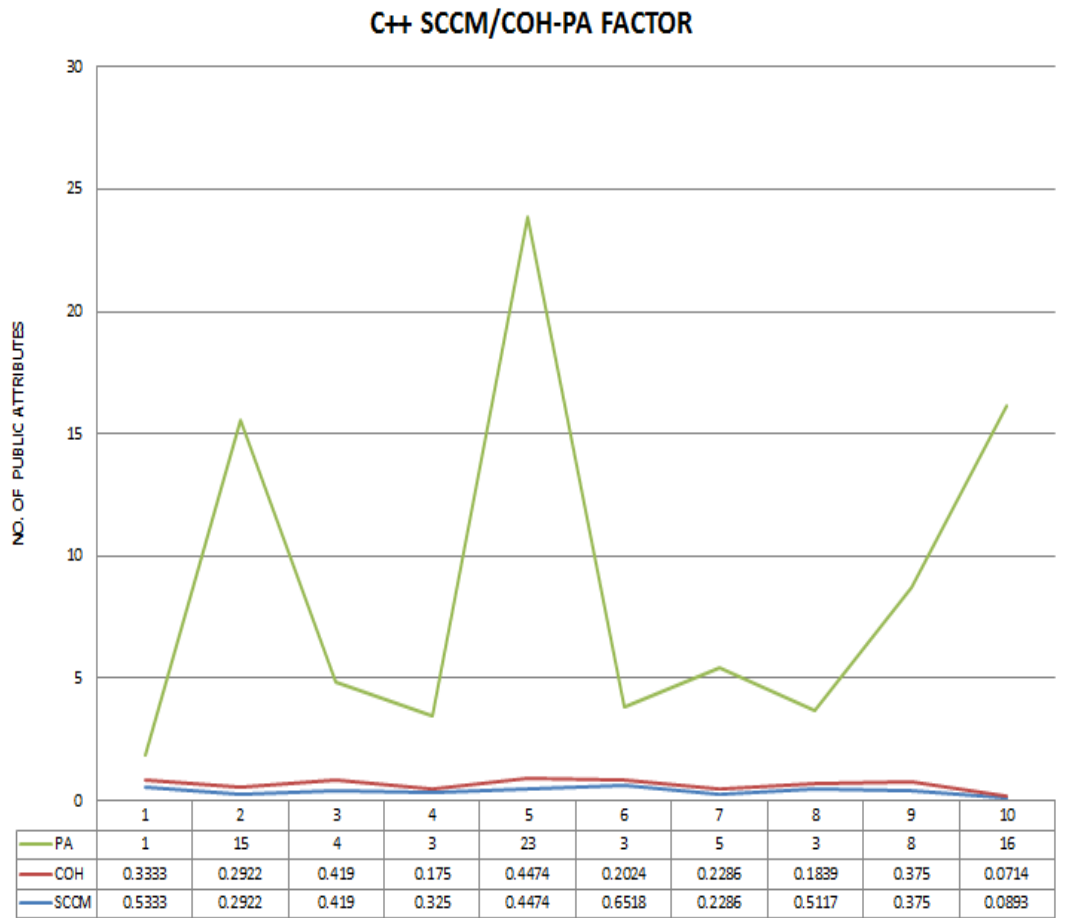


Figure 4.11: Influence of public variables on the SCCM/COH values; a case of C++.

Similarly the trend was noted in the Java Systems (acquired from table 4.1) as evident in figure 4.12.

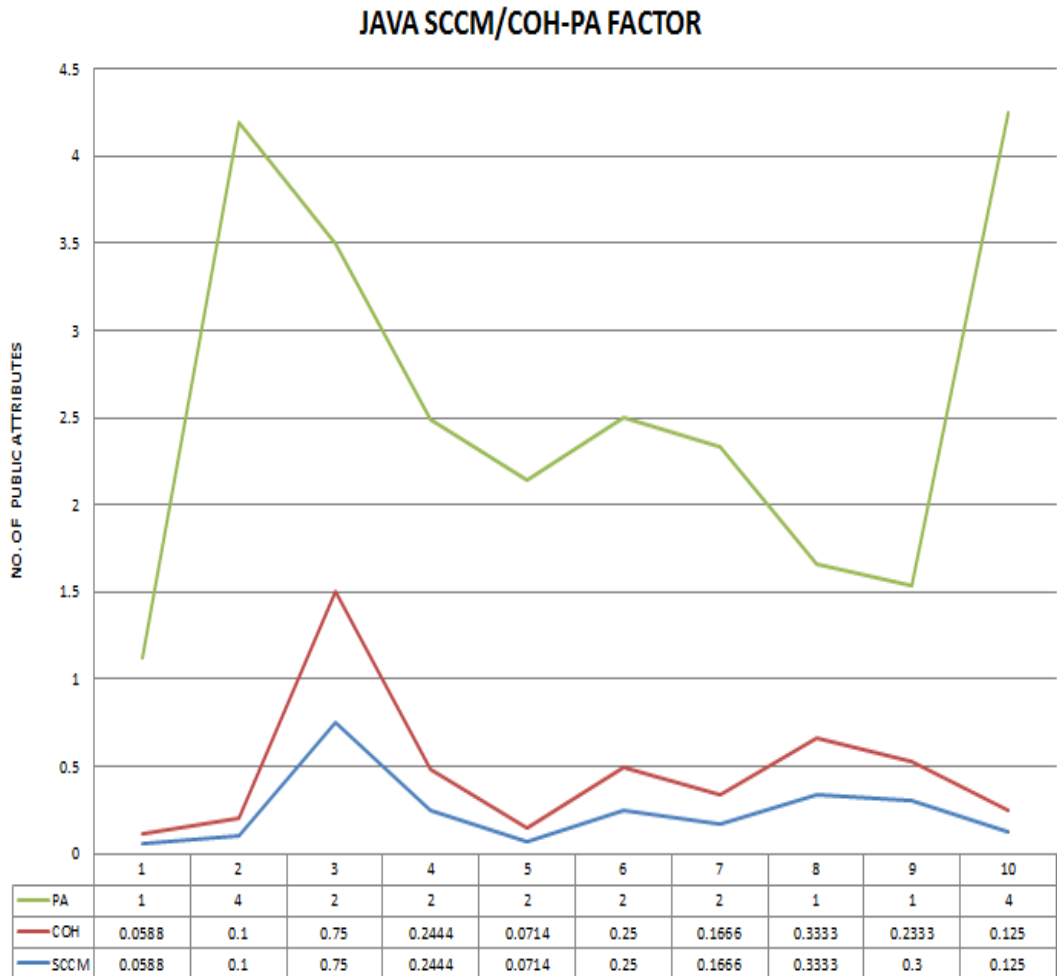


Figure 4.12: Influence of public variables on the SCCM/COH values; a case of Java.

However, in Java the values of SCCM were influenced by private variables in the classes giving an unusual changes compared to the C++ systems classes shown in figure 4.11. Despite that trend, classes with higher number of public variables still gave lower SCCM values compared to those with lower number of public variables.

