

**STRUCTURAL COMPLEXITY FRAMEWORK AND METRICS FOR
ANALYZING THE MAINTAINABILITY OF SASSY CASCADING STYLE
SHEETS**

John Gichuki Ndia

A thesis submitted in partial fulfillment of the requirements for the award of the
Degree of Doctor of Philosophy in Information Technology of Masinde Muliro
University of Science and Technology

2019

PLAGIARISM STATEMENT

STUDENT DECLARATION

1. I hereby declare that I know that the incorporation of material from other works or a paraphrase of such material without acknowledgement will be treated as plagiarism according to the Rules and Regulations of Masinde Muliro University of Science and Technology.
2. I understand that this thesis must be my own work.
3. I know that plagiarism is academic dishonesty and wrong, and that if I commit any act of plagiarism, my thesis can be assigned a fail grade (“F”)
4. I further understand I may be suspended or expelled from the university for academic dishonesty

Name.....Signature.....

Reg. No.....Date.....

SUPERVISOR(S) DECLARATION

I/We hereby approve the examination of this thesis. The thesis has been subjected to plagiarism test and its similarity index is not above 20%.

1. Name.....Signature.....Date.....

2. Name.....Signature.....Date.....

DECLARATION

This thesis is my own original work prepared with no other than the indicated sources and support and has not been presented elsewhere for a degree or any other award.

John Gichuki Ndia

DATE

SIT/LH/004/2015

Certification

The undersigned certify that they have read and hereby recommend for acceptance of Masinde Muliro University of Science and Technology a thesis entitled “Structural Complexity Framework and Metrics for Analyzing the Maintainability of Sassy Cascading Style Sheets.”

Prof. Geoffrey Muchiri Muketha

DATE

Department of Computer Science

School of Computing and Information Technology

Murang’a University of Technology

Dr. Kelvin Kabeti Omieno

DATE

Department of Information Technology and Informatics

School of Computing and Information Technology

Kaimosi Friends University College (A Constituent College of Masinde Muliro

University of Science and Technology).

DEDICATION

This work is dedicated to my lovely wife Velma Auma Gichuki and my parents Danson Ndia and Esther Ndia.

ACKNOWLEDGMENTS

I take this first opportunity to thank God for his enabling grace and renewal of strength throughout this long and demanding PhD journey. The words in Psalms 116:12 remains a great challenge to me “What shall I render unto the Lord for all His benefits toward me?”

My special gratitude goes to my supervisors Prof. Geoffrey Muchiri Muketha and Dr. Kelvin Kabeti Omieno for their tireless effort in guiding me and exposing me to research writing. I also wish to thank the faculty at the the School of Computing and Informatics, Masinde Muliro University of Science and Technology for the valuable research insights they provided to me in the course of my study.

I also thank my dear wife for her moral support and her prayers and encouragement which made me gain new energy in every stage of this Ph.D. study. Special thanks also go to my loving and supportive parents who always wanted to know how far I have proceeded with my study.

I honor my spiritual authority Rev. Daniel Muraya for his spiritual support and understanding in times when I couldn't attend to church duties during the time of writing this thesis. I can't forget Deacon Danson Mutheka who could oftenly offer words of encouragement. I also thank the entire CORMs chapel family for their moral and spiritual support.

ABSTRACT

One of the most popular languages in the web domain is Cascading Style Sheet (CSS). The language has evolved over time with the latest development being the introduction of CSS preprocessors which has made it possible to write CSS codes in a faster and efficient way. Therefore, the migration from CSS to CSS preprocessors by the front-end developers has been tremendous. There are several CSS preprocessors available in the industry with the Syntactically Awesome Style Sheets (SASS) becoming one of the most preferred preprocessors. This elevation of SASS is as a result of influence by its new syntax SCSS (Sassy Cascading Style Sheets) which is closer to CSS syntax. Although SCSS is very promising, it has inherent complexity which keeps increasing with time as a result of maintenance practices. The Entity-Attribute-Metric (EAM) model was used to describe the process followed to identify SCSS metrics while the Boehm model was used to identify the maintainability sub-characteristics. In addition, the Muketha's structural attributes classification framework was extended so as to develop the SCSS structural attributes classification framework. The measurement of software complexity via software metrics for different software's and software paradigms has continued to gain grounds over the years. There exists several structural CSS metrics but they cannot be directly applied to SCSS because SCSS has richer features than CSS. In addition, there is no existing framework that can be used to guide the definition of SCSS structural complexity metrics. To close the gaps identified, the researcher developed an SCSS complexity attributes classification framework which was validated through an expert opinion survey. This study proposed a suite of SCSS structural complexity metrics which were theoretically validated via Weyuker's properties and Kaner framework. In addition, a tool was developed to automate the collection and computation of metric values. The data collected was analyzed through descriptive statistics (frequencies, mean and standard deviation) and inferential statistics (Spearman's rho, ANOVA tests, and principle component analysis). Empirical studies by way of experimentation were conducted and all the proposed metrics strongly correlated with the three aspects of maintainability, namely, understandability, modifiability, and testability. Additionally, the metrics were found to be important for the measurement of SCSS complexity. The findings of this study show that all the proposed metrics can serve as maintainability predictors for SCSS.

TABLE OF CONTENTS

PLAGIARISM STATEMENT.....	ii
DECLARATION.....	iii
DEDICATION.....	iv
ACKNOWLEDGMENTS	v
ABSTRACT	vi
LIST OF FIGURES	xvi
LIST OF TABLES	xviii
ACRONYMS AND ABBREVIATIONS.....	xx
DEFINITION OF OPERATIONAL TERMS.....	xxi
CHAPTER ONE: INTRODUCTION	1
1.1 Overview.....	1
1.2 Background to the Study.....	1
1.3 Statement of the Problem.....	3
1.4 Objectives.....	5
1.4.1 General Objective	5
1.4.2 Specific Objectives	5
1.5 Research Questions	5
1.6 Research Hypotheses	6
1.7 Significance of the Study	7
1.8 Scope of the Study	8
1.9 Limitations of the Study.....	9
1.10 Contributions of the Thesis.....	9
1.11 Thesis Organization	9

CHAPTER TWO: LITERATURE REVIEW	12
2.1 Introduction	12
2.2 Background Information on Cascading Style Sheets (CSS)	12
2.3 SASS Pre-Processor	14
2.4 Software Complexity	19
2.4.1 Software Complexity Measurement.....	19
2.4.2 Software Complexity Attributes Classification	20
2.4.3 Structural Complexity	25
2.4.3.1 Structural Complexity Properties for Traditional Software	25
2.4.3.2 Structural Properties for SCSS.....	28
2.5 Existing Software Complexity Metrics.....	31
2.5.1 Complexity Metrics for Traditional Software.....	32
2.5.1.1 Lines of Code (LOC)	32
2.5.1.2 Function Point (FP).....	33
2.5.1.3 Halstead’s Metrics.....	34
2.5.1.4 McCabe Cyclomatic Complexity	37
2.5.2 Complexity Metrics for Object-Oriented Languages.....	38
2.5.2.1 Chidamber and Kemerer Metrics	38
2.5.2.2 Mishra Inheritance Metrics	39
2.5.2.3 Abreu and Carapuca Metrics.....	39
2.5.2.4 Lorenz and Kidd Metrics Suite	40
2.5.2.5 Li Metrics	40
2.5.2.6 MOOD Metrics Suite	41
2.5.2.7 Misra, Adewumi, Fernandez-Sanz and Damasevicius Metrics	42
2.5.3 Web-Based Metrics	42

2.5.3.1 Misra and Cafer Metrics.....	43
2.5.3.2 Basci and Misra Metrics.....	43
2.5.3.3 Thaw and Misra Metrics	45
2.5.3.4 Tamayo, Granell and Huerta Metrics.....	45
2.5.4 Adewumi, Misra and Ikhu-Omoregbe Metrics	46
2.6 Metrics Validation.....	49
2.6.1 Theoretical validation.....	49
2.6.1.1 Weyuker’s Properties	49
2.6.1.2 Briand’s Property-based Framework	51
2.6.1.3 Kaner’s Framework.....	54
2.6.2 Empirical Validation	54
2.6.2.1 Experiments.....	54
2.6.2.2 Case Studies	55
2.6.2.3 Surveys.....	55
2.7 Metrics Tools	56
2.8 Software Maintainability.....	59
2.9 Gaps Identified in Literature	63
2.10 Theoretical Framework	64
2.11 Conceptual Framework	65
2.12 Chapter Summary.....	66
CHAPTER THREE: RESEARCH METHODOLOGY	68
3.1 Introduction	68
3.2 Research Philosophy	68
3.3 Research Design.....	68
3.3.1 Research Process	69

3.3.1.1	Development of an Attribute Classification Framework	70
3.3.1.2	Definition of SCSS metrics	73
3.3.1.3	Theoretical Validation of SCSS metrics	73
3.3.1.4	Development of a Metrics Tool for SCSS	73
3.3.2	Research Strategy	74
3.4	Population	75
3.5	Sampling Strategy and Sample Size	76
3.6	Pilot Study	77
3.7	Data Collection Instruments.....	78
3.8	Validity and Reliability	79
3.8.1	Validity of the Research Instruments	79
3.8.2	Reliability of the Research Instruments	79
3.9	Experimental Materials	80
3.10	Data Analysis	81
3.10.1	Data Analysis Methods for the Expert Opinion Survey.....	81
3.10.2	Data Analysis Methods for Tool Validation	81
3.10.3	Data Analysis Methods for the Controlled Laboratory Experiment	81
3.11	Ethical Issues.....	83
3.12	Chapter Summary.....	84
 CHAPTER FOUR: DEVELOPMENT OF STRUCTURAL COMPLEXITY		
ATTRIBUTE CLASSIFICATION FRAMEWORK FOR SASSY CASCADING		
STYLE SHEETS (SCACF-SCSS)		
		85
4.1	Introduction	85
4.2	Requirements of the SCACF-SCSS Framework	85
4.3	Architecture of the Proposed Framework	85

4.3.1 Intra-Module Attribute	86
4.3.2 Inter-Module Attribute	88
4.3.3 Hybrid Attribute	90
4.3.4 Extra-Module Attribute	91
4.4 Application of the Framework	93
4.4.1 Intra-Module Attribute	93
4.4.2 Inter-Module Attribute	95
4.4.3 Hybrid Attribute	96
4.4.4 Extra-Module Attribute	97
4.5 Expert Opinion Validation Survey	98
4.5.1 Goal of the Study	99
4.5.2 Context Definition	99
4.5.3 Survey Operation	99
4.5.4 Reliability of the Research Instrument	99
4.5.5 Results	100
4.5.5.1 Respondents Demographics	100
4.5.5.2 Level of Education for Respondents	100
4.5.5.3 Years of Industrial Experience	101
4.5.5.4 Level of Knowledge in Software Engineering Processes	101
4.5.5.5 Level of Knowledge for SCSS	102
4.5.5.6 Relevance of the Framework	103
4.5.5.7 Comprehensiveness of the Framework	104
4.6 Chapter Summary	105
CHAPTER FIVE: STRUCTURAL COMPLEXITY METRICS FOR SASSY CASCADING STYLE SHEETS	107

5.1 Introduction	107
5.2 Determination of Attributes to be Measured.....	107
5.3 Metrics Definition	108
5.3.1 Average Block Cognitive Complexity for SCSS (ABCC _{scss})	110
5.3.2 Nesting Factor for SCSS (NF _{scss}).....	114
5.3.3 Selector Use Inheritance Level (SUIL).....	118
5.3.4 Coupling Level for SCSS (CL _{scss}) metric	120
5.4 Theoretical Validation Results for the Proposed Metrics	122
5.4.1 Validation with Weyuker’s Properties.....	122
5.4.2 Validation with Kaner’s Framework.....	126
5.5 Chapter Summary.....	128
 CHAPTER SIX: IMPLEMENTATION OF A STRUCTURAL	
COMPLEXITY METRICS TOOL FOR SASSY CASCADING STYLE	
SHEETS (SCMT-SCSS)	130
6.1 Introduction	130
6.2 Requirements of the SCMT-SCSS.....	130
6.3 Metrics Implementation	131
6.4 Input File Format	131
6.5 SCMT-SCSS Tool Architectural Design	133
6.5.1 Input Component.....	133
6.5.2 Analyzer Component	133
6.5.3 Output Component.....	133
6.6 User Interface Design.....	135
6.7 Algorithm Design.....	137
6.7.1 ABCC _{scss} Algorithm	138

6.7.2 NF _{SCSS} Algorithm	140
6.7.3 SUIL Algorithm	141
6.7.4 CL _{SCSS} Algorithm	142
6.8 Execution of the SCMT-SCSS Tool	144
6.9 Experimental Validation of the SCMT-SCSS Tool	147
6.9.1 Goal of the Study	147
6.9.2 Context Definition.....	147
6.9.3 Threats to Validity.....	148
6.9.3.1 Internal Validity	148
6.9.3.2 External Validity	148
6.9.4 Experimental Design.....	148
6.10 Results	149
6.10.1 Time to Complete Tasks	149
6.10.2 Suitability, Accuracy and Operability Rating	152
6.11 Chapter Summary.....	153
 CHAPTER SEVEN: AN EXPERIMENTAL VALIDATION OF	
STRUCTURAL COMPLEXITY METRICS FOR SASSY CASCADING	
STYLE SHEETS	155
7.1 Introduction	155
7.2 Context Definition.....	155
7.3 Strategy for Conducting the Experiments.....	156
7.4 Pilot Study.....	156
7.5 Subjects' Background	158
7.6 Subjective Data	160
7.6.1 Experimental Planning	160

7.6.1.1	Effect of SCSS Metrics on Subjects Rating of Understandability	160
7.6.1.2	Effect of SCSS Metrics on Subjects Rating of Modifiability	161
7.6.1.3	Effect of SCSS Metrics on Subjects Rating of Testability	161
7.6.2	Threats to validity	162
7.6.2.1	Internal validity	162
7.6.2.2	External Validity	163
7.7	Objective Data.....	163
7.7.1	Experimental Planning	164
7.7.1.1	Effect of SCSS Metrics on Subjects Understanding time.....	164
7.7.1.2	Effect of SCSS Metrics on Subjects Modifying Time.....	165
7.7.1.3	Effect of SCSS Metrics on Subjects Testing Time	166
7.7.2	Threats to Validity.....	166
7.7.2.1	Internal Validity	166
7.7.2.2	External Validity	167
7.8	Results.....	167
7.8.1	Subjective Results	167
7.8.1.1	Relationship between Metrics and Understandability	168
7.8.1.2	Relationship between Metrics and Modifiability	169
7.8.1.3	Relationship between Metrics and Testability	170
7.8.2	Objective Results	172
7.8.2.1	Relationship between Metrics and Time to Understand	172
7.8.2.2	Relationship between Metrics and Time to Modify.....	175
7.8.2.3	Relationship between Metrics and Time to Test.....	178
7.9	Discussion	181
7.9.1	Implications of Understandability Results	182

7.9.1.1 Relationship between Metrics and Understandability	182
7.9.1.2 Relationship between Metrics and Time to Understand	183
7.9.2 Implications of Modifiability Results	184
7.9.2.1 Relationship between Metrics and Modifiability	184
7.9.2.2 Relationship between Metrics and Time to Modify.....	184
7.9.3 Implications of Testability Results	185
7.9.3.1 Relationship between Metrics and Testability	185
7.9.3.2 Relationship between Metrics and Time to Test.....	185
7.10 Effect of Moderating Variables on the Complexity-Maintainability Relationship	186
7.11 Chapter Summary.....	186
CHAPTER EIGHT: SUMMARY, CONCLUSION AND RECOMMENDATIONS.....	188
8.1 Summary	188
8.2 Conclusion	188
8.3 Recommendations for Future Work.....	191
8.3.1 Define Metrics for other CSS Pre-Processors.....	192
8.3.2 Extending Structural Complexity Framework	192
8.3.3 Metrics Tool Extension to Recognize Multiple Languages.....	192
8.3.4 Further Metrics Experimentation	193
REFERENCES.....	194
APPENDICES	207

LIST OF FIGURES

Figure 2.1: SASS Pre-processor Syntaxes	15
Figure 2.2: An Alert Rule Block	17
Figure 2.3: SCSS Code with Multiple Blocks	18
Figure 2.4: Software Metrics Definition Process	20
Figure 2. 5: Software Complexity Classification.....	23
Figure 2.6: Extended Structural Complexity Classification	24
Figure 2.7: Coupling in OOP	27
Figure 2.8: Coupling in SCSS.....	27
Figure 2.9: Selector Inheritance	30
Figure 2.10: Nesting of Rules	30
Figure 2.11: McCall Maintainability Sub-characteristics	60
Figure 2.12: Boehm’s Maintainability Sub-characteristics.....	60
Figure 2.13: ISO-9126 Maintainability Sub-characteristics	61
Figure 2.14: ISO-25010 Maintainability Sub-characteristics	62
Figure 2.15: Conceptual Framework.....	66
Figure 3.1: Research Process	70
Figure 4.1: SCSS Size.....	87
Figure 4.2: Control-flows in SCSS	88
Figure 4.3: Inheritance in SCSS.....	89
Figure 4.4: Nesting in SCSS	89
Figure 4.5: Association in SCSS.....	90
Figure 4.6: Information Flow in SCSS	91
Figure 4.7: Structural Complexity Attribute Classification Framework for SCSS (SCACF-SCSS).....	92

Figure 4.8: Size Complexity Scenario	94
Figure 4.9: Control Flow Complexity Scenario.....	94
Figure 4.10: Inheritance Complexity Scenario	95
Figure 4.11 Nesting complexity Scenario.....	96
Figure 4.12: Association complexity Scenario	97
Figure 4.13: Information Flow Complexity Scenario.....	98
Figure 5.1: EAMT Model	109
Figure 5.2: ABCC _{SCSS} Metric Example	113
Figure 5.3: Nesting Depth.....	115
Figure 5.4: Nesting Breadth	116
Figure 5.5: NF _{SCSS} Metric Example	118
Figure 5.6: SUIL Metric Example	119
Figure 5.7: CL Metric Example	121
Figure 6.1: The Structure of an SCSS file	132
Figure 6.2: SCMT-SCSS Tool Architecture	134
Figure 6.3: SCMT-SCSS Structure Chart Diagram	135
Figure 6.4: SCMT-SCSS Use Case Diagram.....	136
Figure 6.5: Form Layout Design.....	137
Figure 6.6: SCSS Base Metrics Values.....	145
Figure 6.7: SCSS Derived Metrics Values.....	146
Figure 6.8: SCSS Metrics Values in a Text File	147

LIST OF TABLES

Table 2.1: Comparison between Traditional and SCSS Software	16
Table 2.2: Identified Gaps.....	63
Table 3.1 Metrics Validation Reliability Statistics	80
Table 4.1. Framework Reliability Statistics.....	100
Table 4.2: Level of Education for Respondents.....	101
Table 4.3. Years of Industrial Experience.....	101
Table 4.4: Level of Knowledge for Software Engineering Processes	102
Table 4.5: Level of Knowledge for SCSS.....	103
Table 4.6: Relevance of the Framework	104
Table 4.7: Comprehensiveness of the Framework.....	105
Table 4.8: Adequacy of SCSS Complexity Features	105
Table 5.1: Weights for Basic Control Structures	112
Table 5.2: Validation Results of SCSS metrics with Weyuker’s Axioms	126
Table 6.1: Time to Complete Tasks for SCSS File 1	150
Table 6.2: Time to Complete Tasks for SCSS File 2.....	150
Table 6.3: Time to Complete Tasks for SCSS File 3.....	151
Table 6.4: Time to Complete Tasks for SCSS File 4.....	151
Table 6.5: Average Rating on Suitability.....	152
Table 6.6: Average Rating on Accuracy	153
Table 6.7: Average Rating on Operability	153
Table 7.1: Programming languages taken.....	158
Table 7.2: Software Engineering courses pursued.....	159
Table 7.3: Knowledge of SCSS	159
Table 7.4: Subjects Background Knowledge.....	163

Table 7.5: KMO and Bartlett's Test	164
Table 7.6: Correlation with Understandability.....	168
Table 7.7: Understandability Significance with ANOVA	169
Table 7.8: Correlation with Modifiability.....	169
Table 7.9: Modifiability Significance with ANOVA.....	170
Table 7.10: Correlation with Testability	171
Table 7.11: Testability Significance with ANOVA.....	171
Table 7.12: Correlation Results with Time to Understand	173
Table 7.13: Understanding time significance with ANOVA.....	173
Table 7.14: PCA for Understandability	174
Table 7.15: PCA Loadings for Understandability.....	175
Table 7.16: Correlation Results with Time to Modify.....	176
Table 7.17: Modifying Time Significance with ANOVA	176
Table 7.18: PCA for Modifiability.....	177
Table 7.19: PCA Loadings for Modifiability.....	178
Table 7.20: Correlation Results with Time to Test.....	179
Table 7.21: Testing Time Significance with ANOVA.....	179
Table 7.22: PCA for Testability.....	180
Table 7.23: PCA Loadings for Testability	181

ACRONYMS AND ABBREVIATIONS

ABCC_{scss}	Average Block Cognitive Complexity for SCSS
BSC	Balanced Scorecard
CL_{scss}	Coupling Level for SCSS
CSS	Cascading Style Sheet
DTD	Document Type Definition
EAM	Entity Attribute Metric Model
EAMT	Entity Attribute Metric Tool Model
FP	Function point
GQM	Goal Question Metric
HTML	Hyper Text Markup Language
IEEE	Institute of Electrical and Electronics Engineers
LOC	Lines of code
NF_{scss}	Nesting Factor for SCSS
PCA	Principle Component Analysis
SASS	Syntactically Awesome Stylesheets
SCACF	Structural Complexity Attributes Classification Framework
SCMT	Structural Complexity Metrics Tool
SCSS	Sassy Cascading Style Sheets
SPSS	Statistical Package for Social Sciences
SUIL	Selector Use Inheritance Level
XHTML	Extensible Hyper Text Markup Language
XML	Extensible Markup Language

DEFINITION OF OPERATIONAL TERMS

Empirical Validation: This is the involvement of experiments and surveys to gain knowledge on a particular subject.

Maintainability: The ease with which software codes can be understood, modified, and tested.

Modifiability: This is how easy it is to incorporate changes to an SCSS code.

Regular CSS: This refers to Cascading Style Sheets (CSS).

Software Attribute: This is the structural feature or property of a software artifact.

Software Complexity: This refers to how difficult it is to understand, modify and test a program.

Software Metric: This is a quantitative measure of a degree to which a software artifact possesses some property.

Testability: This is how easy it is to identify errors or faults in an SCSS code.

Theoretical Validation: This is a formal and practical approach for proving the soundness of metrics.

Understandability: This is how easy it is to comprehend an SCSS code.

CHAPTER ONE

INTRODUCTION

1.1 Overview

This chapter gives fundamental information concerning the complexity of Sassy Cascading Style Sheets code (SCSS) and how it contributes to the difficulty in maintaining the code and the need to measure and control software complexity. The objectives to achieve for this study, research questions which are directly mapped with objectives, significance of the study, scope of the study, limitations, and contributions of this study are also presented in this chapter.

1.2 Background to the Study

Web-based applications are developed for personal use, and for private and public institutions. These applications use different languages and one of the integral parts in their development is Cascading Style Sheets (CSS) language (Adewumi, Misra & Ikhu- Omoregbe, 2012). CSS is the standard language for styling structured documents, such as HTML and XHTML. HTML (Hyper Text Markup Language) is used to create content while CSS is concerned with the presentation of the web documents written in HTML, XHTML (Extensible HTML) and it can also be applied in any XML (Extensible Markup Language) document to bring about aesthetically pleasing and user-friendly interfaces. Basically, the motivation for the use of CSS is to separate content from presentation (Adewumi et al., 2012).

Web systems have over the years evolved from simple hypertext markup language (HTML)-based applications to complex cascading style sheets (CSS)-based applications (Adewumi et al., 2012). Further developments have seen the

incorporation of traditional programming language concepts into the regular CSS language resulting in CSS preprocessors. The invention of CSS preprocessors has made the writing of CSS codes faster, efficient, and more maintainable. Therefore, CSS preprocessors have gained popularity with front-end developers and 54 % of them use it in their development tasks in the recent past. A CSS preprocessor is a program that converts the written codes into CSS codes which can be rendered by the web browser. There exist several CSS Preprocessors such as Sass, Less, Stylus, CSS-crush, Myth, and Rework. (Mazinanian & Tsantalis, 2016).

SASS (Syntactically Awesome Style sheets) preprocessor is one of the most popular CSS Preprocessor and governments such as the United States Federal Government advises its front-end developers to use SASS preprocessor to develop style sheets and this has made it very popular in the industry globally (Mazinanian & Tsantalis, 2016). Sass preprocessor has two syntaxes, .sass, which is the older syntax and .scss which is the new and improved standard (Cederholm, 2013). The new syntax SCSS (Sassy Cascading Stylesheets) is closer to CSS syntax, and it introduces the concepts of SASS preprocessor thus making it popular between the two SASS syntaxes (Cederholm, 2013; Catlin & Catlin, 2011). Therefore, the focus of this study was on Sassy cascading style sheets.

SASS preprocessors add extra functionality such as the use of variables, nesting rules, mixins functions, operators, control directives (@for, @if, @else, @each, @while and if()) and selector inheritance. The introduction of these new features makes the language have inherent complexity in comparison to regular CSS which has a simple syntax. This kind of complexity increases with time each and every time new rule

blocks are added to the existing stylesheet code (Mazinanian & Tsantalís, 2016). Software complexity refers to how code is difficult to understand, modify and test, thus its high levels lead to software that is unreliable and difficult to maintain (Ogheneovo, 2014; Mesbah & Mirshokraie, 2012; Shao & Wang, 2003). This raises the need to investigate the complexity of SCSS files which can lead to codes that have errors, are difficult to understand, modify and test.

This risk of having complex SCSS code implies that there is a need to control its level of complexity. Software metrics are central in measurement and control of software complexity (Misra, Adewumi, Fernandez-Sanz, & Damasevicius, 2018; Muketha, Ghani, Selamat & Atan, 2010a). This is achieved by the metrics providing feedback to the software designers concerning complexity thus influencing the decisions made. When there is a lack of this feedback, decisions are made in an ad-hoc manner (Misra & Cafer, 2012).

There are efforts made to define software metrics in the web domain such as CSS complexity metrics (Adewumi et al., 2012), software metrics for XML schema (Basci & Misra, 2011a), web services (Basci & Misra, 2009; Basci & Misra, 2011b) and Document Type Definition (DTDs) (Basci & Misra, 2008). This implies that metrics should be defined to achieve desirable complexity levels for SCSS.

1.3 Statement of the Problem

Front-end web developers are increasingly adopting the use of SCSS because it allows ease of development and maintainability of Web applications (Mazinanian & Tsantalís, 2016). However, SCSS codes have inherent complexity that increase due to

maintenance activities. In addition, a substantial number of web developers and several of them still use regular CSS, because they perceive that it still has simpler syntax than SCSS (Mazinanian & Tsantalis, 2016; Lie & Bos, 2005).

Researchers agree that high levels of software complexity lead to software that is difficult to maintain (Misra, 2018; Ogheneovo, 2014; Adewumi et al., 2012; Ghosheh, Black, & Qaddour, 2008). Therefore, a comprehensive and relevant set of measurable attributes should be identified, then metrics which are based on the attributes should be defined to measure complexity with the aim of controlling it. Although there are several structural CSS metrics proposed in the literature, they cannot be directly applied to SCSS because they do not capture the unique structural properties of SCSS. Furthermore, although the development of metrics tool has been recognized by various studies as a necessary step for making the metrics acceptable in the software industry, most of the reviewed CSS metrics either lack tool support or the tools are not efficient (Adewumi, Emebo, Misra & Fernandez, 2015; Basci & Misra, 2011; Misra & Cafer, 2012; Thaw & Misra, 2013; Misra et al., 2018).

Another aspect of the problem is that there is no existing comprehensive framework that can be used as a guide to define SCSS metrics. The existing frameworks consider attributes in procedural languages, object oriented programming (OOP) domain, business process models but not SCSS language. This lack of a comprehensive framework means that definition of SCSS metrics can only be defined in an adhoc manner, which is not good for the software industry.

1.4 Objectives

The following section stipulates the general and specific objectives achieved by this study.

1.4.1 General Objective

The main objective of this study was to define relevant and comprehensive SCSS measurable attributes and to determine a valid suite of structural complexity metrics that can be used as maintainability predictors of SCSS code.

1.4.2 Specific Objectives

The specific objectives of this study are:

- i. To determine a set of SCSS attributes that affect its structural complexity.
- ii. To define structural complexity metrics for SCSS code.
- iii. To develop a functional and usable metrics analysis tool for SCSS metrics computation.
- iv. To validate the structural complexity metrics for SCSS using a controlled laboratory experiment.

1.5 Research Questions

- i. Which attributes can determine the structural complexity of SCSS code?
- ii. Which metrics can evaluate the structural complexity of SCSS code?
- iii. How can you automate calculation of SCSS metrics?
- iv. Which metrics are effective in predicting the maintainability of SCSS code?

1.6 Research Hypotheses

A set of six pairs of hypotheses were formulated to answer research question four. Each pair represents the null and alternative hypotheses of each of the dependent variables to be tested, namely, understandability, modifiability, testability, understanding time, modifying time, and testing time.

a. Understandability hypotheses

- i. Null Hypothesis ($H_{0.u}$): There exists no significant correlation between the SCSS metrics and subjects rating of understandability of SCSS files.
- ii. Alternative Hypothesis ($H_{1.u}$): There exists significant correlation between the SCSS metrics and subjects rating of understandability of SCSS files.

b. Modifiability hypotheses

- i. Null Hypothesis ($H_{0.m}$): There exists no significant correlation between the SCSS metrics and subjects rating of modifiability of SCSS files.
- ii. Alternative Hypothesis ($H_{1.m}$): There exists significant correlation between the SCSS metrics and subjects rating of modifiability of SCSS files.

c. Testability hypotheses

- i. Null Hypothesis ($H_{0.t}$): There exists no significant correlation between the SCSS metrics and subjects rating of testability of SCSS files.
- ii. Alternative Hypothesis ($H_{1.t}$): There exists significant correlation between the SCSS metrics and subjects rating of testability of SCSS files.

d. Understanding time hypotheses

- i. Null Hypothesis ($H_{0.ut}$): There exists no significant correlation between the SCSS metrics and understanding time of SCSS files.
- ii. Alternative Hypothesis ($H_{1.ut}$): There exists significant correlation between the SCSS metrics and understanding time of SCSS files.

e. Modifying time hypotheses

- i. Null Hypothesis ($H_{0.mt}$): There exists no significant correlation between the SCSS metrics and modifying time of SCSS files.
- ii. Alternative Hypothesis ($H_{1.mt}$): There exists significant correlation between the SCSS metrics and modifying time of SCSS files.

f. Testing time hypotheses

- i. Null Hypothesis ($H_{0.tt}$): There exists no significant correlation between the SCSS metrics and testing time of SCSS files.
- ii. Alternative Hypothesis ($H_{1.tt}$): There exists significant correlation between the SCSS metrics and testing time of SCSS files.

1.7 Significance of the Study

This study aimed at proposing complexity metrics for SCSS. To achieve this, the study began by developing an SCSS attribute classification framework which assisted in identifying all the possible factors that would contribute to the complexity of SCSS. Complexity metrics were defined, theoretically and empirically validated, and the findings indicated that the metrics are useful for predicting SCSS Maintainability. Therefore, researchers in software metrics can use the developed framework to aid in identifying structural complexity attributes or extend it depending on the uniqueness of various software artifacts. The SCSS designers and programmers can use the

metrics to predict SCSS maintainability. A metrics tool was developed for the purpose of automating the collection and computation of SCSS metrics, this means that the users of the tool will quickly receive response which are accurate and make conclusions based on the results acquired. For instance, metrics values can assist SCSS programmers in making certain decisions such as restructuring of SCSS code with high coupling level.

This study provided the background on which other researchers can refer to define metrics in stylesheets field especially with CSS preprocessors.

1.8 Scope of the Study

This study focused on SCSS syntax (.scss) of SASS preprocessor, it didn't look at the alternative SASS syntax known as SASS syntax (.sass). Therefore, all the SASS files not conforming to .scss syntax were not considered in this study.

The proposed metrics were static metrics and so dynamic aspects were not considered in this study. In addition, the metrics focused on structural aspects of SCSS code, meaning other forms of software complexity were not considered.

Finally, this study focused on maintainability aspect of software quality, and other software quality factors such as portability, reliability, usability, efficiency and functionality did not form part of this study.

1.9 Limitations of the Study

A limitation of the study refers to the aspects that the researcher knows may affect the validity of the study conclusion and results generalizability, however, the researcher has no control over (Kumar, 2011; Mugenda & Mugenda, 2008). The identification of SCSS industry experts was a challenge. To overcome this challenge, snowballing technique was used.

1.10 Contributions of the Thesis

This thesis made the following contributions:

- i. An SCSS structural complexity attributes framework was developed and validated through an expert's opinion survey.
- ii. A set of four SCSS metrics were defined for measuring the structural complexity of SCSS code.
- iii. A metrics tool called Structural Complexity Metrics Tool (SCMT) for SCSS was developed for metrics computation. This tool was validated through experiments and proved to be functional and usable
- iv. The proposed SCSS metrics were proved to be theoretically sound via Weyuker's properties and Kaner's framework
- v. The empirical validation of the proposed SCSS metrics proved that they can predict the maintainability of SCSS code.

1.11 Thesis Organization

This thesis is divided into eight chapters as described below:

The first chapter presents an introduction of the thesis. It includes a detailed description of the background to the study, research problem, objectives, the significance of the study, scope of research, limitations, and contributions of this study.

The second chapter presents a review of related literature. It includes literature on structural properties of SCSS, software complexity attributes classification, existing software complexity metrics, metrics validation methods, metrics tools, and software maintainability. The gaps in literature were also identified.

The third chapter is the research methodology and it describes the research philosophy, research design, a summary of the research process, research strategy, sampling strategy, research instruments, validation and reliability of research instruments, how to analyze data, and ethical considerations of this research.

The fourth chapter presents the structural complexity attributes complexity framework for SCSS, and it covers the architecture of the proposed framework which at high level has four categories, intra-module attribute, inter-module attribute, hybrid attribute, and extra-module attribute. The chapter describes how the framework can be applied in a real-life scenario and finally the framework descriptive validation results are presented in terms of SCSS experts background knowledge, relevance, and comprehensiveness of the framework.

The fifth chapter is the structural complexity metrics for SCSS. It defines the metrics, Average Block Cognitive Complexity, Nesting Factor, Selector Use Inheritance Level,

and Coupling Level. The metrics were validated theoretically via Weyuker's properties and Kaner framework. Finally, each metric was demonstrated through a real-life scenario to prove that they are intuitional.

The sixth chapter presents the structural complexity metrics tool for SCSS. It describes the requirements for tool development, tool architecture and design such as user interface design, form layout design, and algorithm design. The chapter also presents the metrics tool validation results.

The seventh chapter is the experimental validation of structural complexity metrics for SCSS. The experiment which consists of subjective and objective phases is described. This chapter presents the validation results in terms of correlation, ANOVA and principle component analysis (PCA).

The eighth and final chapter presents summary, conclusion and future work of the study. This includes a general conclusion of the study findings as well as future research directions.

CHAPTER TWO

LITERATURE REVIEW

2.1 Introduction

This chapter presents a detailed analysis of literature in the area of CSS, SCSS, software attributes or indicators, software complexity metrics, metrics tools, validation of metrics and software maintainability models. Research gaps are also identified thus forming a theoretical basis upon which this research work was established.

2.2 Background Information on Cascading Style Sheets (CSS)

The CSS language is composed of a sequence of style rules, where each rule has a selector that selects the elements needed to style in the HTML or XML document (Hissom, 2011). This language is used by the web developers to define the look and feel of structured documents such as HTML and XML. The web developers have over the years increasingly used CSS in their everyday development tasks of web-based software (Mazinanian & Tsantalis, 2016).

CSS has evolved over the years from CSS1 to CSS3. More recent extensions of CSS have also been proposed such as CSS preprocessors. CSS1 was a simple version with about 50 properties and is mostly used for screen-based presentations. CSS2 includes all CSS1 properties plus an additional around 70 properties of its own. The additional properties have for instance enabled CSS2 to describe aural presentations and page breaks that couldn't be done earlier. An enhanced CSS 2.1 was also released that added more features such as the ability to describe the parts that are supported by two or more browsers (Lie & Bos, 2005). Finally, CSS3 is split into modules such as selectors, box model, backgrounds and borders, text effects, image values and replaced

content, 2D/3D transformations, animations, multiple column layouts, and user interface (Hissom, 2011). The purpose of modularization is to have multiple specifications, where each specification has its own progression path (Hissom, 2011).

Post CSS3, developments have taken the direction of CSS preprocessors. Several CSS preprocessors have so far been proposed such as Syntactically Awesome Style Sheets (SASS), Less, Stylus, CSS-Crush, Myth and Rework with each of them having unique syntaxes (Mazinanian & Tsantalis, 2016; Charpentier et al., 2016). CSS preprocessors add extra features to those found in regular CSS such as the use of variables, nesting of rules, use of mixins, use of function calls, inheritance, use of control flow statements and use of operators (Mazinanian & Tsantalis, 2016).

Variables are defined to store one or more style values and represent data, such as numeric values and characters (Mazinanian & Tsantalis, 2016). A variable enables reuse of the style values stored in the stylesheets. In stylesheets, variables can be used to set up colors and fonts (Henley, 2015). In some instances, variable values are manipulated using arithmetic operators and by passing them to preprocessor built-in function (Mazinanian & Tsantalis, 2016).

Rule nesting is like class nesting in object-oriented programming. According to Mazinanian and Tsantalis (2016), CSS Preprocessors permit a rule to be placed inside another rule as a way of combining multiple CSS rules within one another.

Some other powerful features introduced by CSS preprocessors include mixins and user-defined functions. While mixins store multiple values, functions are invoked to

allow the use of parameters (Mazinanian & Tsantalis, 2016; Henley, 2015). Mixins and functions are beneficial in that they help to avoid writing repetitive codes (Mazinanian & Tsantalis, 2016).

Other new features include inheritance, control directives, and operators. In the SASS preprocessor, for instance, inheritance uses the `@extend` directive to share or extend the behavior of an existing selector (Mazinanian & Tsantalis, 2016). Control directives are the equivalent of the control-flow statement in object-oriented programming. These directives include `@if`, `@for`, `@each`, and `@while` statements (Henley, 2015) and are used for applying a style many times with variations (Catlin & Catlin, 2011). Finally, CSS preprocessors have introduced operators which include addition, subtraction, division, multiplication, relational operators and equality operators (Henley, 2015).

2.3 SASS Pre-Processor

Though there are many available CSS pre-processors for use in the software industry, the SASS pre-processor is one of the most popular (Mazinanian & Tsantalis, 2016). SASS pre-processor supports two syntaxes, Sassy CSS (SCSS) which uses the `.scss` extension and indented syntax which uses the `.sass` extension. SCSS is the newer of the two syntaxes and the most popular among front web developers because it is a superset of CSS making migration to SCSS a lot easier, it is easy to use the existing stylesheets and incorporate SASS features, and it is also more expressive meaning its more logically grouped, for example, one can compress several lines of codes in SASS into just fewer lines in SCSS (Cederholm, 2013). Fig. 1 shows a family tree of SASS pre-processor.

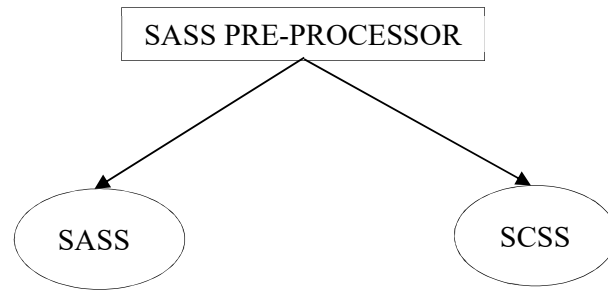


Figure 2.1: SASS Pre-processor Syntaxes

The basic building component of an SCSS is a rule block. A rule block is made up of a selector and one or more attributes (Adewumi et al., 2012). The selector points to the HTML element to be styled while attributes specify the style on the element. An attribute is also known as property name and can have one or more values. SCSS has other blocks such as mixin blocks (comprising of a `@mixin` directive with opening and closing braces), function blocks (comprising of `@function` directive with opening and closing braces), control directives block (comprising of control directive i.e. `@if`, `@each`, `@for`, `@elseif` with opening and closing braces), and media blocks (it comprises of `@media` with opening and closing braces). An SCSS block is defined as any block that consists of a selector or `@rule` directive, opening brace, set of attributes and/or directives and a closing brace.

Sassy CSS is a style sheet language whose aim is to determine how the web pages are presented. In contrast, the aim of conventional programming languages such as Java, C++, etc. is to automate processes. Basically, SCSS is used to describe data while regular programming languages modify data. There are several differences between

SCSS and regular programming languages. Table 2.1 presents the differences between SCSS and other structured and object-oriented software.

Table 2.1: Comparison between Traditional and SCSS Software

Criteria	Traditional software	SCSS software
Modularized by	Modules/classes	SCSS block e.g. rule block, function directive block, mixin block, etc.
Parent module	Coordination of rest of the program is via main function, module, class, or method.	None
Program statement	Simple statements e.g. assignment.	Attributes and rule directives.
Control-flow structure	Sequence, branch, loop, and calls	Branch, loops, and calls
Data types	Variables/constants	Variables
Data definition	Data types are language-specific	SCSS relies on SASS Pre-processor data types
Programming scope	Programs for performing calculations e.g. computing the product of two numbers	Programs for formatting the presentation of web pages. e.g. assigning font size 12 to a paragraph

A simple alert rule block is shown in Figure 2.2 with three regular attributes, i.e. padding, font-size, and text-align. Padding has been used to generate a space of 15px around the element's content while font-size sets the size of text as 1.2em. Finally, text-align centers the content of the element where the alert class is implemented.

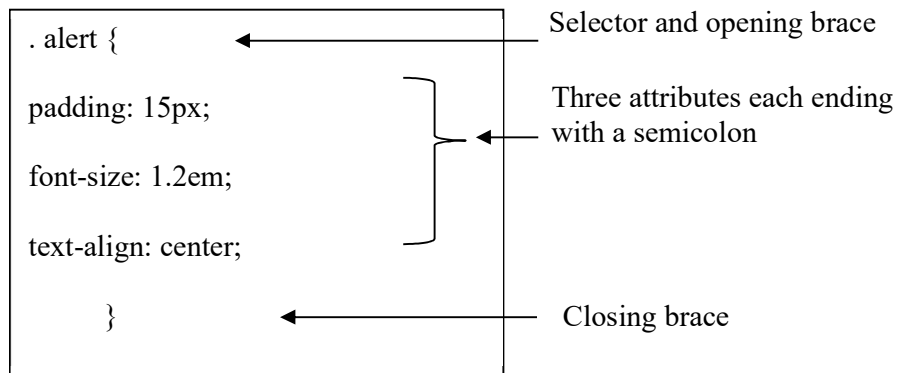


Figure 2.2: An Alert Rule Block

An illustration of multiple blocks is shown in Figure 2.3. The figure has one mixin block which can be called in various places of the code. It also has five rule blocks where the three of them are nested. The figure also demonstrates the use of variables and selector inheritance.

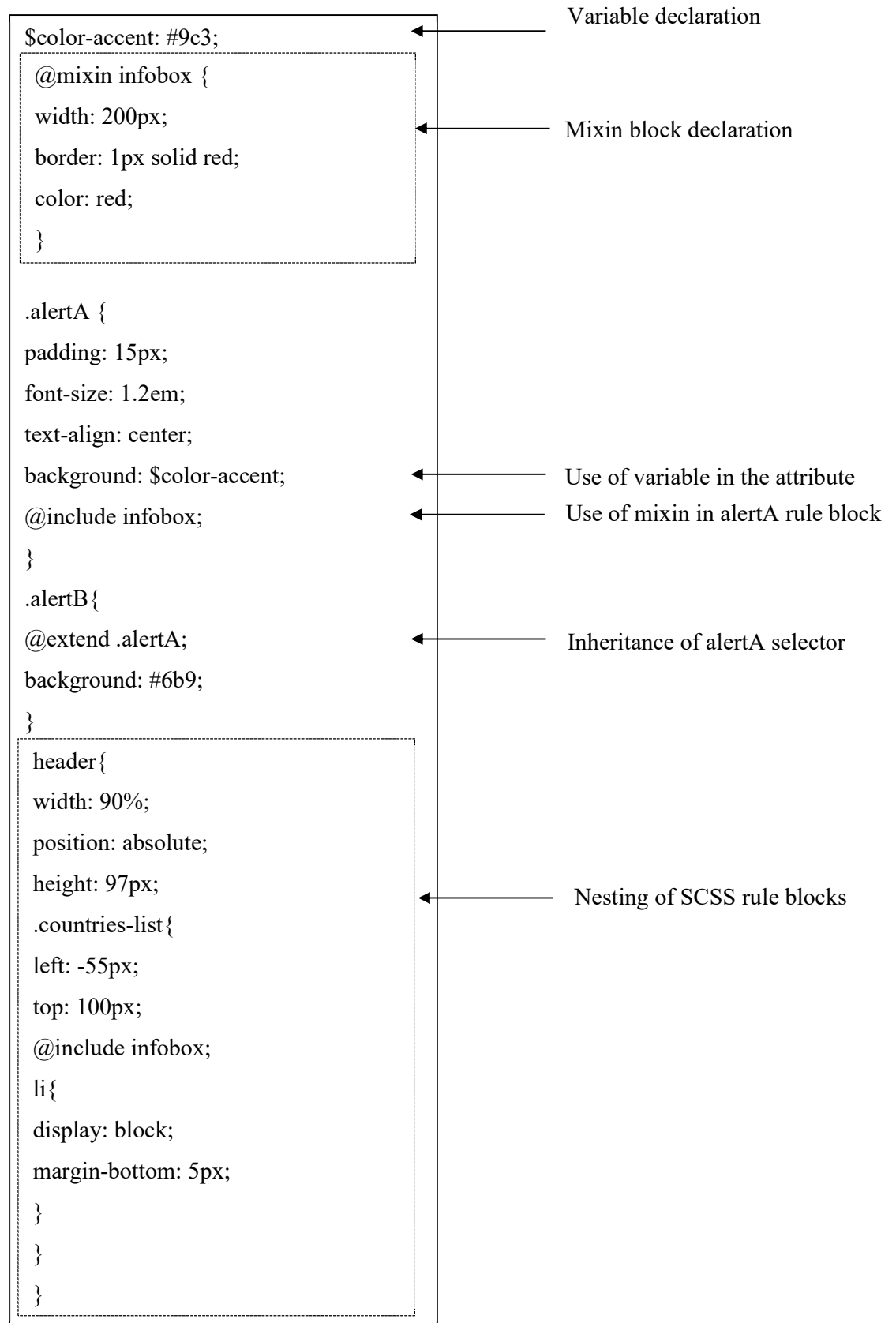


Figure 2.3: SCSS Code with Multiple Blocks

2.4 Software Complexity

Software complexity is how hard it is for a program to be understood, modified and tested (Harrison, , Magel, Kluczny, & Dekok, 1982; Boehm et al., 1978; Curtis et al., 1979). IEEE (1998) defines software complexity as an estimate of effort that is required to develop, maintain and execute the code. Software complexity is divided into various categories, such as computational complexity, representational complexity, functional complexity, organizational complexity and structural complexity (Mens, 2016; Henderson-Sellers, 1996). This categorization is an effort towards measuring the different dimensions of software.

2.4.1 Software Complexity Measurement

Measurement is the process of assigning numbers or symbols to various features of objects (Fenton & Bieman, 2014). In software engineering, measurement of software products is a process which involves defining, collecting and analyzing data, and it makes the designers understand and control their complexity (Fenton & Bieman, 2014; McGarry et al., 2002).

Measurement is based on formal models such as the Goal Question Metric (GQM) (Basili, 1992), Balanced Scorecard (BSC) (Martinsons, Davison & Tse, 1999), and the Entity-Attribute-Metrics model (EAM) (Fenton and Pfleeger, 1997). The GQM focuses on the organizational goals and has a wider scope which is at the project level (Basili, 1992) while the BSC which has its origin from strategic management, focuses on aspects of finance, clients, internal, and learning and development (Martinsons, Davison & Tse, 1999). The EAM model focuses directly on an object or entity such

as an SCSS code and is one of the most popular models for defining metrics(Muketha et al., 2011).

The proponents of EAM model Fenton and Pfleeger (1997), in effort to create an industry standard for determining the process of defining metrics identified three major stages, which include identification of entity to measure (e.g. project, product and process), identification of the entity’s attributes that need to be measured, and then deriving metrics for each of the attributes. These three steps are described in Figure 2.4.

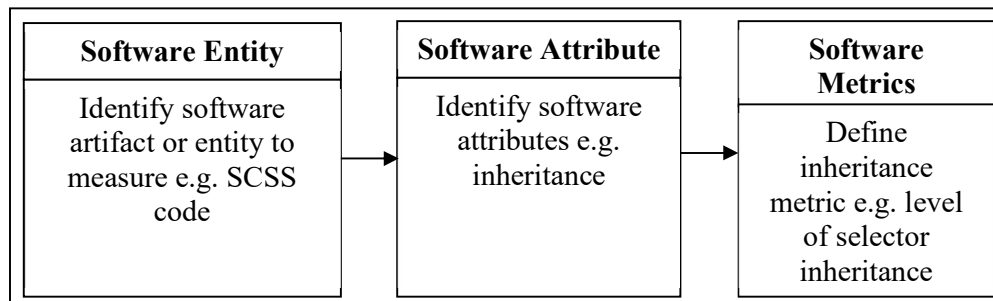


Figure 2.4: Software Metrics Definition Process

2.4.2 Software Complexity Attributes Classification

The various defined metrics in literature, target a particular type of software and are derived from a specific software attribute. For example, the popular McCabe’s Cyclomatic Complexity metric is based on the control flow attribute of software (McCABE, 1976), while Chidamber and Kemerer metrics such as the Depth of Inheritance Tree (DIT) and Number of Children (NOC) are based on inheritance attribute (Chidamber & Kemerer, 1994).

The identification of the right attributes for a given software can help in the evaluation and improvement of a software product (Morasca & Briand, 1997; Muketha, 2011). Software attribute is defined as the feature or property of a product (Bukhari et al., 2015) and these features of a product determine the type of measurement for it.

Several researchers have proposed classification schemes for software attributes in an effort to aid metrics definition (Fenton and Bieman, 2014; Fenton, and Pfleeger, 1997; Morasca, 2015, Daud and Kadir, 2014, Muketha, 2011; Falah & Magel, 2015; Mens, 2016; Henderson-Sellers, 1996). Some of these existing software attributes classification schemes provide a general treatment of complexity (Fenton & Bieman, 2014; Fenton & Pfleeger, 1997; Morasca, 2015), while others focus on a specific kind of complexity (Daud and Kadir, 2014; Henderson-Sellers, 1996; Muketha, 2011).

Fenton and Bieman (2014), proposed three categories for deriving the attributes to measure namely; process, product, and resources. The product category which is the focus of this study further classified attributes as internal or external attributes. Internal attributes are those that can be measured directly such as the size of code while external attributes are measured indirectly, such as reliability and maintainability. The limitation of this classification is that the modularity of the attributes such as control flow, data flow, cohesion, and coupling is not known.

In another study, Falah and Magel (2015) identified four ways of categorizing software attributes into product, process, people, and value to the customer. In this classification scheme, structural complexity falls under the product category. Structural complexity is further divided into control flow complexity, data complexity, and size attributes. The limitation of this classifications scheme is like the Fenton and Bieman

classification, in that, the level of modularity of the attributes is not provided, meaning we can't tell whether all the possible attributes of software are captured.

Daud and Kadir (2014) have classified software structural attributes into static and dynamic attributes. These authors identified three structural attributes, coupling, cohesion and complexity which fall under both static and dynamic. These attributes are the most popular in measuring service-oriented architecture (SOA). The limitation of this classification is that it identified the attributes from the literature and not from the structural properties of SOA. Meaning that the attributes identified may not fully represent SOA structural complexity.