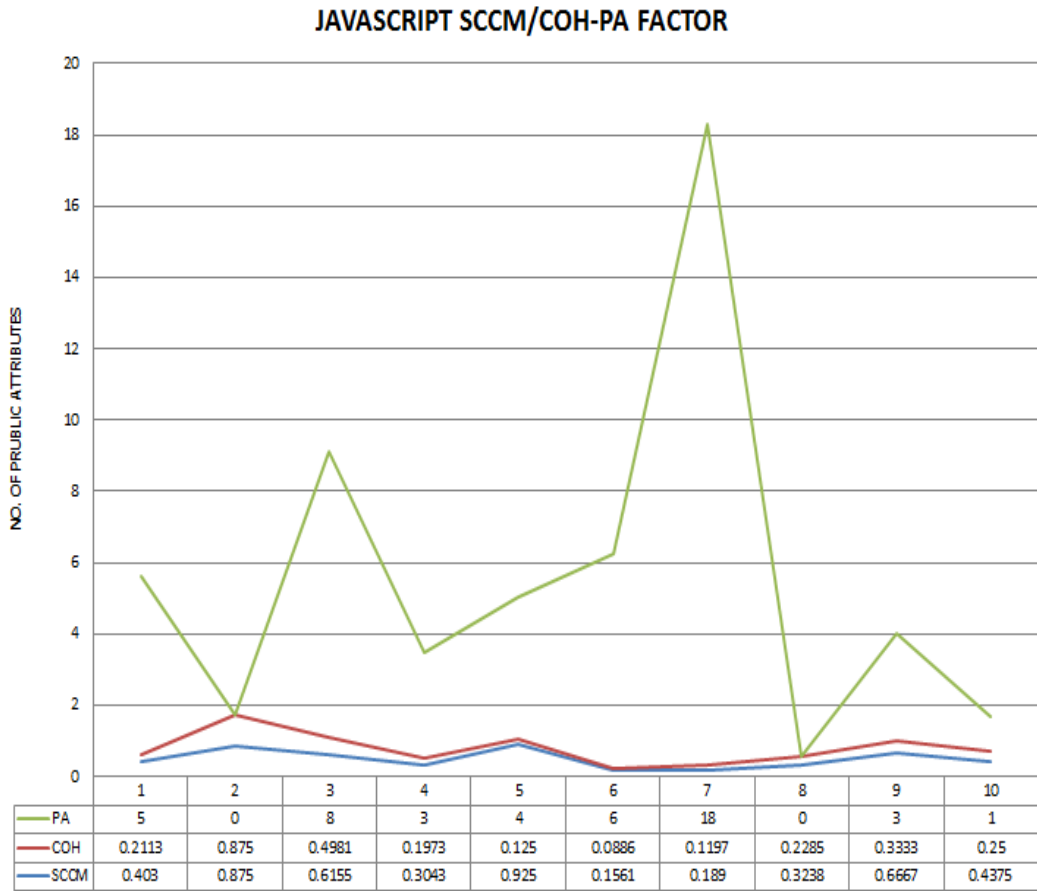


Figure 4.13: Influence of public variables on the SCCM/COH values; a case of JavaScript.

In the JavaScript systems the values of SCCM increased significantly with reduction in the number of public variables within a class. As evident in figure 4.13, system 2 (acquired from table 4.3) which had private variables gave the highest cohesion value compared to other for example, system 7 that had the highest number of public variables but with the lower SCCM values. However, the effects of private variables within a class cannot be ignored in the contribution towards high or lower cohesion values.

In the figure 4.14 below, the SCCM values of classes with lower number of public variables are still lower compared to those of classes with higher number of public

variables. SCCM distinction among the PHP classes is noted where despite some classes having zero (0) number of public methods, they still possess different SCCM values. This is a trend noted from previous cases where a class has some private methods that influence the values of SCCM of that given class.

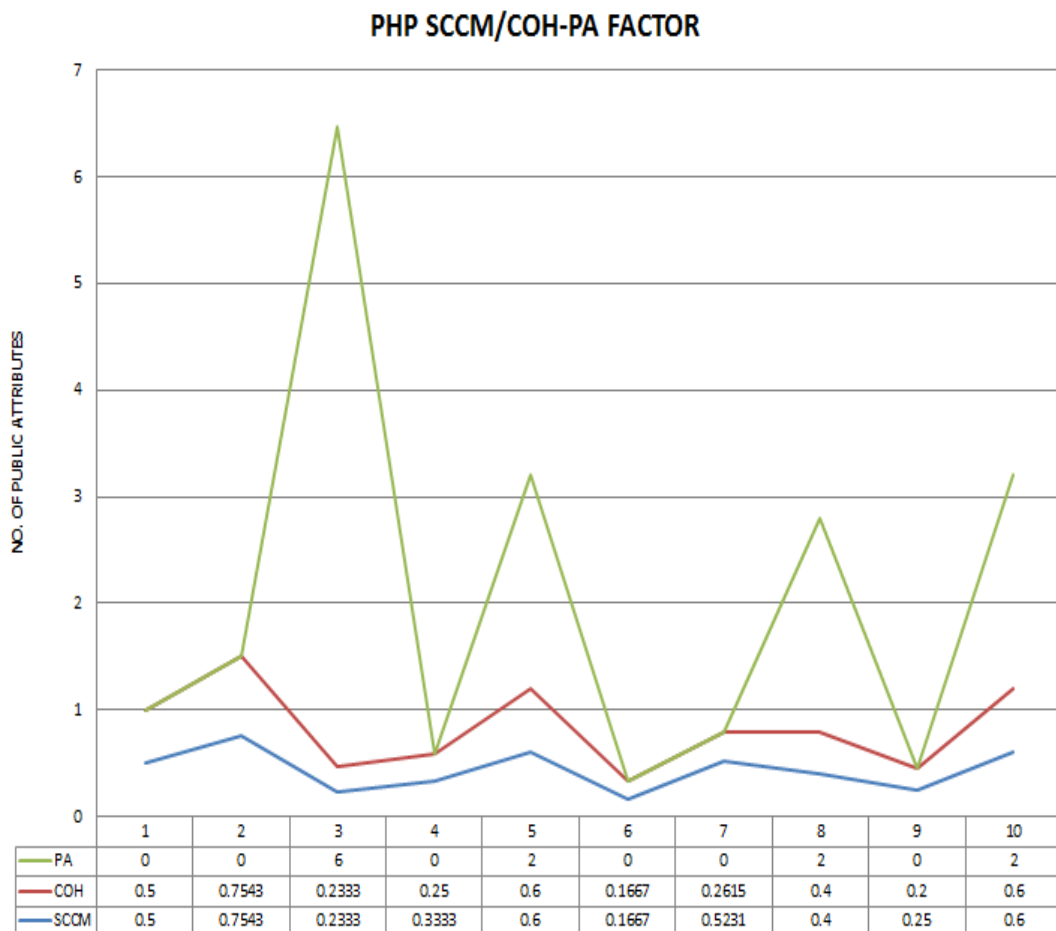


Figure 4.14: Influence of public variables on the SCCM/COH values; a case of PHP.

4.5.4 Effects of Private Variables on SCCM Values

Private members of a class have been noted to have some effect on the SCCM values in the previous mentioned results. Moreover, when looked critically from a private members perspective, these variables behave in a similar manner just like the public

variables. An increase in the number of private variables of a class leads to lower cohesion values of the SCCM as evident in the figure 4.15.