Mens (2016) identified four major dimensions of software complexity, including theoretical complexity, the complexity of use, organizational complexity and structural complexity. Theoretical complexity was further divided into computational and algorithmic complexity, the complexity of use was divided into functional and usability, while structural complexity was divided into module level and system level. This classification scheme does not show what attributes can be derived from module level and system level hence it's not comprehensive.

Henderson-Sellers (1996) categorized software complexity into computational complexity, psychological complexity, and representational complexity. The author further divided psychological complexity into structural complexity, programmer characteristics and problem complexity. Structural complexity was further divided into intra and inter-module categories. The intra-module category is further divided into size, control flow, and cohesion attributes while the inter-module category is

specialized into the coupling attribute. This classification scheme is one of the most popular in terms of structural complexity classification (Muketha, 2011). However, its limitation is that it overlooks some new dimensions of structural complexity found in SCSS software and how they are implemented. The SCSS structural dimensions or features are discussed in depth in section 2.4.3.2. The Henderson-Sellers classification is illustrated in Figure 2.5.

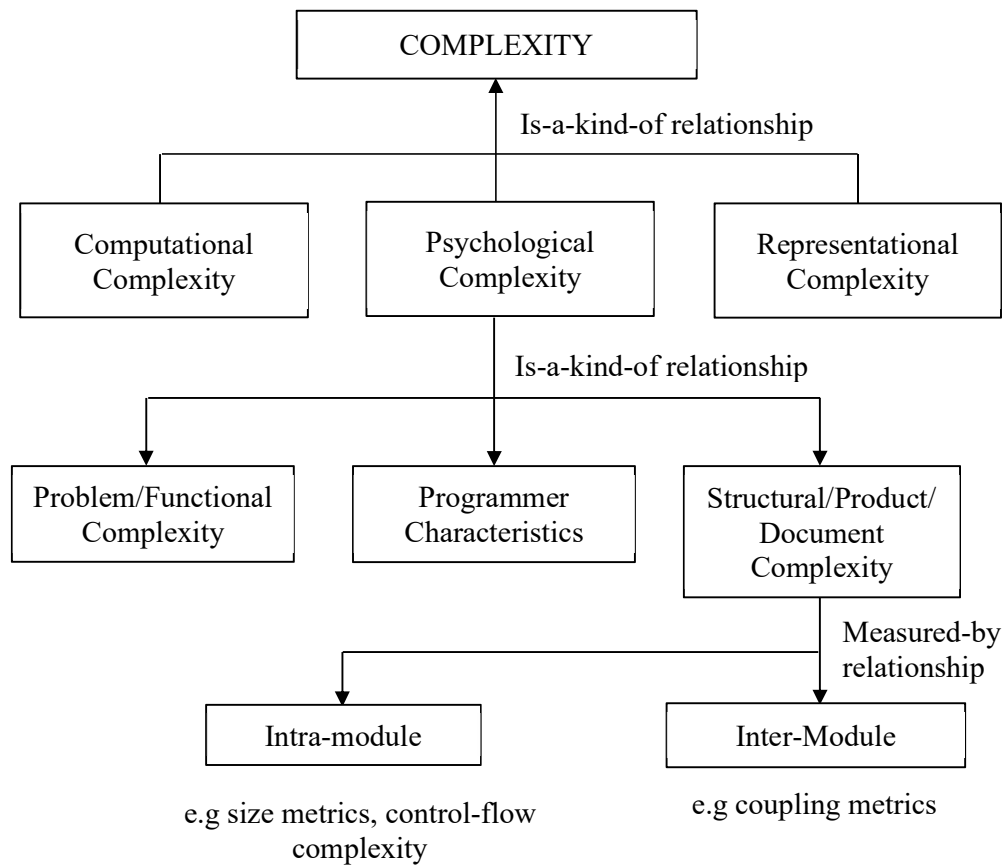


Figure 2. 5: Software Complexity Classification

(Source: Henderson-Sellers, 1996)

The part of structural complexity in the Henderson-Sellers classification scheme has been extended by introducing the hybrid category to the existing inter and intra-

module categories (Muketha, 2011). The hybrid attribute category blends intra-module and inter-module attributes. Muketha's framework is the more recent and comprehensive in the context of structural complexity. However, just like the Henderson-Sellers scheme, it is limited in that it overlooks some new dimension of structural complexity introduced in SCSS software. The uniqueness of SCSS dimensions are discussed in section 2.4.3.2. Figure 2.6 illustrates the classification framework. Intra-module attributes focused on an individual process which is equivalent to a module while inter-module attributes focused on the interaction of two modules. Finally, hybrid attributes blends intra- and inter-module attributes.

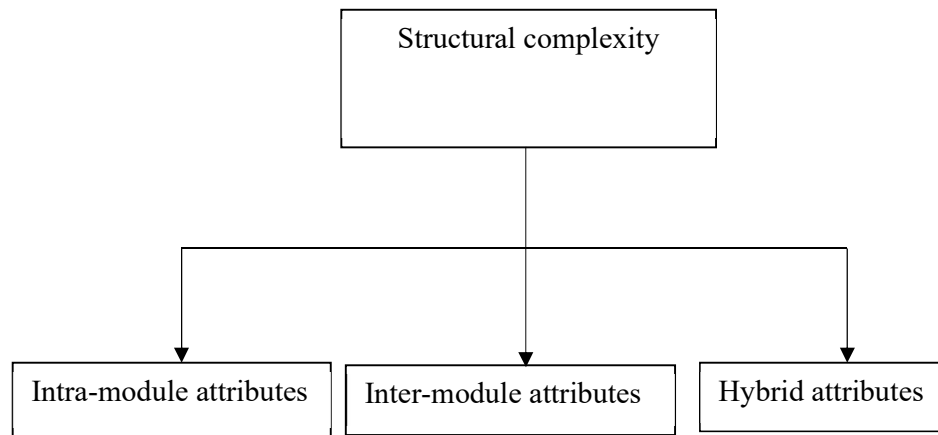


Figure 2.6: Extended Structural Complexity Classification

(Source: Muketha, 2011)

SCSS is an extension of Cascading Style Sheets (CSS) and it combines CSS features and traditional software features such as the use of variables, mixins, functions and control flows (Mazinanian and Tsantalis, 2016). This uniqueness of SCSS software means that the existing classification schemes cannot be used to sufficiently identify the structural attributes for SCSS.

2.4.3 Structural Complexity

Structural complexity is defined as the way in which the program elements are organized and interact within the software system (Ramasubbu and Kemerer, 2012; Darcy, Slaughter & Kemerer, 2005). It focuses on the design and structure of software (Laird and Brennan, 2006) and is concerned with the measurement of internal attributes which are assessed by the difficulty of performance of tasks such as the writing of codes, modifying and testing of software (Mens, 2016; Riguzzi, 1996).

2.4.3.1 Structural Complexity Properties for Traditional Software

Many authors consider size, length, coupling, and cohesion as part of structural complexity (Muketha, 2011; Henderson-Sellers, 1996; Khan, Mahmood, Amralla, & Mirza, 2016). For instance, the lines of code (LOC) metric, also called the physical lines of code, has been used as a size measure, and to some extent, as a complexity measure. The related logical lines of code (LLOC) metric, has been found to have higher accuracy when compared to LOC because it eliminates comment lines, auto-generated code lines, header files, ineffective code lines, compiler directives, labels, and empty case statements (Khan et al., 2016). For example, Adewumi et al. (2012) proposed size in terms of lines of rules for cascading style sheets while Misra & Cafer (2012) considered size in terms of lines of JavaScript code on condition that the only lines to be factored were those that consisted of variable or operators.

The concept of inheritance has been recognized as one of the most important features for software reuse. In object-oriented languages, inheritance supports class hierarchy design and captures the is-a relationship between a class and sub-class (Chung, & Lee, 1992). Inheritance has been studied in object-oriented languages extensively (Chung,

& Lee, 1992;, Chawla & Nath,2013; Gill & S. Sikka,2011;Misra et al., 2011). Though inheritance supports reuse, it can increase complexity if not used in the proper range (Chawla & Nath,2013). Style sheets provide a unique way of supporting inheritance because there are no classes and sub-classes as provided for in the object-oriented domain.

Nesting complexity has also been studied as an important property. Nesting reflects the level of nesting within constructs or control structures (Li, 1987). Constructs are such as if, case, for, while, and do-until can be nested. A statement that is at the innermost level is harder to understand, meaning that it contributes more to complexity than other statements (Chhillar & Bhasin, 2011). In SCSS, nesting occurs with selectors, and the more the selectors are deeply nested the more complex an SCSS code becomes (Frain, 2013).

Coupling has been defined as the measure of the strength of association established by a connection from one module to another (Stevens, Myers, & Constantine, 1974). It has been argued that the stronger the coupling between modules, the more difficult these modules are to understand, change and correct, resulting in more complex software. Coupling has been studied in the domain of procedural programming (Stevens et al., 1974) and object-oriented programming (Chidamber & Kemerer,1994; Li & Henry,1993; Abreu and Melo, 1996). While coupling as a complexity measure has been studied in procedural and object-oriented languages it has not been addressed in the stylesheets' domain. The SCSS language implements coupling in a unique way. The modules also known as rule blocks are coupled to each other through an external module, unlike in OOP where modules are directly coupled to each other.

Coupling in OOP is demonstrated in Figure 2.7, where Class B methods and variables can be accessed by both Class A and Class C. When a change is introduced in Class B methods and attributes, it affects Class A and Class C.

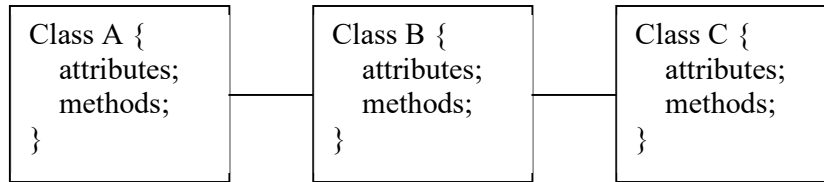


Figure 2.7: Coupling in OOP

Coupling in SCSS is demonstrated in Figure 2.8, where there are three rule blocks or modules namely, rule block A, rule block B and rule block C. These rule blocks are not connected to each other directly but share global data in form of mixins and variables. When a change is made to any of the mixin or variable, the effects are replicated in all the rule blocks.

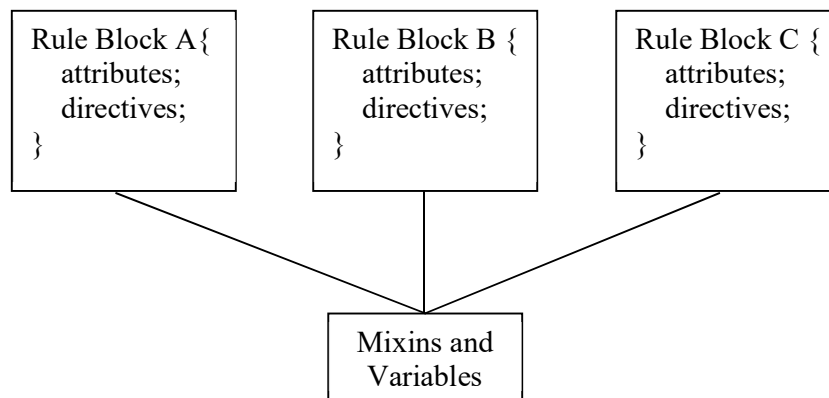


Figure 2.8: Coupling in SCSS

The aspect of cohesion is discussed extensively in the procedural and object-oriented domain. Cohesion is defined as the single-mindedness or relatedness of a module component (Bieman and Ott, 1994). When a module is highly cohesive, it means, all

the defined elements in a module perform a single task. Therefore, it's the goal of software designers to make a program as cohesive as possible.

The complexity of code can be expressed through control structures, and therefore, a program which implements control structures is regarded as more complex in comparison to the program without control structures (Chhillar and Bhasin, 2011). The complexity of a program is directly proportional to the cognitive weights of Basic Control Structures (Misra and Cafer, 2012). For example, iterative control structures like for loop, while, and do...while contribute more complexity than decision making control structures such as if...then...else.

2.4.3.2 Structural Properties for SCSS

SCSS combines the characteristics of CSS, such as the use of selectors, rule blocks, and declarations with those of traditional software such as inheritance, nesting, and coupling (Mazinanian and Tsantalis, 2016). The combination of these features makes the front web developers create more efficient and maintainable code.

Arbitrary and meaningful variables are one of the causes of complexity and if a variable's name is arbitrary given, then the comprehensibility of that code will be lower (Kushwaha & Misra, 2006). In essence, variable names should be meaningful in programming and if variable names are taken arbitrarily they may increase the difficulty in understanding four times more than the meaningful names (Kushwaha & Misra, 2006). SCSS introduced variables to enable developers to easily maintain stylesheets and they are prefixed with a dollar sign (\$). These variables can be global or scoped. Global variables are variables that are defined on its own line, and they

apply to the whole sheet, while scoped variables appear within a selector and will only appear to that selector and its children (Catlin & Catlin, 2011).

A rule block basically consists of properties and values which together form a declaration or an attribute. The more the number of attributes defined in a regular CSS rule block, the more complex it is (Adewumi et al., 2012). SCSS has more factors that contribute to its rule block complexity, for example, use of operators, use of variables, use of function calls, implementation of rule blocks within another rule block and use of control directives such as `@if`, `@for`, `@each`, `@while`, `@else if`, and `if ()` function.

SCSS provides a unique way of supporting inheritance by use of selector inheritance (Netherland, Eppstein, Weizenbaum & Mathis 2013). The selectors are extended in an SCSS rule block by use of `@extend` directive (Cederholm, 2013). This means that all the attributes of the inherited selector are implemented in the rule block that the selector has been extended. Figure 2.9 has code that illustrates the use of selector inheritance. The code has two rule block which has a selector named `.alarm` and is inherited by `.alarm-positive` selector. This means that the `.alarm-positive` selector will have five attributes or declarations i.e. padding, font size, text align, color and background.

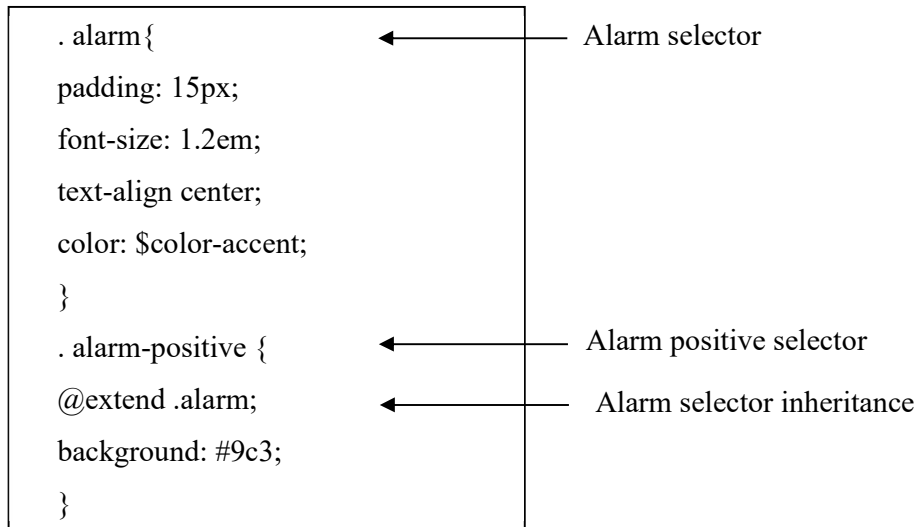


Figure 2.9: Selector Inheritance

SCSS allows nesting of rules inside each other instead of repeating selectors in a separate declaration (Cederholm, 2013). Figure 2.10 illustrates nesting by placing the message rule block inside infobox rule block.

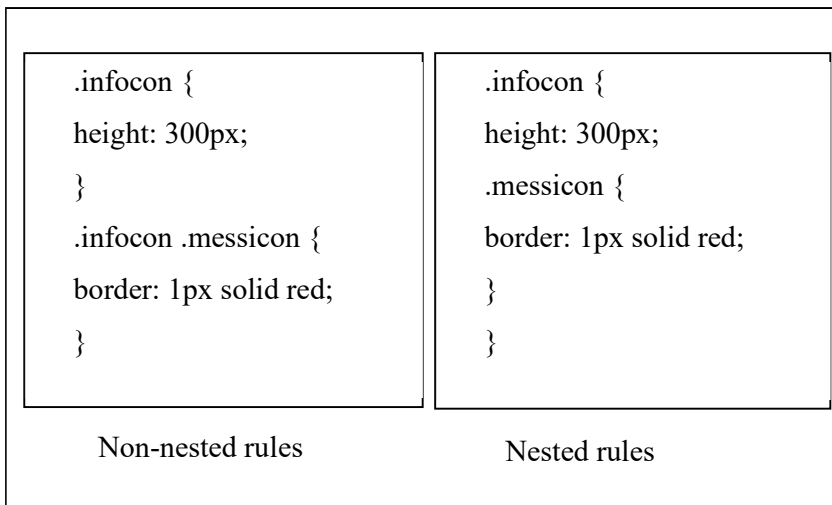


Figure 2.10: Nesting of Rules

SCSS consists of rule blocks, a rule block consists of properties and values which together form a declaration or an attribute. The more the number of components defined in a CSS rule block, the more complex it is (Adewumi et al., 2012). SCSS has

several components which contribute to rule block complexity, for example, attributes or declarations, operators, variables, function calls, control directives, include directive and extend directive.

In SCSS, coupling is manifested when the declared properties such as mixins and variables are used in several places of the code, meaning that the properties can be changed without realizing you are affecting multiple objects at once or not noticing which elements are being affected by the changes. In stylesheets, cohesion is viewed as the rule blocks having a single attribute (Adewumi et al., 2012).

SCSS implements a number of control directives which provide flow and logic to the CSS code. These control directives are; `if()`, `@if`, `@else`, `@for` loop, `@while`, and `@each` (Cederholm, 2013). In the Stylesheets field the use of control structures has not been considered by researchers in relation to software complexity.

2.5 Existing Software Complexity Metrics

The practice of defining software metrics has been continuing over the years for different kinds of software domains such as procedural, object-oriented, and web-based domains. Software metrics assess software from diverse perspectives to reflect the software internal quality such as maintainability (Arar & Ayan, 2016). The use of software complexity metrics has been recognized in software engineering as a way of controlling software complexity. Software metrics play a great role in measuring the level of success and failure of software, and this informs the software issues that require the attention of designers (Misra et al., 2018; Muketha et al., 2010b; Parthasarathy & Anbazhagan, 2006; Verner & Tate, 1992).

The following sub-sections present existing complexity metrics in the various domains of traditional programming languages, object-oriented languages, web-based languages including scripting web languages such as CSS.

2.5.1 Complexity Metrics for Traditional Software

Metrics such as Lines of Code (LOC), Function point (FP), McCabe cyclomatic complexity metric and Halstead's software science metrics are well known and frequently used to measure software complexity. These metrics targeted procedural languages which have major differences with SCSS syntax as shown in Table 2.1, meaning the metrics cannot be directly applied to SCSS.

2.5.1.1 Lines of Code (LOC)

The line of code is the oldest, simplest and most widely used metrics for calculation of program size (Debbarma, M., Debbarma, S., Debbarma, N., Chakma & Jamatia, 2013; Kandpal & Kandpal, 2012). LOC counts the number of instructions of a program in terms of SLOC (source lines of code) and excludes comments and blank lines. LOC is criticized for its lack of accountability, lack of cohesion with functionality, programmer and language dependent and lack of counting standards (Kandpal & Kandpal, 2012). There are alternatives to SLOC such as KLOC (thousands or Kilo of lines of code), KDSI (thousands of delivered source instructions), NCLOC (non-commented lines of code), and the number of characters or number of bytes (Kandpal & Kandpal, 2012). However, both LOC and its variants portray similar limitations.

2.5.1.2 Function Point (FP)

The Function Point metric was initially proposed by Albrecht and his contemporaries at IBM in the mid-1970 (Albrecht, 1979). The FP metric is used for systems measurement from a functional perspective regardless of the technology implemented. The metric basically breaks down a system into smaller components to enhance the understandability and analysis of the system (Praveen, Agarwal & Srivastava, 2018). Function points are weighted sum of inputs, outputs, queries, internal and external files, and are used to indirectly measure a project or application functionality (Kaur & Maini, 2016).

The ultimate measure of software productivity is the number of functions a development team can produce given a certain amount of resource, regardless of the size of the software in lines of code. FP metric addresses some of the problems associated with LOC and productivity measures, especially the difference in LOC counts that result because different levels of languages are used.

The FP technique depends on the counts of distinct types in the following five categories of external inputs, number of external outputs, number of logical internal files, number of external interface files, and number of external inquiries (Borade & Khalkar, 2013). External inputs refer to each user input that adds or changes data in an internal file, for-example input of data via input screen to add to the student's information. The external outputs are outputs by each user that provides application-oriented information, such as a report that contains the number of students pursuing a certain course. The logical internal files are the logical groups of data that are within the application's boundary and form part of the database. For example, group of

information related to the student such as registration number, student name, course, year of study, etc. The external interface files are all machine-readable interfaces such as data files which reside outside the application and are used for reference purpose only. For example, information concerning students' fees can be used by an academic application, but all the information on students' fees is maintained by the student's fees application. Finally, the external inquiries are the user inquiries, where an online input results in an immediate response in form of an online output, this input data doesn't update the internal logical files For- example a student can query on his results for a certain academic year (Borade & Khalkar, 2013).

The formula for computing the function points (FP) is:

$$FP = \text{Count Total} * [0.65 + 0.01 \sum (Fi)]$$

Where Count Total is the sum of all function points entries and F_i is the sum of degree of influence of each of the systems characteristics in consideration.

This approach is very difficult to implement practically because function points are computed manually, and an experienced person is required to use this technique (Praveen, Agarwal & Srivastava, 2018).

2.5.1.3 Halstead's Metrics

Halstead's metrics known as Halstead software science was introduced to build a theory that describes measurable software attributes. This metric assumes that a program consists of only operators and operands The attributes program length, volume, vocabulary, level, and programming effort were deemed as sufficient measurement attributes (Halstead, 1977). Scientific methods were introduced to

analyze software features and structure using Halstead's software science (Awode et al., 2017).

The Halstead metric is defined based on the following four numbers (Halstead, 1977):

n_1 : Number of non-repetitive (distinct) operators

n_2 : Number of non-repetitive (distinct) operands

N_1 : Number of all operators

N_2 : Number of all operands

The various measures derived from the four numbers are:

- i. The measure for program length, which is the total number of operators and operands

$$N = N_1 + N_2$$

- ii. The measure for the vocabulary of the program is the sum of the number of distinct operands and operators

$$n = n_1 + n_2$$

- iii. The measure for volume (V) is the count of the functional points in the program

Halstead defined the volume, V , of a program to be

$$\begin{aligned} V &= (N_1 + N_2) \log_2 (n_1 + n_2) \\ &= N \log_2 (n) \end{aligned}$$

- iv. The measure for program difficulty (D) is proportional to the total number of unique operators and total usage of operands

$$D = (n_1 * N_2) / (2 * n_2)$$

- v. The measure of effort (E) required to implement a program or understanding the program is directly proportional to difficulty and volume

$$E = D * V$$

- vi. The measure of number of bugs (B) expected in the program is proportional to effort

$$B = E * (0.667 / 3000)$$

- vii. The measure of time (T) taken to write the program is proportional to effort

$$T = E / S$$

Where S = 18 seconds.

Halstead metrics addressed LOC weakness where computer algorithm has been defined as a collection of tokens (Halstead, 1977). However, they have been criticized as being difficult to compute and depends on a code that is complete (Dhawan & Kiran, 2012). In addition, Halstead metrics are criticized as being confused and inadequate, though they are reasonable from a measurement theory perspective (Fenton, 1994). Though SCSS language makes use of operators and operands the Halstead metric doesn't capture the SCSS structural attributes, meaning it cannot be relied on to provide SCSS complexity measure.

2.5.1.4 McCabe Cyclomatic Complexity

McCabe's metrics are one of the most popularly used metrics that focus on the control flow structure of a program (McCabe, 1976). Cyclomatic complexity directly measures the number of linearly independent paths through a program source code (Madi, Zein, & Kadry, 2013; Khan et al., 2016).

The metric can be defined in two ways: The first approach is the basic formula for a single program and is computed as the number of decision statements in a program plus one. There are four basic rules for calculating Cyclomatic Complexity (Madi, Zein, & Kadry, 2013):

- i. Count the number of all if statements
- ii. Count the number of all cases in a switch statement, except the default and else case.
- iii. Count all the loops in the program i.e. do, while and for loop statements
- iv. Count all the try/catch statements

Finally, compute the total of all numbers from step 1 to step 4, then add 1

$$CC = \text{Number of Decisions} + 1$$

The second approach is used for large systems with several interconnected components. The graph is drawn and then the cyclomatic complexity is computed with the following formula:

$$V(G) = e - n + 2p$$

Where G represents the graph, n vertices are the number of nodes of the graph, e is the number of edges of the graph, and p is the number of connected components in a graph

The cyclomatic complexity measurement was designed to indicate the testability and understandability of a program. This metric is restricted because it simply counts decision nodes, and it assigns equal weights for both branch and loop statements (Cardoso, 2006; Debbarma et al., 2013). Loops are more complex than branches are more complex than sequences. In addition, the cyclomatic complexity doesn't consider the nesting level of control structures (Debbarma et al., 2013).

2.5.2 Complexity Metrics for Object-Oriented Languages

Over the years, researchers have proposed numerous object-oriented metrics that could be used to measure software complexity. These metrics targeted object-oriented languages and since SCSS language is not an OOP language it means that the metrics cannot be directly applied to SCSS. SCSS programs have major differences with OOP as described in Table 2.1. The subsequent sections present an analysis of these metrics.