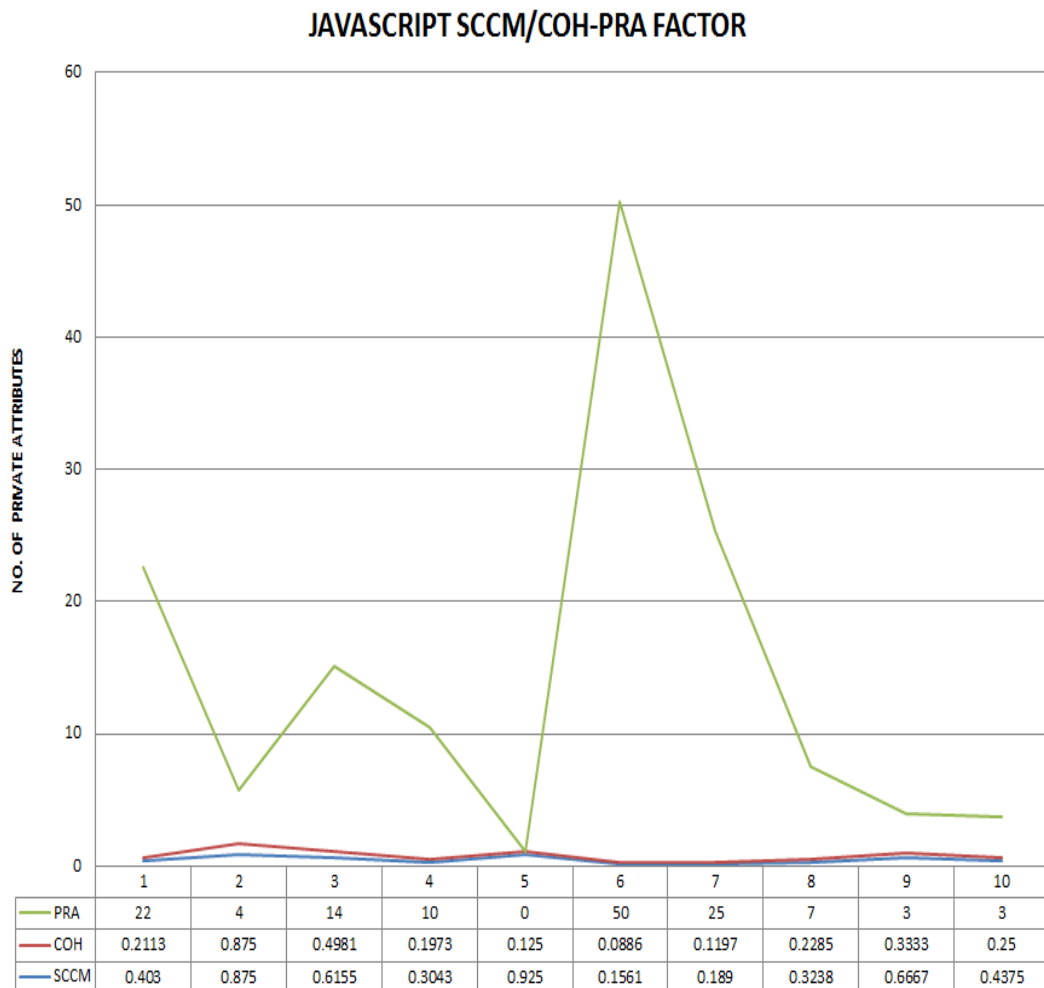


Figure 4.15: Influence of private variables on the SCCM/COH values; a case of JavaScript.

The same trend is noted on the PHP values with private variables increment leading to lower cohesion values as represented in figure 4.16.

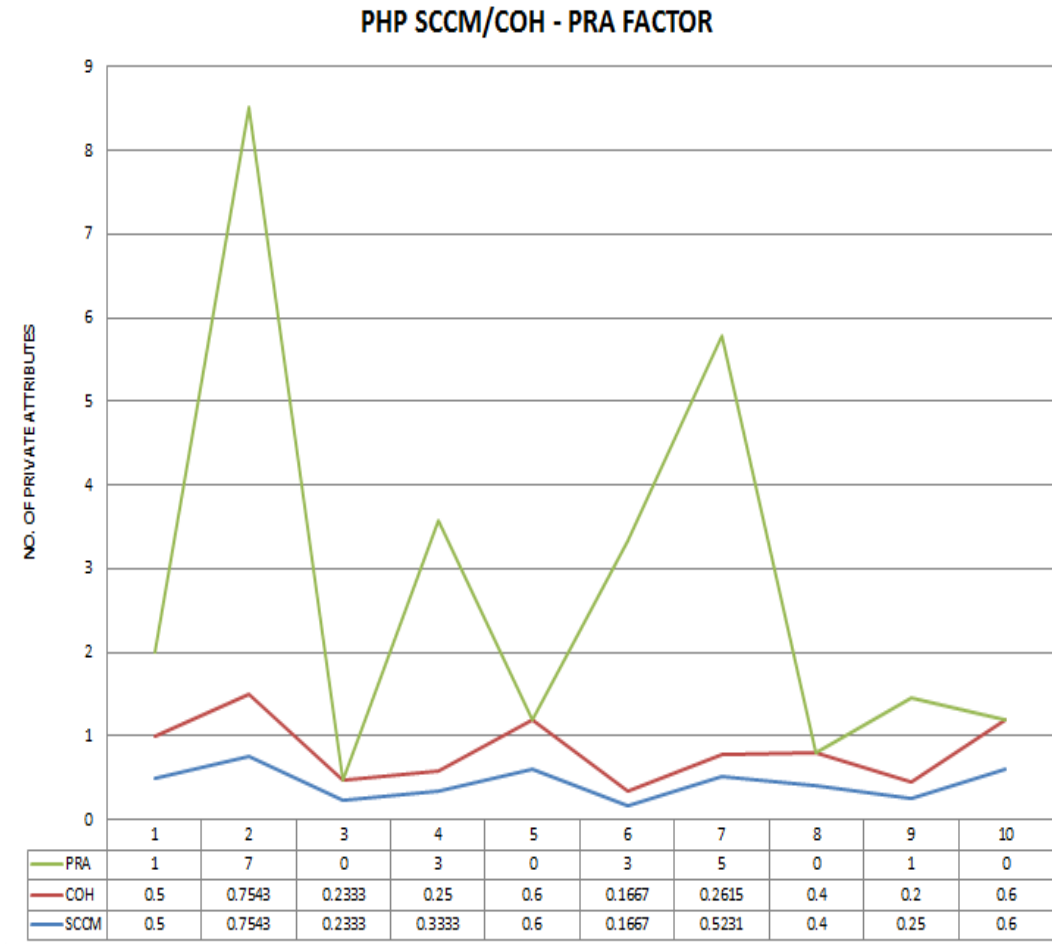


Figure 4.16: Influence of private variables on the SCCM/COH values; a case of PHP.

4.5.5 Effects of Local Variables on SCCM Values

Local variables within a system class are vital since they add muscle to a specific module of their usage. In this research, local variables effects were studied and their contributions recorded in order to give an insight of their influence. Figure 4.17 gives a trend of the number of local variables in a class increased the values of the SCCM decreased accordingly. This trend is similar to the increase in the number of public and private variables noted earlier.