2.5.2.1 Chidamber and Kemerer Metrics

There are several metrics defined for the object-oriented domain; one of the most popular is Chidamber and Kemerer (1994) metrics. The C&K metrics (1994) are; Weighted Methods per Class (WMC), Depth of Inheritance Tree (DIT) which measures the maximum length from the node to the root of the tree, where deeper trees constitute greater design complexity, Number of Children (NOC) which shows the number of immediate subclasses subordinated to a class in the class hierarchy, Coupling between object classes (CBO) which is the count of the number of other classes to which it is coupled, Response for a Class (RFC) which refers to a set of methods that can potentially be executed in response to a message received by an

object of that class and Lack of Cohesion in Methods (LCOM) which refers to the degree of similarity of methods. The larger the number of similar methods, the more cohesive a class is. These metrics have been empirically validated by several researchers (Denaro, Lavazza, & Pezze,2003; El-Emam, Melo & Machado, 2001; Basili, Briand & Melo,1996; Abreu, Melo & Abreu, F., 1996), however they have been found to be deficient theoretically (Koh, Selamat, Ghani & Abdullah, 2008; Li, 1998).

2.5.2.2 Mishra Inheritance Metrics

Mishra (2012) proposed two inheritance metrics namely; class level CCI (Class Complexity due to Inheritance) and program level ACI (Average Complexity of a program due to Inheritance). They metrics are promising in that they have been proven to be mathematically sound using Weyuker's properties. However, these metrics require empirical validation to ascertain whether they are can be useful indicators of external quality of software.

2.5.2.3 Abreu and Carapuca Metrics

Abreu and Carapuca (1994) defined five metrics that can be used to measure inheritance in object-oriented software. These include Total Children Count (TCC), Total Progeny Count (TPC), Total Parent Count (TPAC), Total Ascendancy Count (TAC) and Total length of inheritance chain (TLI).

The TCC is the number of classes that inherit directly, TPC is the number of classes that inherit directly or indirectly from a class, TPAC is the number of super classes from which a class inherits directly, TAC was defined and represents the number of

super classes from which a class inherits directly or indirectly, finally TLI is the total number of edges in the inheritance hierarchy graph. These metrics focused only on the inheritance aspect of object-oriented software and ignores other structural aspects of the software.

2.5.2.4 Lorenz and Kidd Metrics Suite

Lorenz and Kidd (1994) derived three metrics namely; Number of Methods (NMI), Number of Methods Overridden (NMO) and Number of New Methods (NNA). The NMI measure counts the total number of methods inherited by a subclass, while NMO counts the total number of methods overridden by a subclass and a superclass, and NNA counts the number of new methods in a subclass (Mishra, 2012) These metrics have been criticized as simplistic and just counts class properties, meaning they cannot be relied on to evaluate software quality (Baroni & Abreu, 2003; Harrison, Counsell & Nithi, 1997).

2.5.2.5 Li Metrics

Li (1998) proposed a set of six metrics to remedy the shortcomings of Chidamber and Kemerer metrics. The metrics are, Number of Ancestor Classes (NAC), Number of Local Methods (NLM), Class Method Complexity (CMC), Number of Descendants Classes (NDC), Coupling Through Abstract Data Type (CTA), and Coupling Through Message Passing (CTM).

The NAC metric measures the total number of ancestor classes from which a class inherits. The NLM metric counts the number of local methods in a class and are accessible outside the class. The CMC metric sums the internal structure complexity

of all local methods. NDC metric returns the total number of subclasses of a class. The CTA counts the total number of classes that are used as abstract data types. Lastly, the CTM metric returns the number of different messages sent out from a class to other classes, without considering the inheritance feature (Gupta, 2015). Though LI metrics addressed limitations in Chidamber and Kemerer metrics, they require modifications to strongly predict maintainability (Gupta, 2015).

2.5.2.6 MOOD Metrics Suite

The MOOD metrics are structural complexity metrics of the Object-oriented domain. These metrics were proposed in 1994 (eAbreau & Carapuça, 1994) they include; Method Hiding Factor (MHF), Attribute Hiding Factor (AHF), Method Inheritance Factor (MIF), Attribute Inheritance Factor (AIF), Polymorphism Factor (PF) and Coupling Factor (CF).

The MHF and AHF are proposed as measures of encapsulation. The MHF metric is the ratio of the sum of the invisibilities of all methods defined in all classes to the total number of attributes defined while AHF is the ratio of the sum of the invisibilities of all attributes defined in all classes to the total number of attributes. The MIF and AIF are inheritance-based metrics. The MIF metric is the ratio of the sum of the inherited methods in all classes to the total number of available methods while the AIF metric is the ratio of the sum of inherited attributes in all classes to the total number of available attributes in all classes. The PF Metric is the ratio of the actual number of the possible polymorphic situation for a given class to the maximum number of possible distinct polymorphic situations for the same class in consideration. Then the CF metric is the ratio of the maximum possible number of couplings not related to

inheritance (Neelamegam & Punithavalli, 2009). These metrics have been criticized for not being able to predict errors in classes (Shaik, Reddy & Damodaram, 2012).

2.5.2.7 Misra, Adewumi, Fernandez-Sanz and Damasevicius Metrics

Misra et al. (2018) proposed a suite of objected oriented complexity metrics. These metrics are Method Complexity (MC), Coupling Weight for a Class (CWC), Attribute Complexity (AC), Class Complexity (CLC) and Code Complexity (CC). The MC metric is computed by summing up all the assigned weights of methods in a class. The CWC metric sums the weights of calls and weights of called methods. AC metric computes the total number of attributes in a class. The CLC metric computes class complexity by summing up AC with MC and finally, the CC metric considers the complexity of classes brought by their interactions. The metrics emphasize on the inheritance aspect of code, where all classes in the same level are assigned same weight and subclasses weights are multiplied. These metrics have been proved to be theoretically sound, however, they need to be applied to industry projects to establish their usefulness.

2.5.3 Web-Based Metrics

Several researchers have defined metrics in the web domain. This section describes the different metrics based on the existing web-based languages. Since this study focuses on web-based metrics the availability of tool support is considered while discussing each metric.

2.5.3.1 Misra and Cafer Metrics

Misra and Cafer (2012) proposed JavaScript Cognitive Complexity Measure (JCCM), for measuring the design quality of scripts. The motivation for JCCM is to calculate the structural and cognitive complexity of JavaScript. This metric considered five factors that contribute to JavaScript complexity, the number of lines of codes, the number of meaningfully named variables (MNV), the number of arbitrary named distinct variables (ANDV), the cognitive weight of basic control structures (BCS's) and the number of operators (NO). The JCCM metric has been proven to conform to measurement theory, in addition, the metric has been empirically validated for understandability aspect of maintainability. However, there is no indication of tool support, meaning its difficult for the industry and researchers to adopt it. Furthermore, this metric targets JavaScript language, which means it cannot be used to measure programs written in the SCSS language due to the syntactical difference between JavaScript and SCSS.

2.5.3.2 Basci and Misra Metrics

Basci and Misra (2011) defined an entropy measure for the assessment of structural complexity of XML. The schema entropy metric measures the schema documents complexity due to elements structure diversity. This metric has been validated empirically although there is no evidence of theoretical validation and tool support.

Basci and Misra (2011) defined two document type definition (DTD) complexity metrics, Entropy metric: $E(\text{DTD})$ and Distinct Structured Element Repetition Scale metric: $\text{DSERS}(\text{DTD})$ so as to measure the structural complexity of schemas in DTD language. $E(\text{DTD})$ metric value is computed by considering equivalence classes in a

schema document. An equivalence class is the one that its elements have the same value of fan-in and fan-out and number of attributes. DSERS(DTD) metric measures the interface complexity of the schema document. The lower the E metric value and the higher the DSERS value the lesser the effort to understand the element structure. These metrics have been validated both theoretically and empirically, although no support for the automated tool has been seen so far.

Basci and Misra (2009) have also defined a design complexity metric for XML Schema documents(XSD) written in W3X XML Schema language. The metric C(XSD) measures the complexity of XSD based on the internal architecture of XSD components and recursion. It captures all the major factors responsible for XSD complexity. These factors are complexity based on elements and attributes definitions, elements and attributes group definitions, user-defined or built-in simple type and complex type definitions, elements definitions with no recursion and components that are included from external schema files. The proposed metric has been validated both through an experiment and theoretically through the Kaner and Briand's framework. Tool support for this metric has however not been seen although desirable.

Basci and Misra (2011b), described four XML web service metrics namely data weight of a web service description language (DW -WSDL) which is computed by defining the sum of the data complexities of each input and output messages, distinct message ratio (DMR) metric which counts the number of distinct structured messages, message entropy (ME) metric which measures the complexity of similar-structured messages and message repetition scale (MRS) metric analyses the varieties in structures of web service description language. These four metrics have been theoretically validated

using Kaner Framework and Weyuker's properties. They have also been validated empirically although there is no automated tool support.

The Basci and Misra metrics targeted DTD and XML software artifacts which have difference with SCSS in terms of syntax, therefore, they cannot be used to measure SCSS complexity.

2.5.3.3 Thaw and Misra Metrics

Thaw and Misra (2013) have defined an Entropy Measure of Complexity (EMC) for XML documents. The metric measures the reusable quality of XML schema documents. A high EMC value implies that the document is more reusable and that it contains inheritance features, elements, and attributes. Theoretical validation based on Kaner framework and Weyuker's properties were done on the metrics. The metrics were also validated empirically. As is the case with most metrics in this domain, no tool support has been seen for the EMC metric. In addition, the metric targeted XML documents which has major syntactical differences with SCSS, thus cannot measure SCSS complexity.

2.5.3.4 Tamayo, Granell and Huerta Metrics

Tamayo et al. (2011) defined three XML complexity metrics in geospatial web services, Data Polymorphism Rate (DPR), Data Polymorphism Factor (DPF), and Schemas Reachability Rate (SRR). DPR measures schema polymorphism, DPF measures the influence of polymorphic elements in the overall schema complexity, and SRR measures the fraction of imported hidden schema components by the subtyping mechanisms. These metrics have been empirically validated using a case

study and were found to be useful in detecting potential design problems for-example a component with too many information items. However, these metrics have not been theoretically validated, there is no evidence of tool support and they targeted XML software only, meaning that the metrics cannot measure SCSS complexity.

2.5.4 Adewumi, Misra and Ikhu-Omoregbe Metrics

Adewumi et al. (2012) proposed the first set of metrics in the stylesheet field. The metrics focused on CSS and they include, Rule Length (RL), Number of Rule Blocks (NORB), Entropy Metric (E), Number of Extended Rule Blocks (NERB), Number of Attributes defined per Rule Block (NADRB), and Number of Cohesive Rule Blocks (NCRB).

The Rule Length metric measures the number of lines of rules (or code) in a CSS file, and it's intended to measure the size of code. It is adapted from the popular line of code (LOC) metric. The formula for calculating rule length is

$$RL = \sum \text{rule statements}$$

Where RL is the rule length, and rule statements are the number of executable statements in a CSS file.

The limitation with the RL metric is that it does not consider the non-executable parts of CSS code such as white spaces or comment lines.

The Number of Rule Blocks metric counts the number of rule blocks in CSS code. The formula for calculating the Number of Rule Blocks is:

$$NORB = \sum \text{rule blocks in a CSS file}$$

Where NORB is the Number of Rule Blocks in CSS file, and a rule block consists of a selector and its declarations.

The NORB metric is similar to the RL metric because it's intended to measure the size of the code, meaning it achieves the same goal as RL.

The Entropy Metric puts the elements with the same structural complexity in the same category, this category is referred to as equivalence class(C). The entropy of a CSS document is based on n distinct class of elements and is calculated using the relative frequencies as unbiased estimates of their probabilities. $P(C_i)$, $i=1, 2, \dots, n$. The formula for calculating the entropy of CSS file is:

$$E = \sum P(C_t) \log_2 P(C_t) \text{ where } t=1 \dots n$$
$$= \sum (1/n) \log_2 (1/n)$$

Where E represents the Entropy metric value, P represents the probability of occurrence of distinct class elements (C).

The Entropy metric groups similar rule blocks and so when the entropy metric value is low the higher the structural similarity of rule blocks meaning the complexity of the CSS code is low.

The Number of Extended Rule Blocks metric counts the number of rule blocks that are extended in a CSS file. The formula for calculating the Number of Extended Rule Blocks of CSS file is:

$$NERB = \sum \text{extended rule block}(i)$$

where NERB represents the Number of Extended Rule Block and $i = 1 \dots N$

The Number of Attributes defined per Rule Block metric determines the average number of attributes defined in the rule blocks of a CSS file. The formula for calculating the Number of Attributes defined per Rule Block of CSS file is:

$$\text{NADR B} = (\text{Total number of attributes in all rule blocks} / \text{Total number of rule blocks})$$

Where NADR B represents The Number of Attributes defined per Rule Block

A higher NADR B metric value leads to higher complexity of the CSS code.

The Number of Cohesive Rule Block metric counts all rule blocks possessing a single attribute. The formula for calculating the Number of Cohesive Rule Block of CSS file is:

$$\text{NCR B} = \sum \text{rule block (i) possessing only one attribute}$$

Where NCR B represents the Number of Cohesive Rule Blocks and $i = 1 \dots N$

The higher the NCR B metric value the lower the complexity of CSS code.

These metrics are specifically for the CSS and the first of its kind in Stylesheets domain; and though they have been found to be practically valid, they have not been empirically validated and we cannot tell their mathematical soundness. The usefulness of the metrics in predicting the external quality of maintainability for CSS cannot, therefore, be assured.

2.6 Metrics Validation

There are two main stages required to validate software metrics, these are; theoretical validation, and empirical validation (Muketha et al., 2010b; Srinivisan and Devi, 2014).

2.6.1 Theoretical validation

The purpose of theoretical validation is to establish whether the proposed metrics are mathematically sound. Popular theoretical validation frameworks that are frequently cited in software metrics literature include Weyuker's properties (Weyuker, 1988), Briand's framework (Briand et al., 1996) and Kaner framework (Kaner, 2004). These three frameworks have been used extensively by metrics researchers to validate their metrics (Adewumi et al., 2012; Misra et al., 2018; Pichler et al., 2010; Geneves, 2012).

2.6.1.1 Weyuker's Properties

Weyuker proposed nine properties for validating software complexity metrics (Weyuker, 1988). Researchers have argued that it's not necessary for all the properties to be satisfied for a measure to be valid, but it must at least satisfy the majority of the properties (Basci & Misra, 2011b; Misra et al., 2018). These properties were adopted to suit SCSS syntax.

- Property 1 (Noncoarseness): $(\exists P) (\exists Q) (|P| \neq |Q|)$ where P and Q are two different modules. This property is satisfied when there exist two different modules P and Q such that $|P|$ is not equal to $|Q|$, meaning they don't return similar metric results.

- Property 2 (Granularity): Let c be a non-negative number. Then there are finitely many modules of complexity c . This property asserts that if a module changes then its complexity changes.
- Property 3 (Nonuniqueness): There can exist distinct modules P and Q where $|P| = |Q|$. This property affirms that two different modules can have the same metric value, this is to say that two modules have the same level of complexity.
- Property 4 (Design details are important): $(\exists P) (\exists Q)(P \equiv Q \ \& \ |P| \neq |Q|)$. There can be two modules P and Q whose external features look the same, however, due to different internal structure $|P|$ is not equal to $|Q|$. This property asserts that two modules with the same number of attributes and directives could return different metric values.
- Property 5 (Monotonicity): $(\exists P) (\exists Q) (|P| \leq |P; Q| \ \& \ |Q| \leq |P; Q|)$. This property asserts that if we concatenate two modules P and Q , the new metric value must be greater than or equal to the individual module.
- Property 6 (Nonequivalence of interaction): $(\exists P) (\exists Q) (\exists R) (|P| = |Q| \ \& \ |P; R| \neq |Q; R|)$ This property implies that if two modules have same metric value (P and Q), it doesn't necessarily mean that when each of the module is concatenated with similar module R , the resulting metric values are the same.
- Property 7 (Permutation): If you have two modules P and Q which have the same number of attributes in a permuted order, then $|P|$ is not equal to $|Q|$.
- Property 8 (Renaming property): if P is assigned as Q , then $|P| = |Q|$. Where you have two modules P and Q differing only in their selector names, then $|P|$ is equal to $|Q|$.
- Property 9 (Interaction increases complexity): $(\exists P) (\exists Q) (|P| + |Q| < |P; Q|)$. This property asserts that there exist two modules P and Q , where the

complexity metric value of the two modules when summed up is less than when the modules are interacting.

2.6.1.2 Briand's Property-based Framework

A property-based approach for software measurement has been proposed to formalize software attributes into size, length, complexity, cohesion, and coupling (Briand et al., 1996). Each of the five attributes contains a set of properties that should be met by the metrics being evaluated.

Size: The size of the code C is a function $\text{size}(C)$ characterized by the following three properties namely; non-negativity, null value and module additivity which should be satisfied by the size metrics.

- Property 1 (Non-negativity): the size of code must never be negative i.e., $\text{size}(C) \geq 0$.
- Property 2 (Null values): the size of the code is null if there is no module i.e. $\text{size}(C) = 0$.
- Property 3 (Module additivity): the code size is the summation of two modules (B1 and B2) i.e. $\text{Size}(C) = \text{size}(B1) + \text{size}(B2)$.

Length: The length of the code C is a function $\text{length}(C)$ characterized by the following five properties namely; Non-negativity, null value, disjoint modules, non-increasing monotonicity, and non-decreasing monotonicity.

- Property 1 (Non-negativity): The length of the code cannot be negative.
- Property 2 (Null value): The length of the code is null if the code has no modules.

- Property 3 (Disjoint modules): The length of a code that has two separate modules is equal to the lengths of the two modules.
- Property 4 (Non-increasing monotonicity): Adding relation between elements of a module does not increase the length of the code.
- Property 5 (Non- decreasing monotonicity): Adding relation from two modules does not decrease the length of code.

Complexity: The complexity of code C is a function complexity (C) that is characterized by the following five properties namely; Non-negativity, null value, disjoint module additivity, symmetry and module monotonicity.

- Property 1 (Non-negativity): The complexity of the code cannot be negative.
- Property 2 (Null value): The complexity of the code is null if the module is empty.
- Property 3 (Disjoint module additivity): The complexity of the code that has two modules is the summation of the complexities of the two modules.
- Property 4 (Symmetry): The complexity of a code is not dependent on how you choose to represent code elements relationships.
- Property 5 (Module monotonicity): The complexity of a code is no less than the sum of the complexities of any two of its modules with no relationships in common.

Cohesion: The cohesion of the code C is a function cohesion (C) characterized by the following four properties namely; Non-negativity and normalization, null value, monotonicity, and cohesive modules.

- Property 1 (Non-negativity and normalization): The cohesion of the code cannot be negative, and the measure should be independent of the size of the module.
- Property 2 (Null value): The cohesion of the code is null if the module is empty.
- Property 3 (Monotonicity): The relationship between modules cannot decrease cohesion.
- Property 4 (Cohesive modules): The relationship between modules cannot decrease cohesion when two modules showing no relationship are encapsulated.

Coupling: The coupling of code C is a function coupling (C) that is characterized by the following five properties namely; Non-negativity, null value, disjoint module additivity, merging of modules and monotonicity.

- Property 1 (Non-negativity): The coupling of code cannot be negative.
- Property 2 (Null value): The coupling of the code is null if there is no internal relation between the modules.
- Property 3 (Disjoint module additivity): The coupling of the code increases when more modules are added that share global data.
- Property 4 (Merging of modules): The coupling of the code decreases when two modules are merged.
- Property 5 (Monotonicity): The coupling of the code increases when the relationship between modules increases.

2.6.1.3 Kaner's Framework

Kane's framework is claimed to be more practical than the formal approach of Weyuker's properties and Briand's framework (Pichler et al., 2010). The Kaner framework evaluates software metrics to establish the purpose of the defined measure, scope of the measure, the attributes to measure, natural scale of the attributes to measure, natural variability of the attribute, metrics defined, measuring instrument, natural scale for the metric, natural variability of readings, relationship of attribute to the metric value and the natural and foreseeable side effects based on use of the instrument (Kaner, 2004).

2.6.2 Empirical Validation

Metrics researchers frequently employ experiments, case studies, or surveys in their effort to validate their new metrics (Muketha et al., 2010b; Srinivasan & Devi, 2014). Empirical validation is conducted to establish the usefulness of new metrics by the industry (Muketha et al., 2010b).

2.6.2.1 Experiments

Out of the three empirical strategies, experimentation is the more frequently used due to its formal, rigorous and repeatable characteristics (Muketha et al., 2010b, Wohlin et al., 2000). Experimental subjects are randomly assigned different treatments for the purpose of keeping one or more variables constant while other variables are manipulated. The effects of variable manipulation are observed, measured and interpreted (Muketha et al., 2010b).

Several software engineering experiments involve human subjects to investigate the cause-effect relationship (Easterbrook, Singer, Storey, & Damian, 2008). A family of experiments is encouraged such as conducting both subjective and objective experiments, to accumulate knowledge on a certain subject (Canfora, García, Piattini, Ruiz & Visaggio, 2005). Researches select different kinds of experimental designs. For-example within-subject design and between subject designs are some of the most popular designs (Muketha et al., 2011; Ko, Latoza & Burnett, 2015).

2.6.2.2 Case Studies

Case studies involve closer and deeper study on an attribute or relationship between several attributes. The context in which the attributes under study are being observed is an important factor in case studies (Wohlin et al., 2000). The limitation of this approach is that the data collection and analysis is open to researcher's bias, therefore, the selection of cases should follow a defined procedure (Easterbrook et al., 2008).

2.6.2.3 Surveys

A survey is a technique for collecting information from a sample of individuals in a certain population (Easterbrook et al., 2008). The results generated from the sample are analyzed and can be generalized to the population (Wohlin et al., 2000). This method is considered as a retrospective study where you study a situation and unlike experiments and case studies the variables cannot be manipulated (Fenton & Pfleeger, 1997). In a lot of situations, questionnaires are used to collect data, however other instruments such as structured interviews and data logging can be employed (Easterbrook et al., 2008).

2.7 Metrics Tools

A metrics tool is a static analyzer software which collects, computes and displays metrics values (Lincke, Lundberg, & Löwe, 2008). These tools enable programmers to analyze the source code of a programming language (Linos, Lucas, Myers, & Maier, 2007) and provide insight concerning the quality of the source code (Adewumi et al., 2015). The metrics tool has become a requirement for acceptability of any metrics proposed in the software industry (Adewumi et al, 2015). Therefore, it's imperative to develop a tool for the defined metrics.

Several metrics tools have been proposed such as Code Counter tool for C and C++ (CCCC) (Littlefair, 2001), OOMeter (Alghamdi, 2005), Prest (Kocaguneli, Tosun, Bener, Turhan, & Caglayan., 2009), a Multi-language metrics tool (Linos et. al, 2007), Business Process Metrics Tool (BPMT) (Muketha, 2011) and CSS Analyzer (Adewumi et al., 2015). In the absence of the tool, computation becomes a slow and tedious process, thus reducing the acceptability of the metrics in the software industry (Adewumi et al., 2015).

Several researchers have proposed metric tools to automate metrics computation. CCCC metrics tool by Littlefair, (2001) analyses C++ and Java files, by calculating the lines of code, cyclomatic complexity, lines of comments, information flow measures by Henry and Kafura (1984) and Chidamber and Kemerer object-oriented metrics suite (1994). OOMeter was developed to compute metrics for Java, C# source code and Unified Modelling Language models (UML) in eXtensible Mark-up Language (XML) format. The tool collects metrics for size, coupling, cohesion and complexity (Alghamdi et al., 2005). A Multi-language metrics tool (Linos et al., 2007)

has the capability to compute metrics for software developed with many languages under Microsoft Visual Studio .NET. Prest is an intelligent tool that extracts common static code metrics from C, C++, Java, JSP, and PL/SQL languages, this tool is capable of analyzing and predicting errors by applying machine learning concepts (Kocaguneli et al., 2009). Business Process Metrics Tool (BPMT) (Muketha, 2011) recognizes BPEL source code, collects and compute BPEL process metrics of size, information flow, and complexity. CSS Analyzer was developed to automate the computation of size metrics, cohesion, and complexity for cascading style sheets (Adewumi et al., 2015).

The metrics tool makes the work of computing metrics easier and therefore writing programs for the static analysis tool is desirable (Adewumi et al., 2015). The development of this tool is made easy, especially with the object-oriented paradigm languages such as Java which have String tokenizer to enable the splitting of a string into tokens and Parser classes which analyze string to find tokens this results to a reduction in coding. Programming languages in the .NET family provide the functionality to tokenize strings, meaning that they can be used to develop the metrics tools. These languages incorporate LINQ (language integrated query) features, that enable manipulation of data using a little amount of code which is expressive. These capabilities of .NET languages make it easy to build softwares that recognize particular programming language syntax such as software metrics tools (Muketha, 2011).

The efforts towards defining new software metrics have been going on over the years. However, the practice of developing a tool for the metrics is slowly being overlooked

and this is not acceptable (Spinellis, 2005; Adewumi et al., 2015). In literature, there is evidence of many proposed metrics which are validated but otherwise lack tool support.

Misra et al. (2018), defined a suite of object-oriented cognitive complexity metrics; Attribute Complexity, Method Complexity, Class Complexity, Message Complexity, and Code Complexity. This metrics suite was theoretically and empirically validated; however, no tool support was provided for these metrics.

Cognitive Weighted Inherited Class Complexity Metric (Maheswaran and Aloysius, 2018a), measures the inheritance complexity of a class and though it was found to be a better measure than Weighted Class Complexity (WCC) and Attribute Weighted Class Complexity (AWCC) its use in the industry could be undermined by the lack of tool.

The interface-based cognitive weighted class complexity metric (ICWCC) is a promising metric for measuring class complexity based on defined interfaces (Maheswaran and Aloysius, 2018b). However, there is no evidence of tool support. Mishra (2012), proposed two metrics, Class complexity due to Inheritance and Average complexity of a program due to inheritance, these metrics have been theoretically validated, however, they don't have tool support.

In the web domain, there are several metrics that are theoretically and empirically validated; however, they lack tool support. For example, Thaw and Misra (2013) defined an Entropy Measure of Complexity metric for the measurement of reusability

of XML schema documents. Basci and Misra (2011) defined an entropy metric and distinct structured element repetition scale metric for the measurement of structural complexity of document type definition schemas. To measure JavaScript complexity Misra and Cafer (2012) defined JavaScript Cognitive Complexity Measure (JCCM).