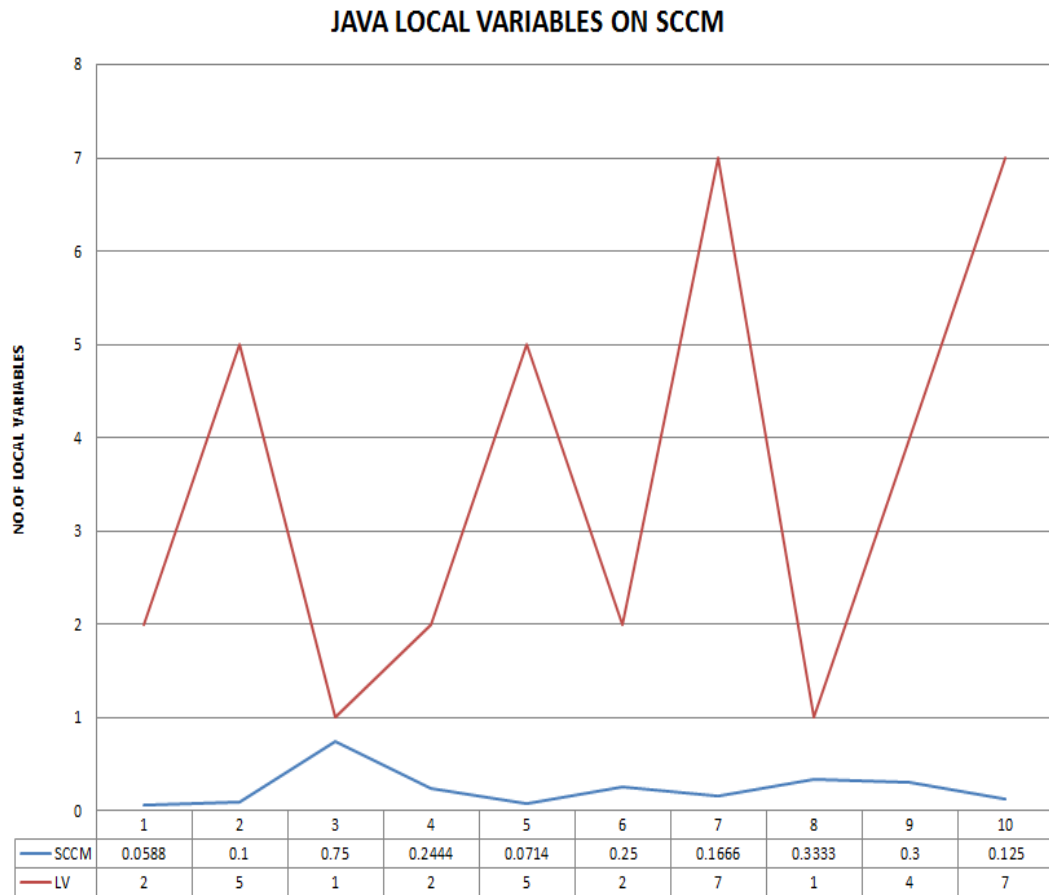


Figure 4.17: Influence of local variables on the SCCM values; a case of Java.

Among the JavaScript systems, similar results are also achieved with systems 2, 5 and 9 giving very high cohesion values compared to the other systems that had very high number of local variables.

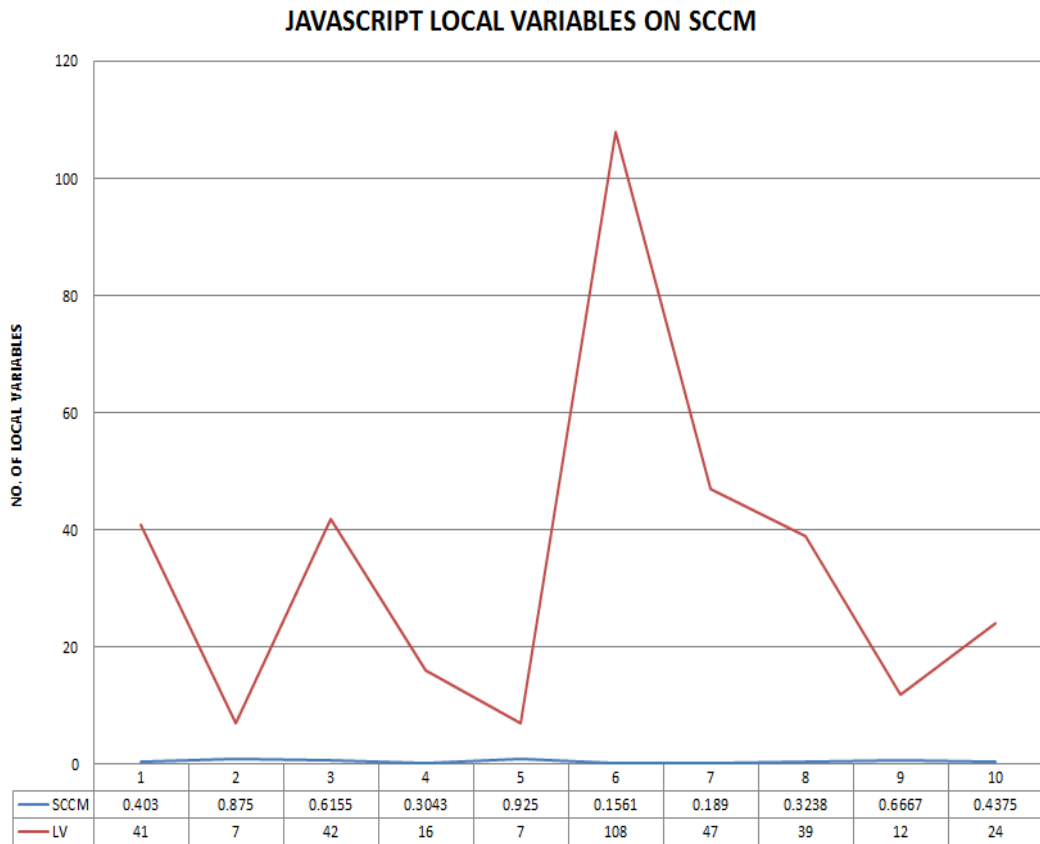


Figure 4.18: Influence of local variables on the SCCM values; a case of JavaScript.

In the PHP systems (shown in figure 4.19) the same trend was noticed with value spikes noted on the systems that had high values compared to those that had low values. This trend had been noticed on the previous cases with the number of public variables within a class in figure 4.18.