2.8 Software Maintainability

There are several software quality models which recognize maintainability as an important aspect of quality. A software quality model is defined by ISO/IEC IS 9126-1 as a set of characteristics that forms the basis for quality requirements specification and evaluation of software products. Maintainability is defined as the ease with which a software product can be understood, modified and tested (Boehm, 1978; IEEE, 1993; Bandi et al., 2003). The most popular quality models are, McCall Model (McCall, Richards, & Walters, 1977), Boehm Model (Boehm et al., 1978), Dromey Model (Dromey, 1995), ISO 9126 Model (ISO, 2001) and ISO 25010 Model (ISO/ IEC CD 25010, 2008).

The McCall model (McCall, Richards, & Walters, 1977), views product quality in terms of product review, product operation and product transition. This model was able to link the software quality characteristics with metrics; however, its limitation is that it lacks accuracy in measurement quality (Dubey, & Ghosh & Rana, 2012; Miguel, Mauricio & Rodríguez, 2014). Maintainability software quality is classified under product review and has three sub-attributes, simplicity, conciseness and self-descriptiveness (Miguel, Mauricio & Rodríguez, 2014) as described in Figure 2.11.

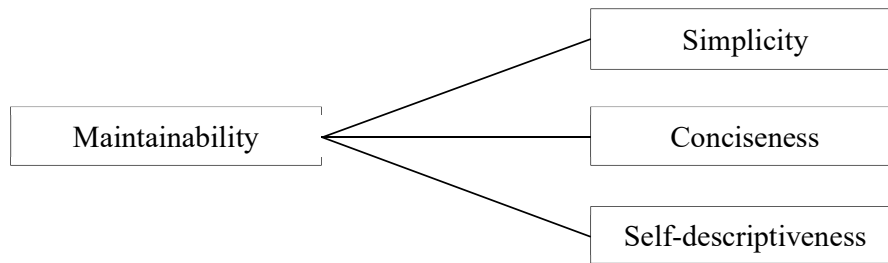


Figure 2.11: McCall Maintainability Sub-characteristics

Source: McCall, Richards, & Walters, 1977.

Boehm model software quality model was an improvement on McCall model. The maintainability aspect was recognized as an important aspect of software quality. Three sub-characteristics of maintainability were defined as; understandability, modifiability and testability. According to Boehm (1978) understandability is defined as the easiness with which the software can be comprehended or understood, modifiability is the easiness to which the software can be changed to fit in new requirements and testability is the easiness with which you can identify errors in software and correct them. Figure 2.12 visualizes the Boehm maintainability model.

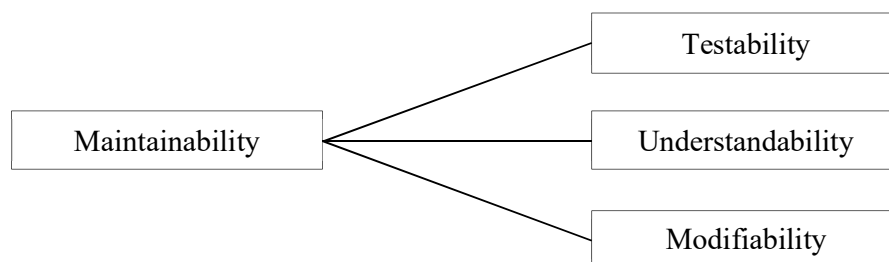


Figure 2.12: Boehm’s Maintainability Sub-characteristics

Source: Boehm et al., 1978.

Dromey (1995) proposed a software quality model to aid in the evaluation of software in terms of requirements, design, and implementation This model recognizes

maintainability as a software quality attribute, amongst other qualities such as functionality, reliability, efficiency, portability and reusability. However, the drawback of this model is that it doesn't specify the sub-attributes that define maintainability (Tomar & Thakare, 2011).

ISO 9126 Model is a standard for evaluation of software and is majorly divided into four parts, quality model, external metrics, internal metrics and quality in use metrics. It identifies maintainability as a high-level software quality characteristic, among functionality, reliability, usability, efficiency, and portability. The maintainability sub-characteristics are defined as analyzability, changeability, stability, testability, and maintainability compliance as shown in Figure 2.13.

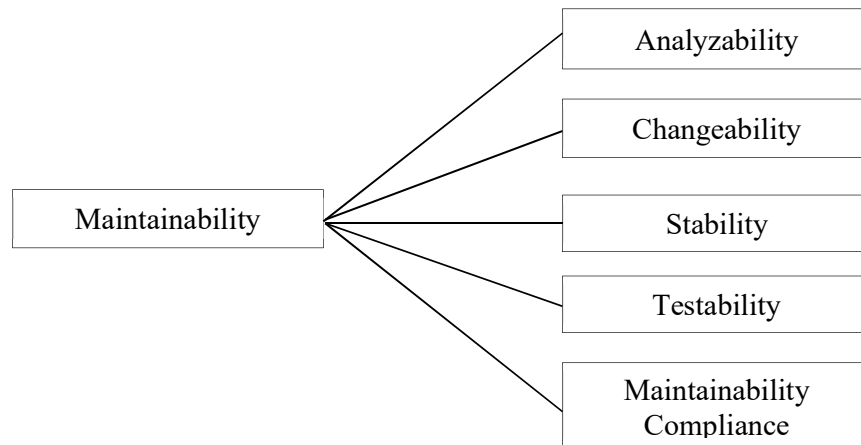


Figure 2.13: ISO-9126 Maintainability Sub-characteristics

Source: ISO, 2001

The ISO 25010 Model (ISO/ IEC CD 25010, 2008) extended the ISO-9126 model. The maintainability quality has eight sub-characteristics, modularity, reusability,

analyzability, changeability, modification, stability, testability and compliance. Figure 2.14 illustrates the maintainability model.

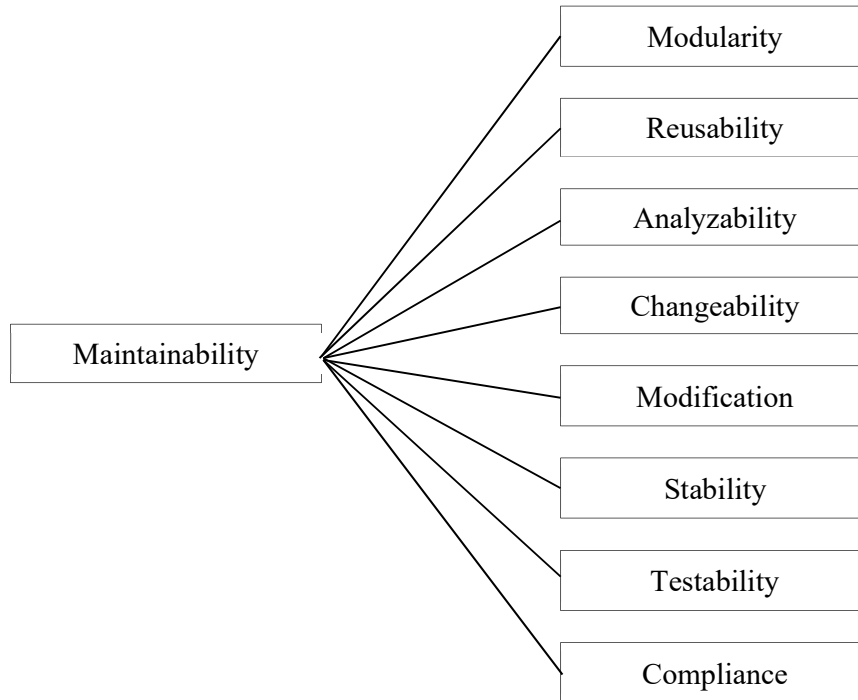


Figure 2.14: ISO-25010 Maintainability Sub-characteristics

Source: ISO/ IEC CD 25010, 2008

The Boehm maintainability model was selected for this study because it's more meaningful from the designer and programmer perspective (Al-Badareen, Selamat, Jabar, Din & Turaev, 2011). The ISO-9126 and ISO-25010 though more recent software quality models than Boehm model, have a focus on user perspective, and therefore were not considered in this study. Moreover, several researchers have sought to understand the maintainability of various software and they focused on one or all of these aspects of maintainability, that is, understandability, modifiability and testability (Muketha, 2011; Rizvi & Khan, 2010; Kiewkanya, Jindasawat, & Muenchaisri., 2004;

Genero et al., 2003). Therefore, the researcher believes that by studying the three sub-attributes of maintainability fully represents software maintainability.

The measurement of external software quality such as maintainability is not possible directly, therefore, researchers identify and measure internal attributes such as complexity i.e inheritance, coupling, cohesion and nesting to predict the external quality of maintainability (Lu et al.,2016; ; Almugrin, Albattah, & Melton, 2016; Kumar, Naik, & Rath; 2015; Muketha, 2011; Mishra and Sharma, 2015).

2.9 Gaps Identified in Literature

A detailed literature survey was done after which several gaps were identified. The gaps are summarized in Table 2.2.

Table 2.2: Identified Gaps

Type of Gap	Description of the Gap
Existing classification frameworks	Existing classification frameworks do not fully identify the SCSS complexity attributes because of the unique features found in SCSS when compared to other software
Existing metrics	A number of metrics exists for procedural, object-oriented, web-based domains and style sheets field. However, they cannot be applied to SCSS language due to its unique structural features.
Existing metrics tools	There are several existing static metrics tool, however, they cannot compute SCSS complexity metrics.

2.10 Theoretical Framework

This research was based on the following theoretical foundations, the EAM model (Fenton and Pfleeger, 1997), extended structural complexity classification scheme (Muketha, 2011), Boehm Model (Boehm et al., 1978), Weyuker's properties (Weyuker, 1988) and Kaner framework (Kaner, 2004).

The EAM model was used to guide in the definition of new metrics and it consists of three steps, entity identification, attributes identification and metrics definition based on the attributes. This model was extended to include a fourth step referred to as metrics tool development. This added step was after scrutiny of existing metrics and it was discovered a lot of metrics may not be implemented by the software industry because they lack tool support. The inclusion of tool development will enforce metrics acceptability by the software engineering community.

The extended structural complexity classification scheme (Muketha, 2011) which categorizes structural complexity into an intra-module attribute, inter-module attribute and hybrid attribute was further extended to include a new category known as an extra-module attribute to cater for the structural properties of SCSS language. This classification aided in the identification of the attributes that affect SCSS structural complexity, which can then be used to predict the maintainability of SCSS code.

Boehm model (Boehm et al., 1978) was used to identify the maintainability sub-characteristics of understandability, modifiability, and testability. These characteristics were used in this research as dependent variables to predict the maintainability of SCSS software.

Weyuker's axioms were used to determine the mathematical soundness of the proposed SCSS metrics (Weyuker, 1988). The theoretical validation of the metrics assures the construct validity of experiments. The nine properties of Weyuker were redefined to fit in the context of SCSS syntax.

The Kaner's framework (Kaner, 2004) was adopted and used to gauge the practicality of the defined metrics. This framework forms part of theoretical validation and enhances the construct validity of experiments.

2.11 Conceptual Framework

Conceptual framework consists of related concepts or views which can explain or make one understand the research problem under investigation. The relationship between the concepts is established, and in a research report, they are referred to as independent and independent variables. These concepts are identified in literature through theoretical and empirical findings (Imenda, 2014; Liehr and Smith 1999).

Several studies show that structural complexity metrics such as module complexity, coupling, nesting, and inheritance are useful in establishing the maintainability of software. These studies show that when complexity increases the understandability, modifiability and testability of code reduces (Lu et al.,2016; Kumar, Naik, & Rath; 2015; Muketha, 2011).

In this study the researcher investigated how the structural complexity attributes of SCSS (independent variables) such as block complexity, nesting, inheritance, and coupling, which are computed by the proposed metrics can be used to predict the

maintainability of SCSS code through its sub-attributes namely; understandability, modifiability and testability (dependent variables). The moderating variables identified that could potentially affect the studied relationships between independent variables and dependent variables were programmer experience and programmer level of education. These variables and their relationships are shown in Figure 2.15. The conceptual framework was used to design the controlled laboratory experiment presented in chapter seven.

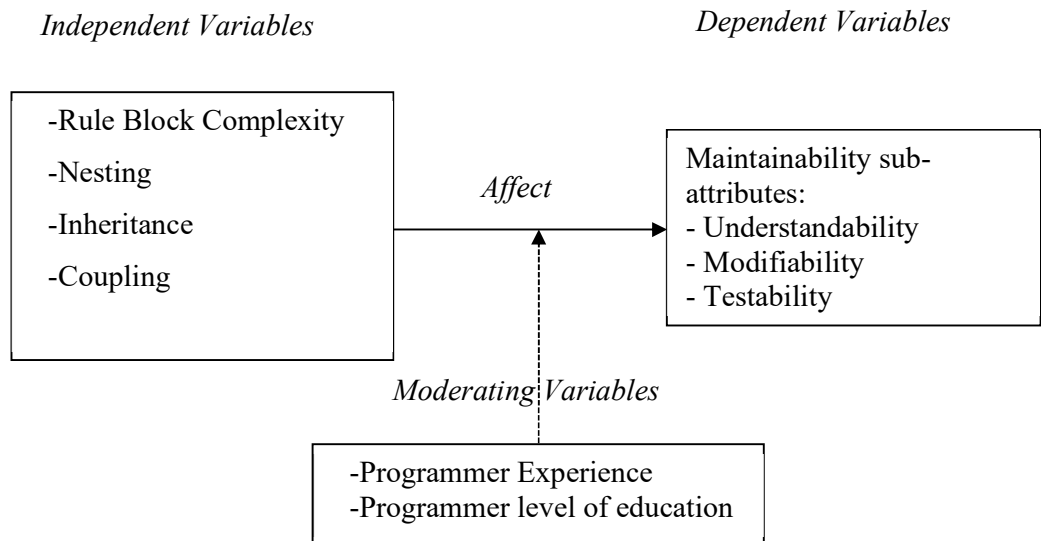


Figure 2.15: Conceptual Framework

2.12 Chapter Summary

In this chapter, existing attributes classification systems and software metrics were identified and examined.

There were very few comprehensive structural complexity classification schemes found and they cannot be used to identify all the attributes of SCSS code. The literature

also showed that there are so many metrics defined in the object-oriented domain, however, there are few metrics proposed in the web-domain and there are even fewer in stylesheets field. The only existing stylesheets metrics are for CSS which their mathematical soundness has not been proved and have not been empirically validated. In addition, there is no metric defined for CSS pre-processors. It was also found that many metrics don't show evidence of metric tool support. The findings in the literature are worrying because the complexity of the code is principally measured through metrics and for them to be useful, they must be theoretically and empirically validated. Moreover, for the metrics to be adopted by the software industry there is a need for the development of a static metrics tool.

This formed the motivation to develop an SCSS structural complexity framework, define metrics for SCSS language, theoretically and empirically validate the new metrics and develop a metrics tool.

CHAPTER THREE

RESEARCH METHODOLOGY

3.1 Introduction

This chapter describes the methodological approach for the study. It presents the research philosophy, research design, research process describing the steps taken to achieve research objectives, research strategy, population, sampling, data collection, data analysis, and ethical issues.

3.2 Research Philosophy

Research philosophy also referred to as a research paradigm is a set of beliefs that guides action in research (Creswell, 2014). There are four main research philosophies widely discussed in the literature: positivism, realism, interpretivism, and pragmatism (Saunders, Lewis, & Thornhill., 2012). Positivism describes an approach to the study where in order to understand a phenomenon, it must be measured and evidence provided (Hammersley,2013). Experiments establish the relationship between independent and dependent variables (Cohen, Manion, & Morrison, 2011), they provide formal propositions, test hypothesis and causal inferences are described from the data collected (Myers & Avison, 2002). Therefore, to achieve the objectives of this research positivism philosophy was chosen.

3.3 Research Design

A research design consists of the conceptual structure or the blueprint for collecting and analyzing data (Kothari, 2004). Research design can either be exploratory or explanatory research. This study implemented explanatory research design which seeks to test theories. A research design shows the overall direction of the research

and is dependent on research objectives, time for research, costs to be involved and the researcher's skill (Remenyi, 2005).

3.3.1 Research Process

This research process shows the steps that were followed to achieve the objectives of this study. In this research four main steps were involved. The first step, addressed objective one, where the researcher identified all the possible structural complexity attributes that indicate the complexity of SCSS. The second step addressed the second objective, where new metrics were defined guided by the EAM model, the metrics were theoretically validated using Weyuker's properties and Kaner's framework to ensure that they are constructively valid. Thirdly, to address the third objective, a metrics tool was designed to promote the acceptability of the proposed metrics and to ease the process of gathering and computing the metric values and lastly to achieve the fourth objectives metrics were empirically validated to affirm that the metrics are good predictors of SCSS code maintainability. A summary of these steps was described in Figure 3.1.

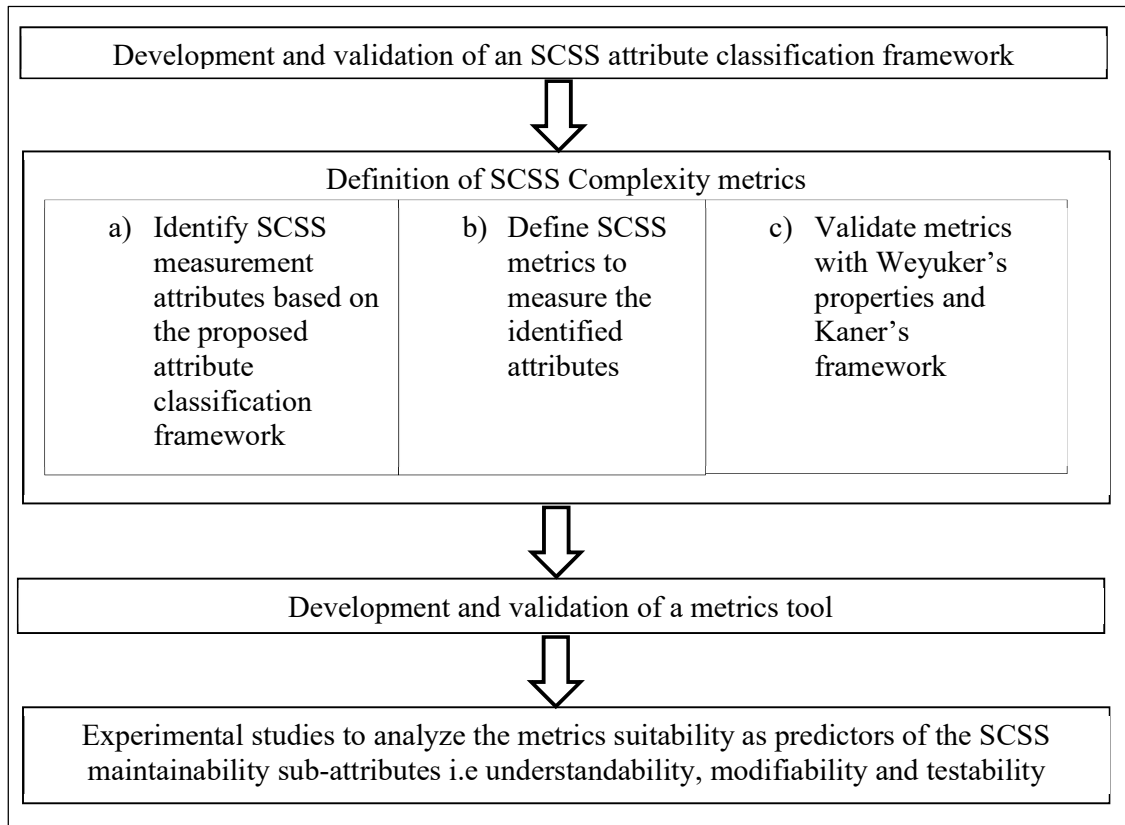


Figure 3.1: Research Process

3.3.1.1 Development of an Attribute Classification Framework

This research focused on the development of an SCSS attribute classification framework with the aim of identifying the structural complexity causing attributes for SCSS language. The framework identified four categories of attributes namely, intra-module attribute, inter-module attribute, hybrid attribute, and extra-module attribute.

In the intra-module attribute, two aspects of complexity were identified, size and control-flow complexity, the inter-module attribute category identified inheritance and nesting complexity, under hybrid attribute an association complexity was identified and finally in the extra-module attribute information flow complexity was identified.

These various types of complexities were used to identify the needed attributes as shown below:

1. Inter-module attributes
 - i. Nesting complexity - Nesting factor for SCSS
 - ii. Inheritance complexity- Selector use inheritance level
2. Hybrid attribute
 - i. Association complexity – Average block cognitive complexity for SCSS
3. Extra-module attribute
 - i. Information flow complexity- Coupling level for SCSS
4. Intra-module attribute

This category assisted in identifying several base metrics including:

- i. Number of attributes
- ii. Number of operators
- iii. Number of rule-blocks
- iv. Weighted Control directives

The framework classified nesting complexity and inheritance complexity as inter-module attributes. The nesting complexity attribute targeted the nesting feature of the SCSS code. This attribute allows the SCSS designers to understand the extent to which nesting has been implemented. The inheritance complexity attribute targeted the inheritance of selectors in SCSS. This attribute returns the inheritance level of SCSS code.

The association complexity attribute was categorized under the hybrid level of SCSS structural complexity framework. This attribute identified all the complexity causing

attributes in a rule block. The motivation was to understand the cognitive complexity of SCSS blocks.

The information flow complexity attribute was placed under the new derived category known as extra-module. This attribute helps SCSS designers understand the extent to which the various SCSS block are connected to each other, that is, how the information flows from one rule block to another rule block. This attribute does not consider the inheritance aspect.

The final categories of complexity attributes are the size and control-flow complexity which are classified as intra-module attributes. The size of SCSS code, is contributed to by the number of declarations, and number of operators which were implemented as base metrics to compute Average Block Cognitive Complexity for SCSS, while the number of rule blocks was implemented as base metrics to compute Nesting factor for SCSS, Selector use inheritance level and Coupling level for SCSS. The control directives contribute to the control-flow complexity of SCSS code and was implemented as base metric in the computation of Average Block Cognitive Complexity for SCSS.

The validation of the framework was conducted to check whether its relevant and comprehensive in identification of all possible SCSS structural complexity attributes. To achieve this an expert opinion survey was carried out (see Appendix 1).

3.3.1.2 Definition of SCSS metrics

The definition of SCSS metrics followed the EAM model. These metrics are Average block cognitive complexity for SCSS ($ABCC_{scss}$), Nesting factor for SCSS (NF_{scss}), Selector use inheritance level (SUIL) and Coupling level for SCSS (CL_{scss}).

The $ABCC_{scss}$ metric is a hybrid metric and was motivated by the existing Number of attributes defined per rule block (NADRB) metric. The NF_{scss} metric falls under the inter-module level and considers the nesting depth and nesting breadth of SCSS code, while SUIL modified the class inheritance factor while at the same time considering the uniqueness of inheritance in SCSS. Finally, CL_{scss} which falls under extra-module category was defined to represent the unique way of information flow while excluding inheritance.

3.3.1.3 Theoretical Validation of SCSS metrics

Theoretical validation was performed on the four metrics with Weyuker's properties (Weyuker, 1988) with the aim of finding out if they were mathematically sound. Validation with Kaner framework was also done to check the metrics' practicality.

Weyuker's properties are a popular technique for metrics validation. The four defined SCSS metrics were validated using Weyuker's properties. Further validation was carried out using Kaner framework to prove the practicality of each of the metric.

3.3.1.4 Development of a Metrics Tool for SCSS

The development of a static analyzer metrics tool is necessary for the new metrics to be appreciated by the software industry. Therefore, a static analyzer metrics tool was

designed and developed to recognize the SCSS syntax, gather and compute metrics. The metrics tool was developed using Microsoft Visual C# 2017 programming language.

The tool was tested to ensure its working properly and validation through experiments was carried out to confirm the tools effectiveness, efficiency, usability, and functionality.

3.3.2 Research Strategy

This study employed survey and experimental research strategies. An online expert opinion survey was used to validate the SCSS complexity attributes classification framework. Expert opinion technique is used to identify problems, clarify some technical issues and evaluate products (Whitfield, Ruddock & Bullman, 2008). The data collected through expert opinion is reliable because the respondent's technical knowledge and competence is high (Libakova & Sertakova, 2015). The experimental design was used because it's very effective in supporting hypotheses about cause and effect relationships (Bhattacharjee, 2012). The metrics tool was validated using between subject design, where the researcher assigned SCSS files randomly to the subjects. The proposed metrics were validated using between-subject design and an experiment consisting of both subjective and objective parts was performed. The between-subject design has the advantage of reducing error variance associated with individual differences.

To conduct validation of new metrics, several SCSS files were provided to subjects.

- a) The subjective part of the experiment followed the stipulated procedure below;
 - i. The subjects studied each of the file provided for a given time period.
 - ii. The subjects were required to rate each of the file provided in terms of understandability, modifiability, and testability using a Likert scale.

- b) The objective part of the experiment followed this stipulated procedure;
 - i. The subjects were given a number of activities to perform based on the programs;
 - ii. The first set of activities required subjects to indicate starting time to ending time in terms of understandability of each of the files provided.
 - iii. The second set of activities required subjects to indicate starting time and ending time on the modifiability of the different files.
 - iv. The third set of activities required subjects to indicate starting time and ending time on the testability of each of the files provided.

3.4 Population

The target population is the entire set of units for which the study data are to be used to make inferences and it defines those units for which the findings of the study are to be generalized (Dempsey, 2003). To validate the SCSS attributes classification framework the target population selected for this study was industry based SCSS programmers.

The use of students as subjects in software engineering experiments is a valid simplification of reality required in laboratory contexts (Falessi et al., 2018).

Moreover, according to Salman, Misirli & Juristo (2015), there are no major differences observed in results for experts and students. Therefore, the target population for validating metrics tool and carrying out experiments were Murang'a University of Technology fourth-year students pursuing Bachelor of Science in Information Technology, Bachelor of Science in Software Engineering and Bachelor of Business Information Technology. In addition, the third-year students pursuing Bachelor of Science in Software Engineering students were involved in this study. Only the students who were trained on SCSS language in the mentioned groups formed the target population

3.5 Sampling Strategy and Sample Size

Sampling process forms the basis for selecting a sample to estimate the outcome on a bigger group (Kumar, 2011). According to Kothari (2004), a sample size should be determined by a researcher and must consider whether the nature of the universe is homogenous or heterogeneous. In the case of the homogenous universe, small sample size can serve the purpose, but if there are many class groups to be formed, then a large sample is a requirement. Secondly, the researcher should consider if the items are to be intensively and continuously studied, and if so, the sample should be small. The sampling technique determines the size of the sample, standard of accuracy and acceptable confidence level.