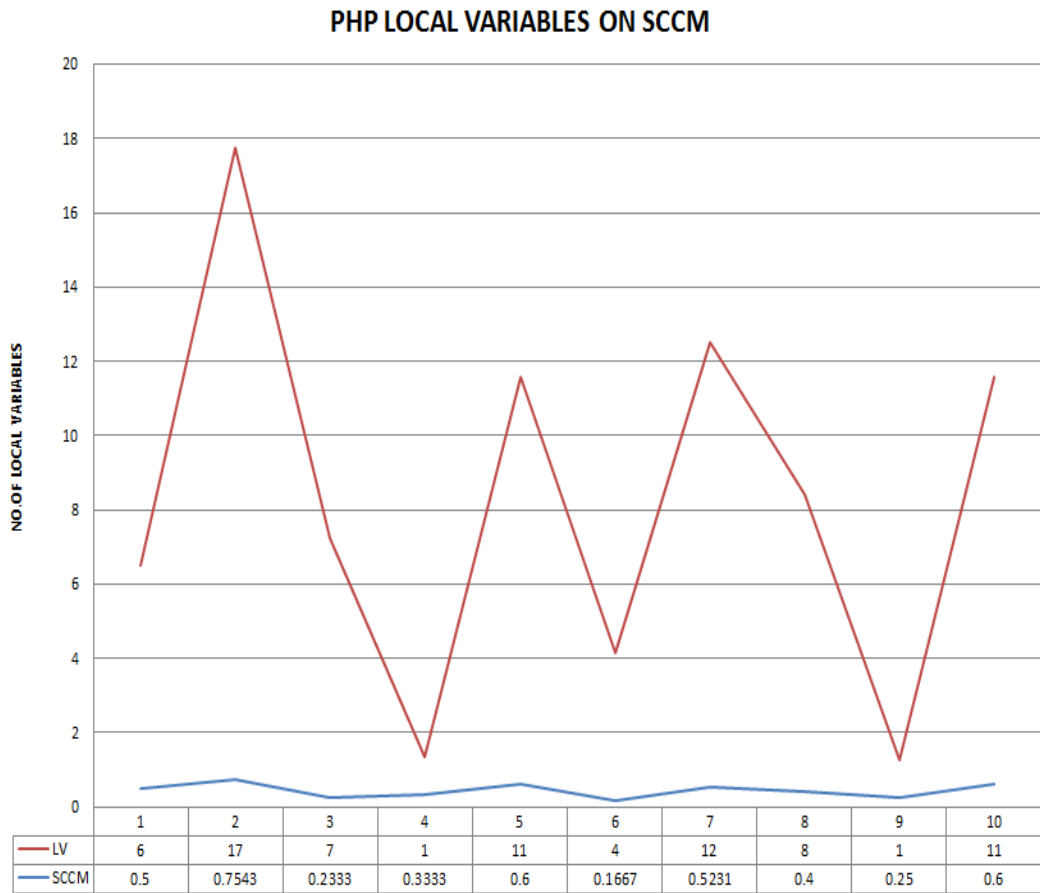


Figure 4.19: Influence of local variables on the SCCM values; a case of PHP.

4.6 Discussion

4.6.1 Discussion of Results of the Effects of Public Methods

Public methods are functionalities or modules that are explicitly available in a class (PHP, 2016). Classes with higher number of public methods were noted to have the least SCCM values whereas classes that possess private methods and variables were found to be more cohesive than those that did not utilize both scoped variables and methods. This effect had been reported in COH and LCOM5 metrics where if the

number of methods increase without equal or more proportionate increase in the number of fields occurrences leads to lower cohesion (Okike, 2010).

A small class (one with fewer numbers of methods) is able to handle things much better as stipulated by Cunningham(2004).Furthermore, Cunningham outlines a class with more than 20 methods is considered “overworked” whereas a class with 10 methods is said to be doing a lot of work. In an extension review of modularity by Pressman (2010) on the modularity principles of design, a given module can be exploded into two or more modules which results when shared processing is noticed in two or more modules. Exploding the module reduces coupling creating well maintainable and cohesive modules.

A class found with many methods makes it hard to maintain and to adapt to an environment. This can be well explained by the Law of Demeter. This law states that given for any class A, a method M may only invoke other methods belonging to the following:

- Class A
- Members of A
- Parameters of M
- Objects created by M
- Objects created by other methods of class A called by M
- Globally accessible objects

This law therefore ensures that a given object has the least knowledge of another object and never places a call to another object through an intermediate object as outlined by

Lieberherr (2006). This law enables the designing of systems to be loosely coupled hence increasing cohesion of a class. When the objects of a given class know less about another object (a provision that is provided through the methods of a class) then the reliance is reduced on the internal structures of that class and even changes on that object negatively result to very light issues that may emerge making the code fairly maintainable. This therefore gives a good reason to have fewer methods in a class ensuring objects independence is enhanced reducing any anticipated “ripple effect” through the entire class. (Bock, 2012).

Following the Demeter Law is crucial and important in class design since encapsulation is improved. Information hiding or encapsulation is one of the key concepts that object oriented systems try to achieve and if it can be done through a reduction in the number of methods of a class, then it is considered a key thing to be done on a system.

When the number of methods in a class is reduced, the code becomes more maintainable and cleaner as stated by Miskov (2011) during one of the Google Tech talks. In a paper written by Guo et al (2011), following this concept in the development of object oriented systems is one of easiest ways of bugs’ reduction and not following makes it one of the easiest ways of increasing bugs within a system which ultimately leads to poor quality software. Figure 4.20 shows a representation of this concept.

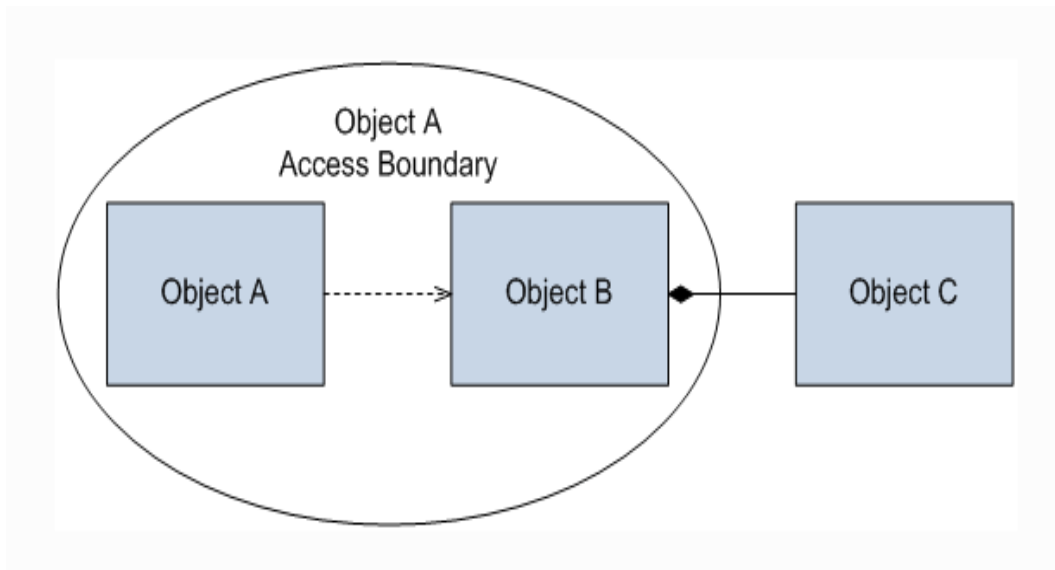


Figure 4.20: A diagram illustrating the Law of Demeter.-Adapted from Lostechies

It can therefore be concluded that presence of many methods within a class makes that class a good candidate for inspection due to a higher likelihood of bugs. The class also becomes strenuous to maintain (Rosenburg, 2012) and adapt to environments which increases coupling among modules and consequently lower cohesion.

4.6.2 Discussion of Results of the Effects of Total Class Variables

Every instance of a class shares a class variable (Oracle, 2015).When a class is developed, objects can be created from it. When accounting for class cohesion, a class is expected to achieve one common thing which is defined in it and its actualization is done through the object. As noted from the results, a class with lower number of variables was noted to be more cohesive than one with higher numbers of variables a similar case reported by Lustman, Keller and Kabaili (2001) and cohesion values only increase if the number of their reference within the methods is higher and not just their numbers.

As the total number of variables within a class increased, the values of SCCM were noted to reduce as evident in figures 4.5, 4.6, 4.7 and 4.8. This is because as the variables and their usage increase within the various modules, then the dependence among them also increase which consequently leads to higher coupling and reduced cohesion.

Although communication is increased when variables communicate with each other and resultant data flow increase, cohesion reduces because as one element is changed then changes occur and due to the high number of elements that exist then chances of many errors bulge, interdependencies increase and even the functional cohesion of each module is reduced. However, if a small number of variables are used and even reused (via inheritance); data control is enhanced since each module is able to work within its functionalities and roles to the system and only passes what is required to the next module (Washington, 2013). When there are too many variables within a system then the interdependencies weaken these roles for the different modules.

4.6.3 Discussion of Results of the Effects of Public Variables

Public variables are those class members that can be accessed anywhere within a class. Classes from systems with higher public attributes gave lower SCCM values a similar situation reported in the CC metric. The use of high number of public variables in a class was also noted to have a negative correlation with cohesion values.

High usage of public variables has been cited by Cunningham (2013) to breaking immutability of objects consequently affecting encapsulation of an OOP class whereas a class with low number of public variables has also been proven to yield clearer,

understandable code of high maintainability when global variables are avoided (Cunningham, 2013). This is as a result of reduction in the non-locality of variables that can be read and modified by any member of module of the class which makes it easier to remember in its every possible usage.

The use of high public variables also makes it hard to control the access of data within a class since they are accessible by any other module or element within the class. This may breach the security of the system and possible intrusion by bugs. Furthermore, public variables should be avoided since they introduce implicit coupling due to lots of inter-variables dependencies between modules and variables. Whenever there is high coupling then cohesion of a class reduces significantly.

A class with high number of public variables makes bug tracking a very strenuous process since an error could be anywhere within the entire code. It can therefore be concluded that values of SCCM tend to decrease with increase in the number of public variables occurrences due to implicit and common coupling (access of global communication) which makes it difficult to understand the use of data, causes inflexibility, introduces dependence clusters (Binkley, 2009) and causes potential ripple effect (Bansal, 2013). It should be noted that public variables within a class should only be introduced and used when and only when it is absolutely required.

In a paper done by Kulkarani and Hemaiyar (2013), global variables have been negatively campaigned for use within a class due to the following reasons. Firstly, they increase the mental effort necessary in the undertaking abstraction in a program making understandability by developers cumbersome. Secondly, the use of public variables makes it hard for developers to test their code and even verification of the

software. This is coupled with difficulties in modifications of the memory locations since it is hard to know their usage state due to their explicitness as they are passed and returned to functions. Thirdly, they also increase program dependence which connects one module to another.

Despite the condemned usage of public variables in a class two alternatives have been offered by Cunningham (2013); the singleton pattern and the hidden objects. Singleton pattern involves creation of a single global object which is accessible through stateful procedures. Stateful procedures involve the use of global getters and setters or functionalities that implicitly act over the underlying state. The use of stateful procedures ensures that globals are localized thereby minimizing many probable linking problems. According to tutorials point (2015), this class provides a way of accessing the object directly without the need to instantiate the object directly.

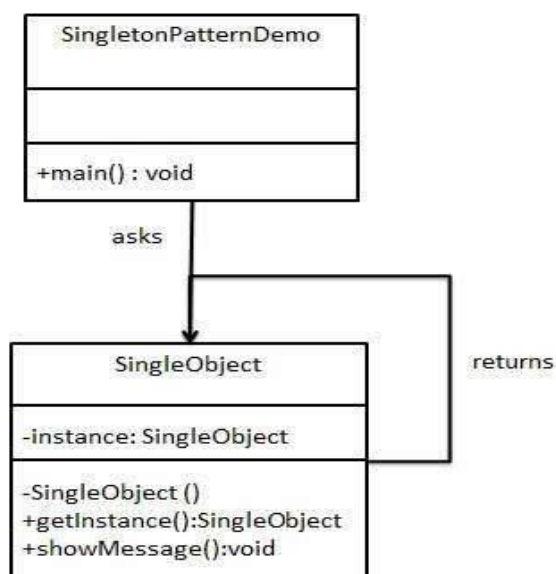


Figure 4.21: An illustration of singleton pattern in OO class design-Adapted from tutorialspoint.com

The second option is the use of hidden objects. Hidden objects are used where globals are tamed for local use. When the globals have been localized they are passed as arguments to the various functions that require them in their local scope. This is a concept that has been successfully implemented in creation of JavaScript objects that uses a self-referencing *this* prefix. The prefix is used for creating an object's properties creating abstraction of the public variables despite the negative data storage ramifications (Gravelle, 2014). The effect of localizing these variable is that they are then visible within the method scope where they are needed which greatly reduces the dependence on the other sections of the program.

There are other two usefulness of global variables namely; efficiency and convenience as discussed by Kukarni and Hemaiyer (2013). In improving code efficiency, global variables have been shown to boost program efficiency and at the same time lowering the storage memory space through reduction or elimination of arguments passed to functions as well as values that are returned or called by the functions. Convenience within the program is added to the code by the global variables through their declaration in the outer scope which makes access of the stored values more easier as compared to the local variables which can only be declared and used within a given function. However, even with this convenience it may still cause trouble in the long run, but it keeps the worry of deciding when to declare the variables and the developer is now only concerned with using them.

4.6.4 Discussion of Results of the Effects of Private Variables

According to George (2013), private variables are class members that are normally declared as private and cannot be referred to from other classes. Within the SCCM metric, private attributes were noted to contribute to class cohesion negatively as shown on tables 4.1 and 4.3 with the negative Pearson's coefficient and they also non-influenced the metric if the class is inherited. This is because private members cannot be accessed outside a class (Dammers, 2012). However, the calculation of the SCCM values does not account the use of constructors and destructors because they artificially increase the cohesion value (Dallal, 2011). Inherited attributes and methods (directly or indirectly) were factored in to cater for inheritance- which is a major concept in object oriented software development (Ibrahim et al, 2012). Classes with lower numbers of both public and private attributes gave higher SCCM values and as the total number of attributes increased, the values of SCCM also decreased.

4.6.5 Discussion of Results of the Effects of Local Variables

A local variable refers to a storage location that is declared within a method or is used as one of the method's arguments (Calvin, 2012). These variables can only be used within the function in which they have been declared in and cannot be able to store persistent data about an object between method calls.

From the analysed data, the presence of high local variables is associated with low SCCM values. However, just like reported by Lustman, Keller and Kabaili (2001), a higher usage and interaction between attributes in a given module leads to higher cohesion. This trend follows what had been previously identified in the results of the

effects of public and private variables as noted in tables 4.1 to 4.4 in the LVUSAGE column.

Furthermore the importance of local variables and their usage in calculating class cohesion value is also reflected by the negative Pearson's correlation value given in tables 4.6 and 4.7. This clearly shows that they are equally important just like the public and private attributes when calculating SCCM values.

CHAPTER FIVE

SUMMARY, CONCLUSIONS AND RECOMMENDATIONS

5.1 Summary and Conclusion

In all the four sampled clusters, the values of SCCM were found to be slightly higher than those of COH this is evident from the tables 4.1,4.2,4.3 and 4.4 where the values of SCCM are higher than the geometric mean which is an average of both SCCM and COH. This is as a result of accounting method calls by SCCM, a factor that is not incorporated by the COH metric. However, with the consideration of the new methodology that integrates the scoping of the occurrences' and adding up of local variables and their usage, new metric gains have been achieved by the SCCM compared to its derivative: the COH metric.