This research employed snowball sampling as a method to get a sample size for validation of SCSS structural complexity attributes classification framework. Snowball technique is used to identify persons who are requested to identify other people who can potentially form part of the sample. This process continues until the

required or reasonable sample size is achieved. The sample size required is subjective meaning that it's determined by the researcher (Kumar, 2011). This is a helpful sampling method in the case where potential subjects are quite hard to find and you require referrals by other persons. Therefore, this study employed snowballing sampling method because, the SCSS language experts are few, and few experts were known to the researcher.

Convenience sampling was used to select subjects for experiments. According to Kumar (2011) in convenience sampling is based on the researchers ability to access the subjects. In this research, a number of subjects who had been trained on SCSS language volunteered themselves to participate in the experiments. A total of 21 subjects were involved in metrics tool validation while 30 subjects were involved for both subjective and objective experiments for validation of SCSS metrics. These numbers of subjects is acceptable as found in other similar research (Manso, Cruz-Lemus, Genero, & Piattini, 2008; Serrano, Calero, Trujillo, Luján-Mora, & Piattini, 2004; Genero, Manso, Visaggio, Canfora, & Piattini, 2007; Muketha et al., 2011; Bagheri & Gasevic, 2011).

3.6 Pilot Study

The questionnaire for the expert opinion survey was pretested by involving 3 SCSS experts. In this study only the persons with industrial experience of at least two years and had moderate level of knowledge for SCSS were considered as experts. Pretesting was carried out before the actual study to ensure validity and reliability of the instrument i.e. to ensure that the items tested what they were intended to and that they consistently measured the variables under study.

The pilot study for both subjective and objective experiment was carried out. The subjective part of the experiment was meant to determine whether there was any correlation between the metrics and subjects rating of understandability, modifiability, and testability. While the objective part of the experiment was carried to determine whether there was any correlation between the metrics and time to understand, time to modify and time to test. A small number of 10 students who were trained on SCSS language was involved in the pilot study. The subjects involved in pilot study were not involved in the final experimental study.

3.7 Data Collection Instruments

The data collection tool used in this study was a structured questionnaire for the expert opinion survey. The structured questionnaire was closed properly formatted with questions adopting a five-point Likert scale with a view to uniformed information (see Appendix 1). A questionnaire is a form used in survey design that participants in a study complete without intervention of the researchers collecting the data and return to the researcher (Wolf, 2009; Creswell, 2014; Babbie & Rubin, 2008).

In the case of validating metrics tool, a questionnaire was provided (see Appendix 7), and the subjects were required to indicate their observation on manual and automated computation of metrics. To validate the new metrics a questionnaire was provided, and subjects were required to fill it based on their observations. The subjects rated the provided SCSS files using a Likert scale in terms of their understandability, modifiability, and testability (see Appendix 2). In addition, the subjects recorded time

taken on working on the tasks under each section of understandability, modifiability, and testability.

3.8 Validity and Reliability

Validity and reliability of instruments used determines the credibility of the research results. Therefore, measures were taken to ensure that the data obtained is trustworthy.

3.8.1 Validity of the Research Instruments

The validity of research is the extent to which scientific research method requirements are followed. To ensure validity, the questionnaires were first scrutinized by the supervisors who gave their input and confirmed that the instruments met the criterion. In addition, to ensure validity, after the subjective part of experiment was conducted which is dependent on subjects opinion, an objective part of experiment was performed because its results are more reliable.

3.8.2 Reliability of the Research Instruments

Reliability is defined as the consistency of a test, survey, observation, or other measuring device and describes the extent to which instruments produce consistent results in similar conditions over time (Mohajan, 2017). To ensure the reliability of the instrument developed pretesting was carried out. The pretesting was done using Cronbach's alpha which is used as a measure of reliability. Cronbach's alpha (α) is the most common internal consistency measure and is normally interpreted as the mean of all possible split-half coefficients. It is a function of the average inter-correlations of items, and the number of items in the scale (Mohajan , 2017).

Table 3.1 shows the reliability statistics of subjective data questionnaire which had a Cronbach's alpha values of 0.976, while objective data questionnaire attained a value of 0.952. The Cronbach's alpha values were way above the recommended threshold value of 0.70 (Nunnally, 2008). The data collection instrument was therefore deemed to be reliable.

Table 3.1 Metrics Validation Reliability Statistics

Scale	Cronbach's Alpha
Subjective data questionnaire	0.976
Objective data questionnaire	0.952

3.9 Experimental Materials

The experimental materials that were used to facilitate data collection are metrics tool, and a set of SCSS files. The Metrics tool was used to compute metrics values for SCSS code. Installation of the tool was done in the computers in a laboratory for the subjects to use it and consequently answer a set of questions for validation of the tool. The metrics tool was installed in the researchers laptop to compute metrics values for SCSS files provided to the subjects. These SCSS files were obtained from existing real projects (websites) via the following link, <http://dmazinanian.me/publications/SANER'16/scss-websites.7z>. These SCSS files had first been gathered through a google search and put together in zipped folder which was downloaded in the link shown. This folder had 50 SCSS files and only 30 files which were randomly selected were used for experimental purposes. The metrics values collected in the SCSS files were then correlated with subjects rating of

understandability, modifiability, and testability and subjects understanding time, modifying time and testing time.

3.10 Data Analysis

The Statistical Package for Social Sciences (SPSS) version 19 was used for analysis of expert opinion survey results, metrics tool validation results, correlation of SCSS metrics with understandability, modifiability and testability and finally for ANOVA analysis. The R statistical tool version 3.6.0 for Windows was used for principle component analysis of the SCSS metrics.

3.10.1 Data Analysis Methods for the Expert Opinion Survey

The quantitative data collected was analyzed using descriptive statistics which included frequency, mean and standard deviation. The validation of SCSS attributes structural complexity classification complexity was done using descriptive statistics.

3.10.2 Data Analysis Methods for Tool Validation

The developed tool was validated using descriptive statistics, the mean time taken to compute metric values for each SCSS file both manually and using the tool was recorded. Then the means and standard deviation on data collected on a Likert scale of 1- 5 based on suitability, accuracy, and operability of the tool was calculated.

3.10.3 Data Analysis Methods for the Controlled Laboratory Experiment

In this research, descriptive statistics were presented in terms of frequency and percentages for the number of programming language students have taken, the software engineering courses pursued, and the level of SCSS knowledge. Correlation analysis was used to determine association of metrics values (independent variables)

and subjects rating of understandability, modifiability, and testability and the mean of time taken to understand, mean of time taken to modify the programs and mean of time taken to test the SCSS files (dependent variables). ANOVA tests were also conducted to establish whether the proposed metrics can actually determine understandability, modifiability and testability of SCSS code.

Multivariate analysis using principle component analysis (PCA) was used to model the contribution of SCSS metrics (independent variables) on dependent variables (understandability, modifiability and testability). When there are several metrics available in the software industry to measure software, a need arises to find the most significant metrics for better use and control of metrics (Saini, Sharma, & Singh, 2015). In this research the independent variables namely: Average Block Cognitive Complexity (ABCC_{SCSS}), Nesting Factor (NF_{SCSS}), Selector Use Inheritance Level (SUIL) and Coupling Level (CL_{SCSS}) were used to model their influence on the dependent variables namely: understandability, modifiability, and testability. The principle component is represented as follows:

$$Y_1 = \Phi^1 X^1 + \Phi^2 X^2 + \Phi^3 X^3 + \dots + \Phi^n X^n$$

Where:

- Y is the first principal component.
- $\Phi^1, \Phi^2 \dots \Phi^n$ are the loading vectors of principal component. In the first principle component the loadings or weights are constrained to a sum of square equals to 1.
- $X^1 \dots X^n$ are normalized predictors.

$$Y = \Phi^{11} * ABCC_{SCSS} + \Phi^{21} * NF_{SCSS} + \Phi^{31} * SUIL + \Phi^{41} * CL_{SCSS}$$

The Second principal component is also a linear combination of predictors which captures the remaining variance in the data set and is uncorrelated with the first component. The subsequent principle components capture the remaining variation without being correlated with the previous components.

There exists several rule of thumbs that determine the suitable cutoff. One of the most popular rule of thumb that has been agreed on by several researchers is that we can retain factors that account to about 70-80% of the variance (Rea & Rea, 2016; Rietveld & Van Hout, 2011). Therefore, this research selected components that cumulatively accounted for 80% of the model variation.

3.11 Ethical Issues

To ensure ethical principles in this research are followed, the researcher sought for an introductory letter from Masinde Muliro University of Science and Technology, Board of Postgraduate Studies (see Appendix 8), then consent from National Council of Science and Technology (NACOSTI) was received (see Appendix 10). A letter of permission was obtained from Murang'a University of Technology where the data collection took place (see Appendix 11). The researcher trained the subjects on SCSS language, who later participated in experiments as subjects. The participation in training and involvement in experiments was conducted on a voluntary basis. The researcher assured the subjects that the information obtained from them was to be treated as confidential.

3.12 Chapter Summary

This chapter described the research process, philosophy, design, strategy, sampling technique, research instruments, data analysis techniques, and research ethics. The research process described a four-step process that was followed to achieve the objectives of this study, research philosophy for this study was positivist in nature, the research design was explanatory, while research strategy used was surveys and experiments. The sampling technique that was used for objective one was the snowball method and for objective three and four, convenience sampling method was employed. The research instrument used for attributes classification framework, metrics tool validation and SCSS metrics validation were questionnaires (see Appendix 1, Appendix 2 and Appendix 7) which were found to be reliable and valid. In addition, the metrics tool and a set of SCSS files were identified as the experimental materials. The analysis of data was achieved through descriptive and inferential statistics. Finally, the research complied with all necessary research ethics and all relevant authorities and institutions were notified of the research prior to conducting expert opinion survey and experimental work.

CHAPTER FOUR

DEVELOPMENT OF STRUCTURAL COMPLEXITY ATTRIBUTE CLASSIFICATION FRAMEWORK FOR SASSY CASCADING STYLE SHEETS (SCACF-SCSS)

4.1 Introduction

This chapter presents the structural complexity attributes classification framework for SCSS, to aid in the determination of SCSS structural complexity measurement attributes. The requirements for the development of the framework, architecture of the framework, application of the framework and expert opinion validation results have been presented.

4.2 Requirements of the SCACF-SCSS Framework

The requirements of the framework are as follows:

- The framework should identify all the structural complexity causing attributes
- The framework should categorize all the identified attributes
- The users should be able to identify SCSS features and use the framework to place it in the right category.

4.3 Architecture of the Proposed Framework

A detailed explanation on the various branches of SCSS structural complexity framework, which are intra-module, inter-module, hybrid and the extra-module attribute is provided. The highlighted areas in Figure 4.7 indicate the extension to Muketha's Framework.

4.3.1 Intra-Module Attribute

The intra-module attributes focus on attributes that can be derived from a rule-block; these attributes don't interact with other rule blocks. A rule-block is equivalent to a module. In SCSS two categories of attributes were identified, size and control-flow complexity.

When the size of a code increases, its complexity increases (Muketha et al., 2010b; Adewumi et al., 2012; Misra and Cafer, 2012; Khan et al., 2016). In addition, use of operators increases the size of code thus complexity increases (Misra & Cafer, 2012). The size of SCSS is contributed to by the number of declarations, number of operators and number of rule blocks. SCSS declarations refer to all the statements terminated with a semicolon, the operators are the mathematical symbols such as plus, minus, division, multiplication and equal sign, while the rule blocks refer to a block of code with an opening brace “{“ and a closing brace “}”.

In SCSS every statement terminating with a semicolon is counted as a declaration or attribute. Rule-block *a* has 4 declarations and rule-block *b* has 2 declarations, meaning that the total number of declarations or attributes as shown in Figure 4.1 are 6. On the other hand, the total number of rule-blocks in the figure are 2, namely rule-block *a* and *b*.

```
a{
    -----;
    -----;
    -----;
    -----;
}
b{
    -----;
    -----;
}
```

Figure 4.1: SCSS Size

Control flow of any software artifact increases the complexity of the code. The different control-flows are assigned weights, for example, *for* statement is assigned higher weight than *if* statement, meaning that a *for* statement contributes to a higher cognitive complexity as compared to *if* statement (McCABE, 1976; Cardoso, 2006; Muketha et al., 2010b; Misra & Cafer, 2012). SCSS code implements control directives such as `@for`, `@if`, and `@each`, meaning that the control flow complexity of SCSS code should be determined.

An illustration of the control flow complexity is shown in Figure 4.2, for SCSS code and it has 2 *if* directives and 1 *for* directive.

<pre> if { -----; if { -----; -----; } } </pre>	<pre> for \$x from 1 through n { a{ -----; } { -----; } } </pre>
---	--

a)Branch

b)Loop

Figure 4.2: Control-flows in SCSS

4.3.2 Inter-Module Attribute

The Inter-module attribute of SCSS focuses on the interaction of the various rule-blocks. In the proposed framework, inter-module has been divided into inheritance complexity and nesting complexity categories.

Inheritance feature in software's has been widely studied and proved to contribute to the complexity of software products (Chawla & Nath, 2013; Misra et al., 2011). Inheritance complexity in SCSS is evidenced when the styles or values are shared by using extend directive and is known as selector inheritance. Figure 4.3 illustrates inheritance complexity, where the *b* selector inherits from *a* selector by use of *@extend a* statement.

```
a {
    -----;
    -----;
    -----;
}

b {
    @extend a;
}
```

Figure 4.3: Inheritance in SCSS

Nesting feature contributes to software complexity (Li, 1987; Chhillar & Bhasin, 2011; Frain, 2013). SCSS language implements nesting of rules. where the rules are placed inside other rules. Therefore, nesting complexity is presented in the framework. SCSS permits nesting of rules, for instance, In Figure 4.4 the *b* selector is placed inside *a* selector and *c* is placed inside *b* selector.

```
a {
    -----;
    -----;
    -----;
    b {
        -----;
        -----;
        c {
            -----;
            -----;
        }
    }
}
```

Figure 4.4: Nesting in SCSS

4.3.3 Hybrid Attribute

The hybrid attribute combines features of at least two categories of structural complexity, for example, intra-module and inter-module (Muketha, 2011). In SCSS the hybrid attribute has one category falling under it referred to as association complexity. This kind of complexity is brought about by the different features which are found in different categories of SCSS structural complexity being implemented in a single rule block. For example, the cognitive block complexity is as a result of mixin calls which are found in extra-module attribute category, extend directives which are in the inter-module category, number of declarations, number of operators, number of rule blocks and control directives which fall under intra-module attribute category. The convergence of the intra-module, inter-module, and extra-module attributes led to the hybrid attribute category.

SCSS association of different attributes category is illustrated in Figure 4.5. The *a* rule block makes use of global variables and mixins (include directive) which fall in extra-module category attribute. An extend directive is also used in rule block *a* and falls under the inter-module category. This implies there is a convergence of extra-module and inter-module category.

```
Variable 1;  
Mixin 1;  
a{  
    Use of global variables  
    Use of Include directive  
    Use of extend directive  
}
```

Figure 4.5: Association in SCSS

4.3.4 Extra-Module Attribute

A newly added attribute called Extra-module focuses on the interaction of rule-blocks via an external module, meaning that the rule-blocks are coupled to each other indirectly. In SCSS the Extra-module attribute focuses on rule-blocks interacting with mixins and/or global variables. These mixins and global variables are defined outside of SCSS rule blocks. When there are several rule blocks sharing the same mixin and global variable, then the rule blocks are deemed to be coupled with each other. This implies that a change in the values of a mixin and a variable will affect all the rule blocks that are sharing the mixin and global variable.

SCSS allows information to flow from one rule block to another as illustrated in Figure 4.6. The information flow occurs when rule blocks share styles and values from variables and mixins. The rule blocks *a* and *b* are sharing from the same set of variables and mixins.

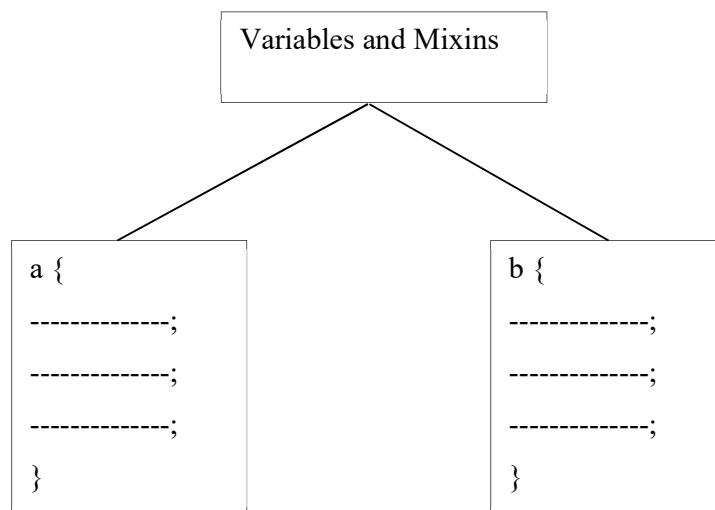


Figure 4.6: Information Flow in SCSS

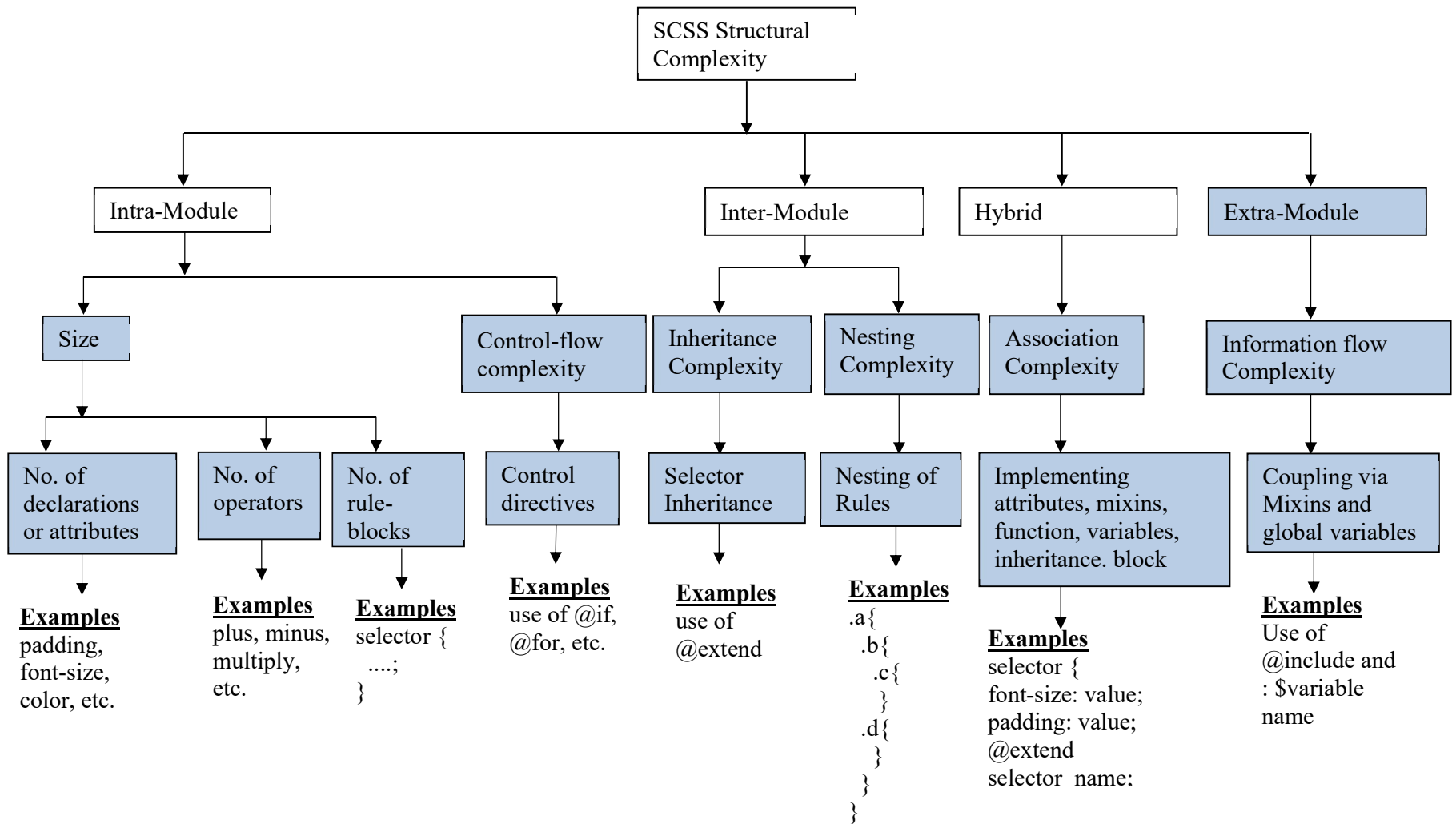


Figure 4.7: Structural Complexity Attribute Classification Framework for SCSS (SCACF-SCSS)

4.4 Application of the Framework

The aim of this section is to provide an interpretation of the proposed framework through real-life scenarios.

4.4.1 Intra-Module Attribute

The intra-module attribute is the first category of the SCSS structural complexity, and it considers complexity in terms of size and control flow complexity. The size of the SCSS file can be determined based on the number of attributes, number of operators or number of rule blocks.

To determine the size of the SCSS code in Figure 4.8 in terms of number of declarations count all the statements ending with a semicolon (;), to count size from the perspective of rule blocks, count all the rule blocks, where each rule block is recognized by an opening brace “{“ and a closing brace “}” and finally to count size in terms of operators, count all the operators i.e plus “+”, minus “-“, multiplication “*”, division “/” and equal “=” signs.

The returned results are: number of declarations is 5, i.e. (font-size, color, display, border-bottom and another font-size declaration), the number of operators is only 1, i.e the Plus symbol (+), and the number of rule blocks is equal to 2 i.e (p{ } and li{ }).

```
p{
    font-size: 5px;
    color:#ff0000;
}
li {
    display: block;
    border-bottom: 1px solid;
    font-size: 1.6rem + 2;
}
```

Figure 4.8: Size Complexity Scenario

The control flow complexity of SCSS code is determined by the control directives implemented in the code. In the SCSS code provided in Figure 4.9, the *@if* directive has been implemented, meaning that the measurement for the control flow complexity can be determined.

```
$colortest: 1;
p {
    font-size: 5px + (6px * 2);
    color:#ff0000;
    @if $colortest >1 {
        text-color: blue;
        @if $colortest == 1 {
            text-color: white; } }
}
```

Figure 4.9: Control Flow Complexity Scenario

4.4.2 Inter-Module Attribute

The inter-module attribute category describes the inheritance and nesting complexity. Inheritance complexity in SCSS is introduced by the use of *@extend* directive. In Figure 4.10, the extend directive has been used in the *h2* element selector to inherit *p* element selector.

```
p{
  font-size: 5px;
  color:#ff0000;
}

h2{
  @extend p;
  font-color:$color1;
}
```

Figure 4.10: Inheritance Complexity Scenario

One of the unique feature of SCSS is nesting, where a rule block is placed inside another rule block. For instance, the *ul* selector is placed inside *js-offcanvas* class selector and *li* is place inside *ul* selector as shown in Figure 4.11.

```
.js-offcanvas {
  color: $color1;
  background: $color2;
  ul {
    padding-left: 0;
    margin-bottom: 0;
    li {
      display: block;
      border-bottom: 1px solid;
      font-size: 1.6rem;
    }
  }
}
```

Figure 4.11 Nesting complexity Scenario

4.4.3 Hybrid Attribute

In the hybrid attribute category, a form of complexity known as association complexity is identified. In the SCSS code provided in Figure 4.12. The *h1* rule block makes use of a global variable. *\$color1* and *include* statement to make use of mixin block *Raleway-SemiBold* (they fall under extra-module attribute). An *extend* statement is also used in *h1* rule block and it falls under the inter-module category.

```
. $color1: #04f5f7;
@mixin Raleway-SemiBold {
  font-family: 'Raleway-SemiBold';
}

p{
  font-size: 5px;
  color:#ff0000;
}

h1 {
  font-color:$color1;
  @include Raleway-SemiBold;
  @extend p;
}
}
```

Figure 4.12: Association complexity Scenario

4.4.4 Extra-Module Attribute

The final category known as the extra-module category is illustrated. The information flow complexity which is a result of coupling is demonstrated in the SCSS code provided in Figure 4.13. The SCSS code has one defined variable *\$color1* and one defined mixin *Raleway-Medium*. The variable and mixin are shared by *p* and *span* rule blocks. This sharing of the same variables and mixin brings about coupling.

```
$color1: #04f5f7;

@mixin Raleway-Medium {
  font-family: 'Raleway-Medium';
}

p {
  font-size: 5px + (6px * 2);
  font-color: $color1;
  @include Raleway-Medium;
}

span{
  width: 60px;
  height: 45px;
  color: $color1;
  position: absolute;
  @include Raleway-Medium;
}
```

Figure 4.13: Information Flow Complexity Scenario

4.5 Expert Opinion Validation Survey

This section presents the evaluation results obtained from an expert opinion survey. An expert opinion survey technique is used to identify problems, give clarity to issues under study and evaluate products (Whitfield, 2008).

4.5.1 Goal of the Study

The goal of the study was to evaluate the relevance and comprehensiveness of the framework from the point of view of SCSS experts.

4.5.2 Context Definition

SCSS experts who have an online presence were invited to participate in the survey. The SurveyMonkey platform was used to host the study questionnaires. A total of 13 experts participated in the survey and were identified through snowball sampling technique. The researcher stopped at 13 experts and was considered as sufficient because SCSS experts are hard to find. Therefore, the researcher believed that this forms a good saturation point (Naderifar, Goli & Ghaljaie, 2017; Kumar, 2011).

4.5.3 Survey Operation

The respondents were provided with the SCSS attributes classification framework, a write-up explaining how to interpret the framework and a survey questionnaire.

4.5.4 Reliability of the Research Instrument

Reliability of the questionnaire was conducted on the relevance and comprehensiveness of the framework to ensure consistent results are achievable with different persons using the same instrument. As shown in Table 4.1, relevance achieved a Cronbach alpha of 0.894 while comprehensiveness achieved a Cronbach alpha of 0.854. Therefore, the instrument can be considered reliable since its reliability values exceeded the prescribed threshold of 0.7 (Nunnally, 2008).