From the conducted research it can therefore be summarized that:

- i) Class cohesion can be evaluated from a scoping perspective through the use of visibility of elements that make up a class. Scoping in this research has been evaluated from the local or global scopes.
- ii) With slight adjustments to the SCCM tool, the values of LCOM 5 can also be achieved; since the COH metric is its derivative. This gives an easier way of collecting the data for developers when conducting class design tests.
- iii) Classes with higher number of public methods were noted to have the least SCCM values compared to those with lower number of public methods. Therefore, classes with large number of methods need to be broken down into

smaller classes with fewer number of methods if higher cohesion in classes is to be achieved.

- iv) As the total number of variables within a class increased, the values of SCCM decreased accordingly as a result of increased communication which ultimately leads to increased coupling and subsequently lower class cohesion.
- v) The use of high number of public variables in a class was noted to have a negative correlation with cohesion values since high number of variables increased communication coupling and subsequently led to lower cohesion.
- vi) Classes with lower numbers of both public and private attributes gave higher SCCM values and as the total number of attributes increased, the values of SCCM also decreased.
- vii) From the analyzed data, it has also been found out that local variables also play a critical role similarly to the public and private variables in enhancing data control and that large classes do not necessary mean they are cohesive compared to smaller classes. This is therefore important for developers to introduce them when necessary if at all understanding, easier maintenance, better testing and good class design is to be achieved in the long run.

Therefore, a cohesive class is characterised by low coupling (independence between modules), only introduces attributes when necessary, has a better chance of reusability, enables faster and effective testing, flexible modifiability with low maintenance costs and enhances a developer's understanding of the software product. All these attributes make up a quality software product.

5.2 Recommendations for Future Work

The future scope of this work can be extended by in the following ways. Firstly, a source code parser for the four clustered languages can be created to cater effectively for semantic and syntactical analysis of each language instead of just a JavaScript tool. Secondly, class cohesion can be evaluated from the methods perspective using the local attributes. Lastly, the analysis of the COH metric using the SCCM software can be done to enable the calculations of other metrics for example the LCOM5.

REFERENCES

- Badri, L., Mourad, B., & Fadel, T. (2011). An Empirical Analysis of Lack of Cohesion Metrics for Predicting Testability of Classes. *International Journal of Software Engineering and Its Applications*, 5(2), 69-86.
- Baig, I. (2005). *Measuring Cohesion and Coupling of Object-Oriented Systems: Derivation and mutual study of cohesion and coupling* (Master's thesis. School of Engineering, Blekinge Institute of Technology Sweden). Retrieved from <https://pdfs.semanticscholar.org/3873/723f33cfc541d76a3535d1f04ae0d6b978d3.pdf>
- Badri, L., & Badri, M. (2004). A Proposal of a new class cohesion criterion: an empirical study. *Journal of Object Technology*, 3 (4).
- Barbacci, R. M. (2004). Software Quality Attributes: Modifiability and Usability. Software Engineering Institute Carnegie Mellon University. Retrieved from <http://www.ieee.org.ar/downloads/Barbacci-05-notas1.pdf>
- Bansal, G. (2013). Software Design Coupling/Cohesion in Software Engineering. Retrieved from <http://girdhargopalbansal.blogspot.co.ke/2013/02/software-design-couplingcohesion-in.html>
- Basili, V.R., Briand, L.C., & Melo, W.L. (1996). A Validation of Object-Oriented Design Metrics as Quality Indicators. *IEEE Transactions on Software Engineering*, 22(10), 751-761.
- Bieman, J., & Kang, B. (1995). Cohesion and reuse in an object-oriented system. *Proceedings of the 1995 Symposium on Software Reusability*, Seattle, Washington, United States, 259–262.

- Binkley, D., Harman, M., Hassoun, Y., Islam, S., & Zheng, Li. (2009). Assessing the Impact of Global Variables on Program Dependence and Dependence Clusters.
- Bock, D. (2012). The Paperboy, the Wallet, and the Law of Demeter. Retrieved from <http://www.ccs.neu.edu/research/demeter/demeter-method/LawOfDemeter/paper-boy/demeter.pdf>
- Bonja, C., & Kidanmariam, E. (2006). Metrics for class cohesion and similarity between methods. *Proceedings of the 44th Annual ACM Southeast Regional Conference*, Melbourne, Florida, 91-95.
- Briand, L.C., Daly, J.W., & Wust, J. (1997). A Unified Framework for Cohesion Measurement in Object-Oriented Systems. *Software Metrics Symposium, Proceedings, Fourth International*, 43-53.
- Calvin, A. (2012). Local and Global variables. Retrieved from http://pages.cs.wisc.edu/~calvin/cs110/LOCAL_GLOBAL.html
- Chandrika, S.M., Babu, E.S., & Srikanth, N. (2011). Conceptual Cohesion of Classes in Object Oriented Systems. *International Journal of Computer Science and Telecommunications*, 2(4), 38-44.
- Chappell, J. (2012). An Introduction Software Quality. Retrieved from <http://www.infoq.com/news/2012/04/An-Introduction-Software-Quality>
- Chidamber, S.R., & Kemerer, C.F. (1991). Towards a metrics suite for object-oriented design. *Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, 26,197–211.
- Chidamber, S.R., & Kemerer, C.F. (1994). A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20, 476–493.
- Class Scopes. (2015). Retrieved from <https://msdn.microsoft.com/en-us/library/b7kfh662.aspx>

- Counsell , S., Swift, S., & Crampton, J.(2006).The interpretation and utility of three cohesion metrics for object-oriented design, *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 15(2), 123-149.
- Cunningham, C. (2004).Many Short Methods per Class. Retrieved from <http://c2.com/cgi/wiki?ManyShortMethodsPerClass>
- Cunningham, C. (2013).Global Variables are Bad. Retrieved from <http://c2.com/cgi/wiki?GlobalVariablesAreBad>
- Daghaghzadeh, M., Dastjerdi, A., & Daghighzadeh, H. (2011). A Metric for Measuring Degree of Service Cohesion in Service Oriented Designs. *IJCSI*, 83-89
- Dallal, A.J. (2011). Validating Object-Oriented Class Cohesion Metrics Mathematically. *Recent Advances in Software Engineering, Parallel and Distributed Systems*.
- Dallal, A.J, & Briand, L.C. (2010). An object-oriented high-level design-based class cohesion metric. *Information and Software Technology*, 1346–1361.
- Dallal, A.J. (2010). A Design-Based Cohesion Metric for Object-Oriented Classes. *International Journal of Computer Science and Engineering*, 1(2), 195-200.
- Dammers, T. (2012). Retrieved from <http://programmers.stackexchange.com/questions/143736/why-do-we-need-private-variables>
- Dasari, R., & Vasanthakumari, G. (2011). Fault Prediction in Object-Oriented Systems Based on C³ (Conceptual Cohesion of Classes).*International Journal of Modern Engineering Research (IJMER)*, 1(1), 113-119.