Table 4.1. Framework Reliability Statistics

Scale	Cronbach's Alpha
Relevance of the Framework	0.894
Comprehensiveness of the Framework	0.854

4.5.5 Results

Feedback from the respondents was received and thereafter checked for completeness. All questionnaires were found to be completed satisfactorily, and therefore were accepted for data analysis.

4.5.5.1 Respondents Demographics

The researchers first sought to establish the characteristics of the respondents, and so characteristics such as the level of education, years of industrial experience, level of knowledge for software engineering processes and level of knowledge of SCSS was considered from all respondents.

4.5.5.2 Level of Education for Respondents

Respondents were asked to state their education background. Results indicate that 11 (84.6%) of the respondents are bachelor's degree holders while the remaining 2 (15.4%) respondents have master's degree qualifications. These results imply that all the SCSS experts involved in this study have attained at least the bachelor's degree, implying that they have the capability to study the framework and respond accordingly. These findings are shown in Table 4.2.

Table 4.2: Level of Education for Respondents

Level of Education	Frequency	Percent (%)
Bachelors	11	84.6
Masters	2	15.4

4.5.5.3 Years of Industrial Experience

This research sought to find the number of years the respondents have worked in the industry. It was observed that 2 of the respondents had an experience of between 2-3 amounting to 15.4% while the rest of the respondents had 4 years of experience or higher. This implies that the respondents in this study are highly experienced in the software engineering field and can be considered as experts.

Table 4.3. Years of Industrial Experience

Years of Industrial Experience	Frequency	Percent (%)
2-3 Years	2	15.4
4-5 Years	6	46.2
6-7 Years	2	15.4
Above 7 Years	3	23.1

4.5.5.4 Level of Knowledge in Software Engineering Processes

An analysis of the respondent's level of knowledge was also conducted as indicated in Table 4.4. Findings indicate that 12 respondents representing 92.3% had a high level of knowledge while 1 respondent representing 7.7% had a very high knowledge of software

engineering processes. These findings imply that all participants can be trusted for analysis and opinions on the state of artifacts that are intended for use in the software engineering process.

Table 4.4: Level of Knowledge for Software Engineering Processes

Level of Knowledge for Software Engineering Processes	Frequency	Percent (%)
High	12	92.3
Very High	1	7.7

4.5.5.5 Level of Knowledge for SCSS

Since the proposed framework focuses only on the structural complexity of code developed using the SCSS language, all respondents are expected to be knowledgeable SCSS programmers. Findings indicate that 8 respondents had a high level of knowledge and this corresponding to 61.5%, 3 respondents corresponding to 23.1% had a moderate level of knowledge, and 2 respondents corresponding to 15.4% had a very High level of knowledge. This implies that the data collected from all the respondents can be deemed as valid. The respondents result with a moderate level of knowledge are also acceptable because they can be regarded as having a considerable level of SCSS knowledge in addition to their software engineering knowledge, which is acceptable for the purposes of this study. These findings are shown in Table 4.5.

Table 4.5: Level of Knowledge for SCSS

Level of knowledge for SCSS	Frequency	Percent (%)
Moderate	3	23.1
High	8	61.5
Very High	2	15.4

4.5.5.6 Relevance of the Framework

The researchers sought to know if the developed framework is relevant for the industry experts to identify the attributes that lead to SCSS complexity. Table 4.6. shows computed means from a Likert scale of 1 to 5 – Don't Agree, Slightly Agree, Agree, Strongly Agree and Very Strongly Agree. Findings show that the respondents agree that there is a great need for a classification framework with a mean of 3.46, which falls between agree and very strongly agree (i.e. between 3 and 4 in the Likert scale). The respondents also agree that the framework is useful for the process of identification of SCSS attributes as indicated by the mean of 3.62, these findings are shown in Table 4.6. Standard deviation was interpreted as low if the value is less than or equal to 1, while values greater than 1 are high. When the value is low it implies that the respondents didn't differ much in their opinion and high values indicate respondents considerably differed in their opinion. The standard deviation values shown in Table 4.6 indicates that the respondents didn't vary considerably.

Table 4.6: Relevance of the Framework

	Need for the Framework	Usefulness of the Framework
Mean	3.46	3.62
Standard Deviation	0.776	0.870

4.5.5.7 Comprehensiveness of the Framework

In a Likert scale, respondents were asked of their opinions on whether the proposed framework is comprehensive or not. Findings showed that global variables and declarations least contribute to SCSS complexity with a mean of 2.54 and 2.85 respectively. These values fall within the range of slightly agree and agree (i.e. between 2 and 3 in the Likert scale). This implies that SCSS programmers somehow agree that the two features cause complexity in SCSS and should not be overlooked. Findings also show that all other remaining features fall in the range of agree and strongly agree (i.e. between 3 and 4 in the Likert scale). These mean values imply that the respondents agree that the concerned features contribute to SCSS complexity. The standard deviation values are high, but this is a result of the small sample size. Sullivan (2015) argued that the standard deviation of the means decreases as the sample size increases. Therefore, the high standard deviation can be explained and doesn't make the results unreliable. These results are shown in Table 4.7.

Table 4.7: Comprehensiveness of the Framework

SCSS features	Mean	Standard Deviation
Global Variables	2.54	1.127
Declaration	2.85	1.214
Operator	3.00	1.000
Control Directives	3.31	1.032
Function	3.54	1.050
Mixins	3.38	1.193
Extends	3.15	1.519
Nesting	3.46	1.561

Finally, respondents were asked whether they agree that the SCSS features identified in Table 4.7. wholly represents all the possible features that need to be considered when analyzing the complexity of code written in SCSS language. Findings show that 12 respondents agree corresponding to 92.3% while 1 respondent corresponding to 7.7% disagree. The findings, shown in Table 4.8, imply that the proposed framework is adequate as an indicator of features that cause structural complexity in SCSS code.

Table 4.8: Adequacy of SCSS Complexity Features

Adequate Features	Frequency	Percent (%)
Yes	12	92.3
No	1	7.7

4.6 Chapter Summary

In this chapter, a new SCSS structural complexity attribute classification framework was proposed. The framework extended Muketha’s classification framework as it was found

to be the most closely related framework to this study. A high-level category of attribute referred to as extra-module attribute was added and more lower levels were identified and added in all high level categories of SCSS attributes.

The proposed framework was validated through an expert's opinion survey. The experts agreed overwhelmingly that the framework is relevant and comprehensive. This means that the framework can be relied on as a formal approach of identifying all SCSS structural complexity attributes that can then be used as the basis of defining SCSS metrics.

CHAPTER FIVE

STRUCTURAL COMPLEXITY METRICS FOR SASSY CASCADING STYLE SHEETS

5.1 Introduction

This chapter proposes a set of metrics to measure SCSS code complexity. The chapter was intended to solve the second research objective as described in the first chapter, that is, defining a suite of theoretically sound metrics for measuring the structural properties of SCSS.

5.2 Determination of Attributes to be Measured

SCSS structural complexity attributes framework was employed to identify measurable attributes for SCSS language. Four types of attributes were identified including, intra-and inter-module, hybrid, and extra-module attribute. The intra-module attribute identified base metrics which were then used to derive other metrics found in other types of attributes. For example number of rule blocks was implemented in all the derived metrics of Nesting Factor for SCSS, Selector use Inheritance Level, Average Block Cognitive complexity for SCSS and Coupling Level for SCSS.

The identified measurement attributes were classified as follows:

- a) Inter-module attribute
 - Nesting Factor for SCSS (NF_{scss})
 - Selector use Inheritance Level (SUIL)
- b) Hybrid attributes
 - Average Block Cognitive complexity for SCSS (ABCC_{scss})

- c) Extra-module attributes
 - Coupling Level for SCSS (CL_{SCSS})

5.3 Metrics Definition

The proposed metrics are derived from existing CSS metrics and other software metrics through the process of modification. This study followed the Entity-Attribute-Metric model in the definition of metrics for SCSS (Fenton and Pfleeger, 1997), where the entity is SCSS code, attributes identified to be measured from SCSS code were cognitive complexity of SCSS blocks, nesting level for SCSS code, selector inheritance level for SCSS code and coupling level of SCSS code.

The EAM model was extended to EAMT meaning Entity Attribute Metrics Tool model, this was after review of literature and many researchers over the years agree that metrics tool development is a necessary step for metrics to be acceptable by the software industry (Littlefair, 2001; Spinellis, 2005; Linos et. al, 2007; Lincke et al., 2008; Muketha, 2011; Adewumi et al, 2015; Misra et al.,2018). Figure 5.1 illustrates the newly extended EAMT model.

The introduction of the new EAMT model will enforce the development of metrics tool and mainstream tooling as a requirement in the process of definition of metrics.

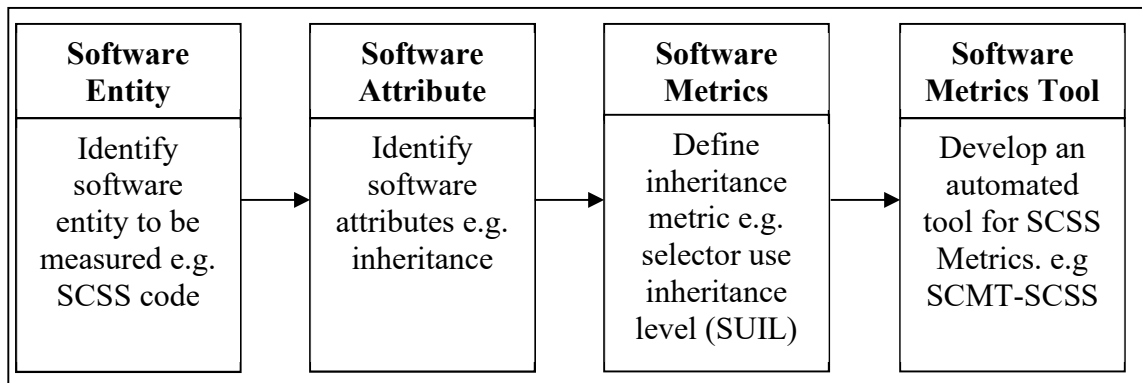


Figure 5.1: EAMT Model

The SCSS blocks are the fundamental building units for SCSS code. The formal definition of a SCSS block SCSSB is $SCSSB = \langle A, D \rangle$

An SCSS block (SCSSB) is a 2-tuple $\langle A, D \rangle$, where A is the set of attributes, and D is the set of directives such as mixin directives, control directive, function directive, and media directives.

A suite of four metrics were defined namely; $ABCC_{scss}$, NF_{scss} , SUIL and NF_{scss} . To prove the intuitionality of the metrics, metric values were computed using code snippets and three real world projects. The code snippets were written by the researcher while the code for projects was obtained by using google advanced search feature and files with .scss extension were identified and downloaded from www.happy-shala.com, www.greatjewishmusic.com and www.mce.ie.

5.3.1 Average Block Cognitive Complexity for SCSS (ABCC_{scss})

The metric ABCC_{scss} extends Number of Attributes Defined per Rule Block (NADRB) and is used to compute the complexity of a rule block in regular CSS. NADRB metric calculates complexity by determining the average number of attributes defined in the rule blocks. The proposed ABCC_{scss} metric will consider other factors beyond the number of attributes, such as @rule and directives, operators, function calls, and variables.

Researchers have in the past identified several factors that they claim contribute to complexity. Those factors that relate to SCSS were considered in this research and they are as discussed in the subsequent paragraphs.

The number of regular attributes (NRA) was considered in the stylesheets field, as a factor that contributes to CSS complexity. According to Adewumi, et al., (2012), the more the number of attributes in a rule block the more complex the rule block becomes.

The number of operators (NO) has been recognized by several researchers as a factor that contributes to the complexity of code. For example, Misra & Cafer (2012), in their definition of JavaScript Cognitive Complexity Metric (JCCM) included the number of operators in this metric. In addition; Halstead (1977) in the development of Halstead science theory, posited that the complexity of software is due to operators and operands.

The consideration of control flows in terms of their contribution to the complexity of code cannot be ignored, this is supported by several studies (Muketha, et al., 2010; Misra and

Cafer, 2012; McCABE, 1976). In rule blocks, the use of control directives is assigned weights as shown in Table 5.1. The weights allocates a value of 1.3 for a branch and 1.5 for a loop (Törn et al.,1999). In SCSS the control flows have been categorized into two, that is, branch statements and loop statements. The number of branch statements (NB) and the number of looping statements (NL) are counted, while at the same time considering their weights.

The consideration of function calls as an aspect that contributes to code complexity is supported by several studies (Misra and Cafer, 2012; Shao and Wang, 2003). SCSS allows the definition of functions and in effect, function calls are introduced. This research introduced number of function calls (NFC) metric as an SCSS complexity contributing factor. Function calls are similar to branches and therefore a weight of 1.3 was allocated to them. This conformarms with the way Misra and Cafer (2012) assigned selection or branch statements and function calls with the same weight.

Mixins are blocks of codes that are defined and can be included in various parts of SCSS code by use of the `@include` statement. The number of mixin calls (NMC), just like function calls increases complexity because the control of the program is dependent on the mixin calls. These mixins are called from different places in the code. The `@include` directive rule is weighted at 1.3 same as the function calls.

The number of extend directives (NE) are considered as one of the contributors to rule-block complexity. This rule directive inherits a selector, meaning that code complexity

increases when it's implemented. @extend directive rule is weighted at 1.3, just like function calls because some code in a different place is being referred.

Table 5.1: Weights for Basic Control Structures

Type of directive	Statements	Cognitive weight
Branch	@if , @else if , if () and function calls,mixin calls, use of extends	1.3
Loop	@for, @while and @each	1.5

To calculate $ABCC_{SCSS}$, the complexity of each SCSS block is computed herein referred to as Block Cognitive Complexity (BCC). The sum of complexity of all SCSS blocks is computed and is represented by the Total Block Cognitive Complexity metric (TBCC). TBCC is then divided by the number of all SCSS blocks (NOBL). NOBL is a simple size metric that counts all the blocks used in SCSS.

$$TBCC = \sum_{i=1}^n BCC_i \dots\dots\dots(i)$$

Where n is the total number of SCSS blocks and

$$BCC = NRA + NO + (NB * 1.3) + (NL * 1.5) + (NFC * 1.3) + (NMC * 1.3) + (NE * 1.3)$$

$$ABCC_{SCSS} = TBCC / NOBL \dots\dots\dots(ii)$$

The demonstration of the computation of $ABCC_{SCSS}$ metric is done using the example in Figure 5.2. The number of regular attributes is 9, number of operators is 1, number of branch statements is 0, number of loop statements is 1, number of function calls is 0,

number of mixin calls is 1 and number of extend directives is 0. The total number of SCSS blocks is 4. Therefore;

$$\begin{aligned} \text{TBCC} &= 9 + 1 + 0 + 1 * 1.5 + 0 + 1 * 1.3 + 0 \\ &= 12.8 \end{aligned}$$

$$\text{NOBL} = 4$$

$$\begin{aligned} \text{ABCC}_{\text{SCSS}} &= 12.8 / 4 \\ &= 3.2 \end{aligned}$$

```
$color1: #f4f4f4;
$color2: #000;
@mixin fonts {
  font-color: #ff21a3;
  font-family: sans-serif;
  font-size: 12px; }
p {
  @include fonts;
  font-weight: bold; }
span{
  width: 60px;
  height: 45px;
  position: absolute; }
@for $i from 1 through 4 {
  .p#{$i} { padding-left : $i * 10px; }
}
```

Figure 5.2: ABCC_{SCSS} Metric Example

The $ABCC_{SCSS}$ metric values obtained from the websites are 2.58 for happy-shala.com, 2.17 for greatjewishmusic.com and 2.9 for mce.ie.

5.3.2 Nesting Factor for SCSS (NF_{SCSS})

Nesting refers to the use of constructs such as if, while, for and each are found within other constructs and it increases program complexity (Li, 1987). SCSS allows nesting of CSS rules inside each other instead of repeating selectors in a separate declaration (Cederholm, 2013). According to Frain (2013), the nesting of rules should be kept as shallow as possible otherwise, it reduces the maintainability of the code. This means the higher the nesting level the more complex a program.

Regular CSS doesn't have nesting feature, therefore nesting concept in SCSS is borrowed from structured programming languages and object-oriented programming (OOP) languages. However, nesting in SCSS has an extra component as compared to other languages. In the regular programming languages when defining metrics only nesting depth is normally considered, while in SCSS we should consider nesting depth and nesting breadth. Figure 5.3 demonstrates nesting depth where we have *countries-list* rule block inside *header* rule block and *li* rule block inside *countries-list* rule block.

```
header{
width: 90%;
position: absolute;
height: 97px;
  .countries-list{
    position: absolute;
    top: 100px;
      li{
        display: block;
        margin-bottom: 5px;
      }
    }
  }
```

Figure 5.3: Nesting Depth

Nesting breadth occurs when there are independent rule blocks inside a single rule block. Meaning that we consider the rule blocks which are independent of each other but dependent on a single rule block also known as parent block. For example, in Figure 5.4 the *countries-list* rule block and the *li* rule block are two independent rule blocks inside *header* rule block. The two blocks *countries-list* and *li* rule blocks have no any relationship with each other, only that they share the features of the *header* rule block. However, the nesting breadth is not considered with the control directives of SCSS, since all the nested blocks have a relationship with each other.


```
header{
width: 90%;
position: absolute;
height: 97px;
    .countries-list{
        position: absolute;
        top: 100px;
    }
    li{
        display: block;
        margin-bottom: 5px;
    }
}
```

Figure 5.4: Nesting Breadth

In the computation of the nesting depth, a metric value of 1 is assigned to the first level, a value of 2 to the second level, a value of 3 to the third level and so on (Chhilar and Bhasin, 2011). A nesting depth of 3 means we have three levels of nesting, meaning the depth cognitive complexity (DCC) value is $3+2+1=6$ and if it's a nesting depth of 5 then DCC value will be $5+4+3+2+1=15$. The calculation of nesting breadth simply counts the number of SCSS blocks inside a single SCSS block. Therefore, if there are two independent rule blocks in a single block, then the complexity is assigned as 2.

The proposed metric NF_{SCSS} computes the nesting level by considering the total depth nesting level (TDNL) and the total breadth nesting level (TBNL) of all SCSS blocks.

$$TDNL = \sum_{k=1}^n DCC_k \dots\dots\dots(iii)$$

Where n = number of SCSS blocks

$$DCC = \sum_{i=0}^{m-1} (m - i)$$

Where m is the nesting depth

$$TBNL = \text{number of independent blocks in different single rule blocks} \dots\dots\dots(iv)$$

$$NF_{SCSS} = TDNL * TBNL \dots\dots\dots(v)$$

The demonstration of the computation of NF_{SCSS} metric was done using the example in Figure 5.5. The SCSS code provided has a *header* rule block with *countries-list* rule block placed inside it and the *li* rule block is placed inside *countries-list*. The *p* rule block was not considered in calculating nesting depth because it's in the same level as *countries-list* rule block. This means that the nesting depth is 2. Therefore, $TDNL = 2+1=3$.

The SCSS code provided in Figure 5.5 has *p* rule block which is independent of *countries-list* rule block and *li* rule block, but is dependent on the *header* rule block, because its placed inside *header* rule block. The *countries-list* rule block is dependent on *header* rule block and is in the same level as *p* rule block. Therefore, the *countries-list* rule block and the *p* rule block form part of nesting breadth, meaning that $TBNL = 2$.

The derived metric NF_{SCSS} is computed

$$NF_{SCSS} = 2 * 2 = 4$$

```

header{
width: 90%;
position: absolute;
height: 97px;
    .countries-list{
        position: absolute;
        top: 100px;
            li{
                display: block;
                margin-bottom: 5px;
            }
        }
    }

p {
    @include fonts;
    font-weight: bold;
}
}

```

Figure 5.5: NF_{SCSS} Metric Example

The metric values computed for NF_{SCSS} metric obtained a value of 6960 for happy-shala.com, 8019 for greatjewishmusic.com and 3034 for mce.ie websites.

5.3.3 Selector Use Inheritance Level (SUIL)

This metric measure complexity brought about by inheriting selectors in SCSS. Though there is form of inheritance in the regular CSS, it doesn't allow inheritance of selectors.

The inheritance concept in SCSS is borrowed from the object-oriented software. Therefore, the class inheritance factor (CIF) metric (Vinobha, Velan & Babu, 2014) in OOP domain motivated the definition of SUIL metric for SCSS.

The proposed SUIL modifies the CIF metric and is calculated by taking the sum of all inherited selectors which is divided by the total number of all selectors.

$$\text{SUIL} = \frac{\sum_{i=1}^n \text{NSI}}{\sum_{i=1}^n \text{NS}} \dots\dots\dots(\text{vi})$$

Where NSI is the Number of all selector inheritance instances and NS is the Number of all selectors in the program and n is the number of SCSS blocks

The demonstration of the computation of SUIL metric was done using the example in Figure 5.6. The number of selector inheritance instances is the number of @extend directives in the rule blocks while the number of selectors is the total number of SCSS blocks excluding the mixin blocks and control-flow blocks, media and function blocks.

```
$color1
p {
  font-type:italic;
  text-transform: uppercase; }
h1 {
  @extend p; }
h2{
  @extend p;
  font-color: $color1; }
```

Figure 5.6: SUIL Metric Example

As shown in Figure 5.6, the selector inheritances are in h1 and h2 rule blocks and are 2 in number. The total number of selectors is 3. Therefore, $SUIL=2 / 3 = 0.67$.

The metric values obtained for SUIL metric from the websites were 0 for happyshala.com, 0 for greatjewishmusic.com and 0.03 for mce.ie.

5.3.4 Coupling Level for SCSS (CL_{SCSS}) metric

Coupling is the measure of the strength of association established by a connection from one class to another (Stevens et al., 1974; Chidamber and Kemerer, 1994). In OOP, coupling occurs when methods of one class use methods or variables of another class. In SCSS, coupling occurs when rule blocks share mixins and variables. The more the rule blocks sharing the same mixin or variable, the higher the coupling level.

A need for a new metric for measuring coupling level in SCSS arises. The CL_{SCSS} metric is proposed and it's computed by summing the number of all declared mixins (NDM) with the number of all declared variables (NDV) which is then divided by the summation of all the number of mixin calls (NMC) and total number of all variable instances (NVI) in the program.

$$CL_{SCSS} = (NDM+NDV) / (\sum_{i=1}^n NMC + \sum_{i=1}^n NVI) \dots\dots\dots(vii)$$

where, n is the number of SCSS blocks in the program

In the SCSS code example in Figure 5.7, there is only 1 mixin declared (@mixin fonts) and 1 declared variable (\$color1). The total number of mixin calls are 3. i.e where we have all @include statements, and the total number of variable instances are 2. Therefore,

$$CL_{Scss} = (1 + 1) / (3 + 2)$$

$$= 2 / 5 = 0.40$$

```
$color1: #f4f4f4;
@mixin fonts {
  font-color: #ff21a3;
  font-family: sans-serif;
  font-size: 12px;
}
p {
  font-type:italic;
  @include fonts;
}
h1 {
  font-color: $color1;
}
h2 {
  @include fonts;
  text-transform: uppercase;
}
h3 {
  @include fonts;
}
h4 {
  background: $color1;
}
```

Figure 5.7: CL Metric Example

The CL_{SCSS} metrics results obtained after analysis of the three websites were 0.31 for happy-shala.com, 0.27 for greatjewishmusic.com and 2.33 for mce.ie.

5.4 Theoretical Validation Results for the Proposed Metrics

Two methods were used, namely, Weyuker's properties to establish the theoretical soundness of the metrics and the Kaner framework to prove the practical value of the metrics.

The software community fully accepts software metrics when they have sound theoretical and mathematical foundation. Therefore, the proposed metrics have been validated using Weyuker's properties and Kaner framework. Weyuker's properties have been used by several researchers to evaluate their proposed software metrics and they agree to the fact that it's a necessary framework and that for a measure to be valid it must satisfy most of its properties (Cherniavsky and Smith,1991; Abreu and Carapuca,1994; Chidamber and Kemerer,1994; Gursaran,2001; Sharma et al., 2006; Muketha et al., 2010a; Basci and Misra, 2011b). The Kaner framework has been used by a number of researchers (Adewumi et al., 2012; Basci and Misra, 2011b), and has been applied in this research for practical evaluation of the proposed metrics.

5.4.1 Validation with Weyuker's Properties

Property 1: $(\exists P) (\exists Q) (|P| \neq |Q|)$ where P and Q are two different SCSS blocks.

This property is satisfied when there exist SCSS blocks P and Q such that $|P|$ is not equal to $|Q|$. Therefore, if we can't find two SCSS blocks of different complexity, then all SCSS blocks have the same complexity value. All the metrics proposed $ABCC_{SCSS}$, NF_{SCSS} ,

SUIL and CL_{SCSS} , return different complexity value for any two SCSS blocks that are not identical and therefore they satisfied this property.

Property 2: Let c be a non-negative number.

Then there are finitely many SCSS blocks of complexity c . This property asserts that if an SCSS block changes then its complexity changes. When the number of attributes is changed, complexity values change for the $ABCC_{SCSS}$. In addition, when the number of extend rule directives changes then SUIL value change, and when the number of include statements and variables change then CL_{SCSS} metric value changes. In addition, NF_{SCSS} metric value changes when you reduce or increase nested SCSS blocks, meaning it also satisfies this property.

Property 3: There can exist distinct SCSS blocks P and Q where $|P| = |Q|$.

This property affirms that two different SCSS blocks can have same metric value, this is to say that two SCSS blocks have the same level of complexity. This property was satisfied by all the proposed metrics.

Property 4: $(\exists P) (\exists Q)(P \equiv Q \ \& \ |P| \neq |Q|)$

There can be two SCSS blocks P and Q whose external features look the same, however, due to different internal structure $|P|$ is not equal to $|Q|$. This property asserts that two SCSS blocks with the same number of attributes and directives could return different metric values. This property is satisfied by $ABCC_{SCSS}$, SUIL and CL_{SCSS} . The NF_{SCSS}

metric values could change even in the circumstances where the number of nested rules is the same. Therefore, NF_{SCSS} satisfies this property.