- Design Pattern-Singleton Pattern. (2016). Retrieved from
http://www.tutorialspoint.com/design_pattern/singleton_pattern.htm
- Ducasse, S., Anquetil, N., Bhatti, U., & Hora, A., C. (2011). Software Metrics for Package Remodularisation. *Cutter ANR 2010 BLAN 0219 02*.
- George, R. (2013). Using private class variables. Retrieved from
<http://www.pp.rhul.ac.uk/~george/PH2150/html/node47.html>
- Geometric mean. (2011). Retrieved from
<http://www.meracalculator.com/math/geometric-mean.php>
- Gravelle, R. (2014). Class Member Encapsulation in JavaScript: Data hiding. Retrieved from <http://www.htmlgoodies.com/beyond/javascript/class-member-encapsulation-in-javascript-data-hiding.html#fbid=5GG3z4kvPOv>
- Greer, D. (2008). Distilling the Law of Demeter. Retrieved from
<https://lostechies.com/derekgreer/2008/06/10/distilling-law-of-demeter/>
- Gorton, I. (2011). Essential Software Architecture. 2nd Ed. Springer Science.
- Gueye, B.A., Badri, M., & Badri, L. (2008). Revisiting Class Cohesion: An Empirical investigation on several systems. *Journal of Object Technology*, 7(6).
- Gui, G., & Scott, P. (2009). Measuring Software Component Reusability by Coupling and Cohesion Metrics. *Journal of Computers*, 4(9).
- Guo. (2011). An Empirical Validation of the Benefits of Adhering to the Law of Demeter. *18th Working Conference on Reverse Engineering*.
- Henderson-Sellers, B. (1996). Object-Oriented Metrics Measures of Complexity. Prentice-Hall, Inc., Upper Saddle River, NJ.

- Houston, D. (2015). Software Quality. Retrieved from <http://www.asq.org/learn-about-quality/overview/overview.html>
- Hiscott, R. (2014). *10 Programming Languages You Should Learn Right Now*. Retrieved from <http://mashable.com/2014/01/21/learn-programming-languages/>
- Ibrahim, S.M., Salem, S.A., Manal, A.I., & Eladawy, M. (2012). Identification of Nominated Classes for Software Refactoring Using Object-Oriented Cohesion Metrics. *International Journal of Computer Science Issues (IJCSI)*, 9(2), 68-76.
- Kaur, K., & Singh, H. (2012). An investigation of Design Level Class Cohesion Metrics. *The International Arab Journal of Information Technology*, 9(1).
- Kaur, R., & Kaur, T. (2013). Comparison of various lacks of Cohesion Metrics. *International Journal of Computer Trends and Technology (IJCTT)*, 4(5).
- Kaur, A., & Kaur, P. (2013). Class Cohesion Metrics in Object Oriented Systems. *IJSWS*, 2(3), 78-82.
- Kayarvizhy, N., Kanmani, S., & Uthariaraj, R.V. (2013). High Precision Cohesion Metric. *WSEAS TRANSACTIONS on INFORMATION SCIENCE and APPLICATIONS*, 10(3), 79-89.
- Hemaiyer, S., & Kulkarni, P. (2013). Source-to-Source Refactoring and Elimination of Global Variables in C Programs. *JSEA*.
- Lane, D. (2016). Values of the Pearson Correlation. Retrieved from http://onlinestatbook.com/2/describing_bivariate_data/pearson.html
- Li, W. & Henry, S.M. (1993). Maintenance metrics for the object oriented paradigm. *Proceedings of 1st International Software Metrics Symposium*, Baltimore, 52-60.

- Li, X., Liu, Z., Pan, B., & Xing, B. (2006). A Measurement Tool for Object Oriented Software and Measurement Experiments with It. Proc. IWSM 2000. (Lecture Notes in Computer Science 2006, Springer-Verlag, Berlin, Heidelberg, 2001), 44-54.
- Lieberherr, K., Palm, J., & Skotionitis, T. (2006). Demeter Interfaces: Adaptive programming without surprises. *European Conference on Object-Oriented Programming*, 477-500.
- Marsic, I. (2013). *Class Cohesion Metrics*. Retrieved from <http://www.ece.rutgers.edu/~marsic/books/SE/instructor/slides/lec-16%20Metrics-Cohesion.ppt>.
- Meyers, M., & David, B. (2007). An Empirical Study of Slice-Based Cohesion and Coupling Metrics. *ACM Transactions on Software Maintenance*, V (N), 1-25.
- Miskov, H. (2011). Clean Code Talks- Don't Look For Things. Retrieved from <https://www.youtube.com/watch?v=RlflCWKxHJ0>
- Patidar, K., Gupta, R., & Chandel, G. (2013). Coupling and Cohesion Measures in Object Oriented Programming. *International Journal of Advanced Research in Computer Science and Software Engineering*, 3(3).
- Perepletchikov, M., Ryan, C., & Frampton, K. (2007). Cohesion Metrics for Predicting Maintainability of Service-Oriented Software. *Seventh International Conference on Quality Software (QSIC 2007)*.
- Pena, R., & Fernandez, L. (2006) A sensitive Metric of Class Cohesion. *International Journal "information Theories & Applications"*, 13.
- Rajnish, K., & Sandip, M. (2014). New Class Cohesion Metric: An Empirical View. *International Journal of Multimedia and Ubiquitous Engineering*. 9(6), 367-376.

- Sharma, S., & Srinivasan, S. (2013). A review of Coupling and Cohesion metrics in Object Oriented Environment. *International Journal of Computer Science & Engineering Technology (IJCSET)*, 4(8), 1105-1111.
- Sreeja, S., & Sridaran, R. (2012). A survey on different approaches of determining cohesion based object oriented metrics. *International journal of engineering research and development, Vol 4*.
- Software Quality Attributes. (2015). Retrieved from <http://advoss.com/software-quality-attributes-maintainability.html>
- Sommerville, I. (2015). *Software Engineering (10th Ed.)*. Pearson Education.
- ISO 9126 Software Quality Characteristics. (2010). Retrieved from <http://www.sqa.net/iso9126.html>
- Understanding Class Variables. (2015). Retrieved from <https://docs.oracle.com/javase/tutorial/java/javaOO/classvars.html>
- Visibility. (2016). Retrieved from <http://php.net/manual/en/language.oop5.visibility.php>
- Yadav, S., Sunil, S., & Utpal, S. (2014). A Review of Object-Oriented Coupling and Cohesion Metrics. *International Journal of Computer Science Trends and Technology (IJCST)*, 2(5).
- Yourdon, E., & Constantine, L. (1978). *Structured Design*. Yourdon Press.
- Zhang, J., Bai, X., Yu, S., & Yang, Z. (2016). Influence analysis of Github repositories. *Springerplus*, 5(1):1268.

APPENDICES

Appendix A: Abstract of publication on IJCSI

International Journal of Computer Science Issues

More than a traditional journal...

[Home](#) | [Call For Papers](#) | [Submission](#) | [Authors](#) | [Reviewers](#) | [Editors](#) | [Publications](#)

Scoped Class Cohesion Metric for Software Process Assessment

Raphael Wanjiku, George Okeyo and Wilson Cheruiyot

Class Cohesion is an important software quality that can be used to improve software development process and the software product: process merit assessment and dependable software product. Many Class cohesion metrics measuring the relationship between methods and attributes have been developed and extensively researched. However, the use of relationships among attributes in measuring class cohesion from class scopes has been ignored and the effects of local variables on class cohesion need to be factored in the measurements. This research paper presents a new class cohesion metric that uses attributes relationships within class scopes with data collected using the SCCM software tool that was developed for the purpose this study. The results give higher metric values showing the importance of scoped relationships among these class members while giving a simpler and better interpretation of class cohesion through class attributes interaction.

Keywords: Class, Cohesion, Attribute, Method, SCCM, Scoping

[Download Full-Text](#) 

Figure A-1: Extract of SCCM publication abstract