Property 5: $(\exists P) (\exists Q) (|P| \leq |P; Q| \ \& \ |Q| \leq |P; Q|)$

This property asserts that if we concatenate two SCSS blocks P and Q, the new metric value must be greater than or equal to the individual rule block. All the analyzed metrics returned numeric values meaning that they satisfy this property.

Property 6: $(\exists P) (\exists Q) (\exists R) (|P| = |Q| \ \text{and} \ |P; R| \neq |Q; R|)$

This property implies that if two SCSS blocks have same metric value (P and Q), it doesn't necessarily mean that when each of the SCSS blocks is concatenated with similar SCSS block R, the resulting metric values are the same. All the proposed metrics have physical components meaning that they return fixed values. Therefore they don't satisfy this property.

Property 7: If you have two SCSS blocks P and Q which have the same number of attributes in a permuted order, then $|P|$ is not equal to $|Q|$.

This property implies that the order of similar attributes affects their complexity. Therefore, if two rule blocks have the same number of attributes but differ in the ordering, it's not necessary that they have the same complexity level. In the case where the SCSS blocks length is constant and you only change the permutation of the order of statements then all the proposed metrics will retain the same level of complexity. Therefore all the metrics defined didn't meet the property requirements.

Property 8: if P is a renaming of Q, then $|P| = |Q|$

Where you have two SCSS blocks P and Q differing in the naming of selector names, it means $|P|$ is equal to $|Q|$. The metric values for all the proposed metrics are either size measures, complexity measures or coupling measures and they all return numeric values. Therefore, all proposed metrics satisfied this property.

Property 9: $(\exists P) (\exists Q) (|P| + |Q| < (|P; Q|))$

This property asserts that there exist two SCSS blocks P and Q, where the complexity metric value of the two SCSS blocks when summed up is less than when the rule blocks are interacting. The interaction between rule blocks and the growth of rule blocks over time adds to the complexity of rule blocks. The growth of blocks complexity happens when new attributes are added or even when a new SCSS block is added to the existing SCSS block, meaning that the new metric value is equal to or greater than the sum of the two original rule blocks. All the metrics $ABCC_{SCSS}$, NF_{SCSS} , $SUIL$ and CL_{SCSS} satisfied this property.

Findings in Table 5.2 show that all the metrics didn't satisfy property 6 and 7, this is because SCSS interactions don't add any extra external complexity, meaning that the attributes and rule directives are assigned fixed weights. In addition, the permutation of statements don't add any complexity

Table 5.2: Validation Results of SCSS metrics with Weyuker’s Axioms

Property	ABCC _{SCSS}	NF _{SCSS}	SUIL	CL _{SCSS}
1	✓	✓	✓	✓
2	✓	✓	✓	✓
3	✓	✓	✓	✓
4	✓	✓	✓	✓
5	✓	✓	✓	✓
6	×	×	×	×
7	×	×	×	×
8	✓	✓	✓	✓
9	✓	✓	✓	✓

Key: ✓ represents satisfied property

× represents property not satisfied

5.4.2 Validation with Kaner’s Framework

The aim of implementing Kaner framework is to find out if the metrics defined make any sense and to enable the designers to see how the metrics can be used for experimental purposes, thus proving their practicality (Misra et al., 2018). According to Kaner (2004), the following eleven questions should be addressed for purposes of evaluation of software metrics.

i. What is the purpose of this measure?

The purpose of the measure must be clear so as consider it as a valid measure. Therefore, the purpose of this measure is to evaluate the complexity of sassycascading style sheets (SCSS).

ii. What is the scope of this measure?

The measure used should have a specific area it acts on. The proposed metrics will be used by front web developers in web-based projects, particularly those who style the web-documents.

iii. What attribute are we trying to measure?

The attribute to measure will be maintainability through its sub-attributes; understandability, modifiability, and testability.

iv. What is the natural scale of the attribute we are trying to measure?

The proposed metrics will measure understandability, modifiability, and testability and they can all be measured on an ordinal scale

v. What is the natural variability of the attribute?

The quality attributes are subjective in nature, meaning that different SCSS developers can rate the understandability, modifiability and testability of same code differently.

vi. Metrics definition

The metrics must be clearly defined and in this study, the metrics have been defined in section 5.3.

vii. What is the metric and what measuring instrument do we use to perform the measurement?

There are four proposed metrics; $ABCC_{scss}$, NF_{scss} , $SUIL$ and CL_{scss} and they have been computed manually. In addition, a static metrics tool was developed to measure the metrics.

viii. What is the natural scale for this metric?

The natural scale for all the metrics defined fall in the ratio scale

ix. What is the natural variability of readings from this instrument?

When we manually compute the metrics there is no subjectivity to it, meaning that there is no variability. For the metrics tool, the software was tested to ensure no bugs that would lead to erroneous metric values.

x. What is the relationship of the attribute to the metric value?

The maintainability of SCSS is directly related to the proposed complexity metrics. This means we can tell the understandability, modifiability, and testability of SCSS by using the proposed metrics.

xi. What are the natural and foreseeable side effects of using this instrument?

Since the static metrics tool was thoroughly tested and validated, then there will be no negative effects after the implementation of the tool.

The validation results of metrics using Kaner framework show that all the four metrics satisfied its requirements.

5.5 Chapter Summary

This chapter proposed four metrics for measuring the complexity of SCSS code. Code snippets and three Real world projects were used to demonstrate the computation of each of the metric and the metrics proved to be intuitional as shown by the different metrics

values obtained. The metrics were validated using Weyukers properties and the results showed that all the metrics satisfied most of its properties, meaning they are mathematically sound. The study further used Kaner framework to prove the practicality of the metrics and they all proved practical, meaning they can be used for experimental purposes.

CHAPTER SIX

IMPLEMENTATION OF A STRUCTURAL COMPLEXITY METRICS TOOL FOR SASSY CASCADING STYLE SHEETS (SCMT-SCSS)

6.1 Introduction

This chapter presents the Structural Complexity Metrics Tool for Sassy Cascading Style Sheets (SCMT-SCSS) tool which is a prototype metrics tool meant to automate the collection and computing of SCSS complexity metrics values. The chapters intention was to meet the third research objective as stated in the first chapter, which was to develop a functional and usable static metrics analysis tool.

6.2 Requirements of the SCMT-SCSS

The metrics tool was developed to enable the process of collecting, computation and presenting the metrics values. The static analysis metrics tool was developed using Microsoft C# programming language. To ensure the acceptability of the four SCSS metrics, the developed SCMT-SCSS tool was validated by involving 21 subjects who were randomly provided with SCSS files to manually compute metric values and to also compute metrics values with aid of SCMT-SCSS tool.

The tool requirements were identified as:

- The metrics tool accepts all files with .scss extension and the users should be able to locate .scss source files and open them to the tool's user interface.
- The tool operators should compute the metrics and view the computed results. These results are displayed via the tools' textboxes.

- The operators should be able to save the metrics results for future retrieval of results
- The operators of the tool should clear or delete the unnecessary results
- The operators should print the acquired metrics results
- The operators should make use of help functionality to get assistance in the use of the tool.

6.3 Metrics Implementation

The metrics were computed in two levels, i.e. base metrics and derived metrics. The base metrics collects and computes all the metrics directly from the .scss source file. The base metrics were number of regular attributes, number of operators, number of decision nodes, number of function calls, number of mixins defined, number of mixin calls, number of extend directives, number of selectors, number of rule blocks, number of variables defined and number of variables instances. The derived metrics were, Average Block Cognitive Complexity for SCSS (ABCC_{SCSS}), Nesting Factor for SCSS (NF_{SCSS}), Selector Use Inheritance Level (SUIL) and Coupling level for SCSS (CL_{SCSS}), and they make use of the base metrics to compute the final metrics required to measure the complexity of SCSS files.

6.4 Input File Format

The SCSS files serves as the input files. An SCSS file has several features, such as, use of variables, use of mixins, rule-blocks which consists of a selector, opening brace, attributes or declarations, and a closing brace. SCSS file also implements rule nesting, use

of control flows, use of functions and inheritance feature via extend directive. Figure 6.1 illustrates the typical structure of an SCSS file.

```
Variable declarations;
Mixin declarations {
  Attributes/declarations;
}
Selector1 {
  Attributes/declarations;
  Attribute/declaration with variable use;
  Implementation of Mixin;
Selector2 {
  Attributes/declarations;
}
}
Selector3 {
  Attributes/declarations;
  Extend Selector1;
}
Implement control flows {
  Attributes/declarations;
}
Implement function {
  Attributes/declarations;
}
```

Figure 6.1: The Structure of an SCSS file

6.5 SCMT-SCSS Tool Architectural Design

The software system architecture describes the various components of software and how they relate with each other. The SCMT-SCSS tool comprises of three major components, that is, input, analyzer, and output.

6.5.1 Input Component

This purpose of this component is to read and load an SCSS code into the memory. This is achieved by the user clicking on a button named “Open”, or toolbar open icon or via Menu option (File -> Open). Only files with .scss extension are recognized by this component. Once the file is loaded the source code is visible in the textbox which is in the landing tab of the user interface.

6.5.2 Analyzer Component

This role of this component is divided into two phases i.e lexical analysis and parsing. In lexical analysis phase, the .scss source code is broken into tokens and in the parsing phase, the parser accepts input in the form of a sequence of tokens and increments the token flag when it's recognized. The parser is invoked by clicking on analyze button and SCSS metrics are computed.

6.5.3 Output Component

This component enables the user of SCMT-SCSS tool to view the metrics values and save the values in a database (text file format). The user views report as presented via textboxes and can print preview before printing the report. Figure 6.2. Displays the SCMT-SCSS tool architecture.

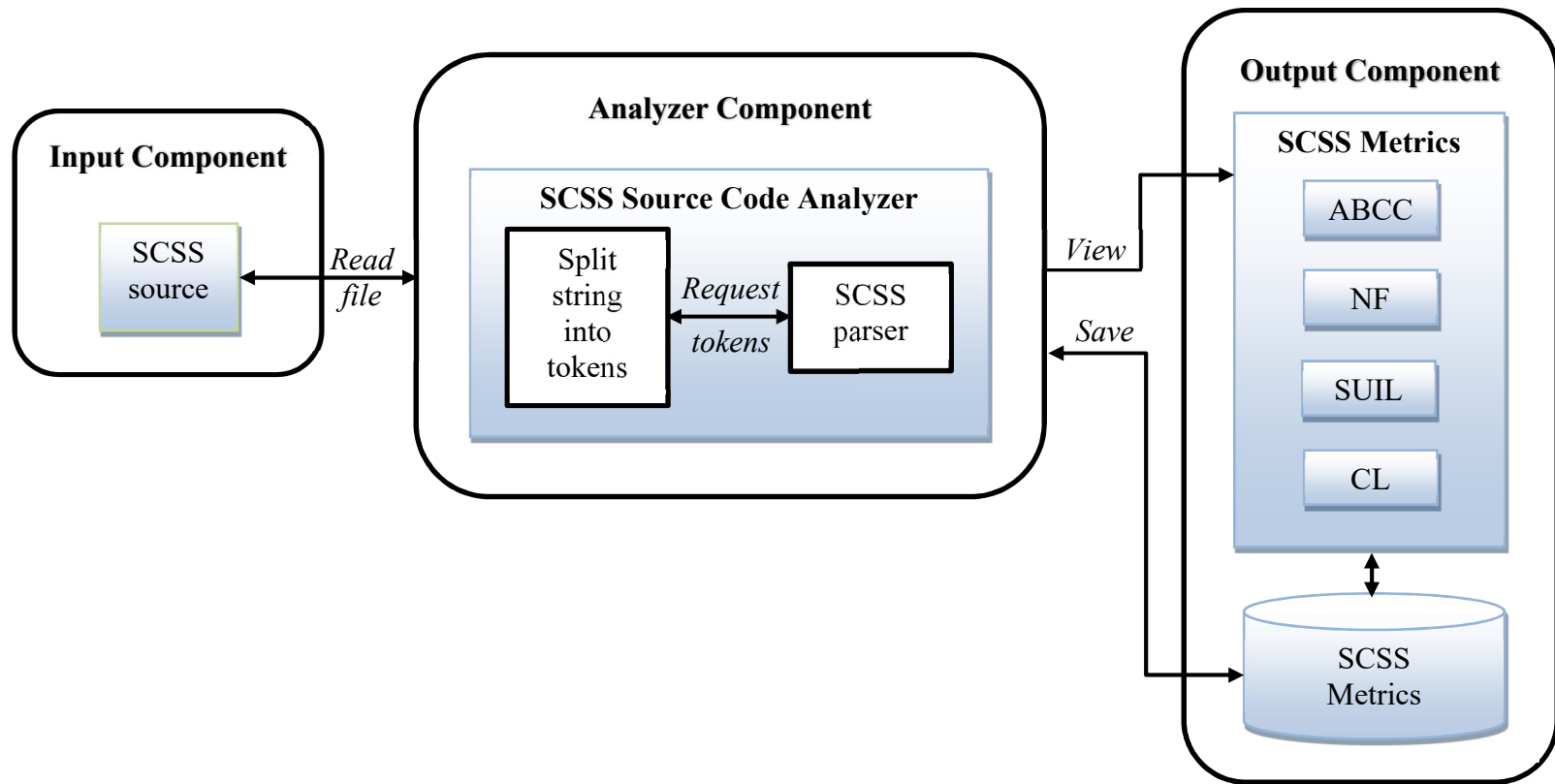


Figure 6.2: SCMT-SCSS Tool Architecture

The representation of the design of a software system is through different types of diagrams such as, data flow diagrams, entity relationship diagrams, Unified Modelling Language diagrams etc. The choice of the diagram to use depends on the programming paradigm. The SCMT-SCSS tool has several modules that interact with each other and some parameters are passed between the modules. Therefore, the structure chart was selected because it well represents the module structure of the software design.

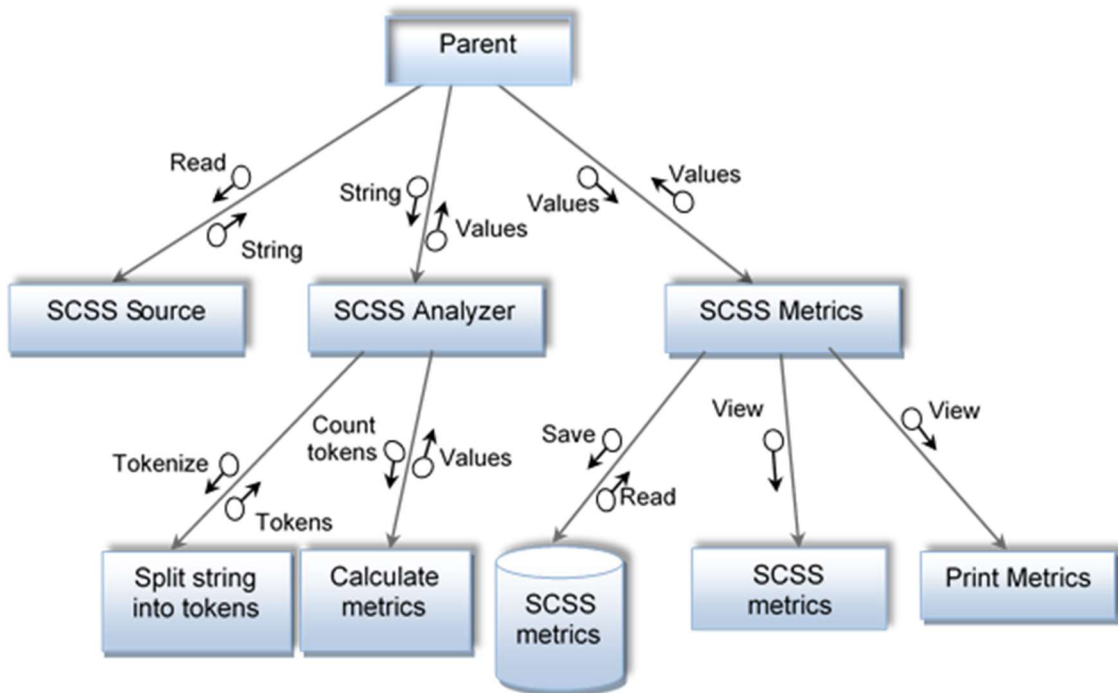


Figure 6.3: SCMT-SCSS Structure Chart Diagram

6.6 User Interface Design

The Use Case diagram was used to represent the user's interaction with the SCMT-SCSS tool. The use cases include viewing the source code, analyzing the SCSS code, displaying SCSS metrics, save metrics, and print the metrics.

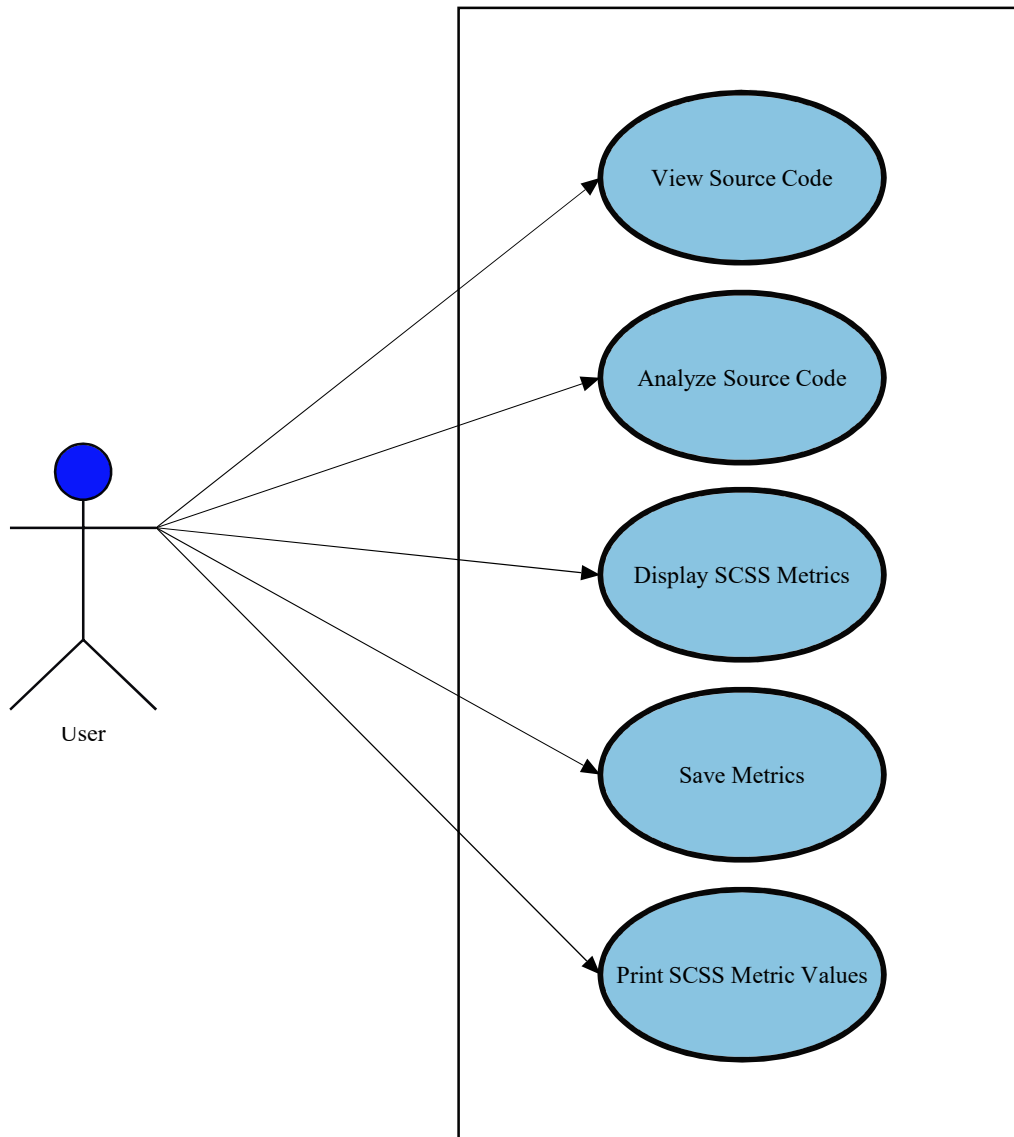


Figure 6.4: SCMT-SCSS Use Case Diagram

The form layout design as shown in Figure 6.5 displays the interface of the tool. The design has a menu bar which has File, View and Help options at top level. The file menu option is used to access the Open, Save, Print and Exit options. The View option determines whether to view tool bar and status bar, while the Help option guides the user on the operation of the system. The interface also shows the tool bar with open, save and print options. The Landing tab allows the user of the tool to open an SCSS file and analyze

the file for the purpose of computing the metrics values. The Base Metrics tab displays the metris collected directly from the file while the derived metrics tab displays the final computed metric.

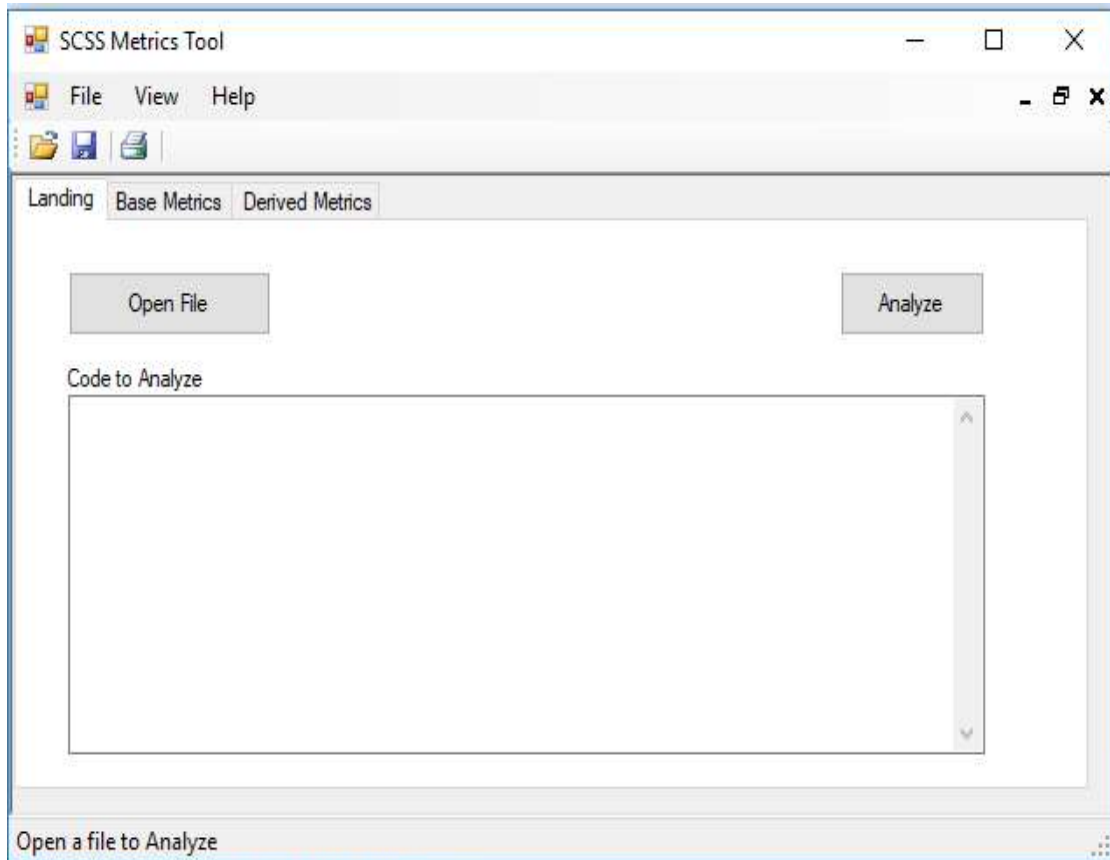


Figure 6.5: Form Layout Design

6.7 Algorithm Design

Algorithms show the steps to be followed to solve a problem and therefore this section identifies the steps for calculating the metric values for Average Block Cognitive Complexity for SCSS ($ABCC_{SCSS}$), Nesting Factor for SCSS (NF_{SCSS}), Selector Use Inheritance Level (SUIL) and Coupling Level for SCSS (CL_{SCSS}).

6.7.1 ABCC_{SCSS} Algorithm

To calculate metric value for ABCC_{SCSS} the following steps were followed:

- i. Count the number of regular attributes (NRA),
- ii. Count the number of operators (NO)
- iii. Count the number of branch statements (NB)
- iv. Count the number of looping statements (NL)
- v. Count the number of function calls (NFC)
- vi. Count the number of mixin calls (NMC)
- vii. Count the number of extend directives (NE)
- viii. Count the number of SCSS blocks (NOBL)

To count NRA:

- i. Lexical analyzer flag is raised to indicate if a regular attribute exists at the beginning of the line.
- ii. A true value is set if the flag exists, otherwise its false.
- iii. NRA count is incremented, if not, no change.

To count NO:

- i. Lexical analyzer flag is raised to indicate if a flag in the lexical analyzer that records whether an operator exists at the beginning of the line.
- ii. A true value is set if the flag exists, otherwise its false.
- iii. NO count is incremented, if not, no change

To count NB:

- i. Lexical analyzer flag is raised to indicate if a branch statement exists at the beginning of the line.

- ii. A true value is set if the flag exists, otherwise its false.
- iii. NB count is incremented, if not, no change.
- iv. Final NB count is multiplied by 1.3 as the assigned weight of a branch

To count NL:

- i. Lexical analyzer flag is raised to indicate if a looping statement exists at the beginning of the line.
- ii. A true value is set if the flag exists, otherwise its false.
- iii. NL count is incremented, if not, no change.
- iv. Final NL count is multiplied by 1.5 as the assigned weight of a loop.

To count NFC:

- i. Lexical analyzer flag is raised to indicate if a function call exists at the beginning of the line.
- ii. A true value is set if the flag exists, otherwise its false.
- iii. NFC count is incremented, if not, no change.
- iv. Final NFC count is multiplied by 1.3 as the assigned weight of a function call.

To count NMC:

- i. Lexical analyzer flag is raised to indicate if a mixin call exists at the beginning of the line.
- ii. A true value is set if the flag exists, otherwise its false.
- iii. NMC count is incremented, if not, no change.
- iv. Final NMC count is multiplied by 1.3 as the assigned weight of a mixin call.

To count NE:

- i. Lexical analyzer flag is raised to indicate if a extend directive exists at the beginning of the line.
- ii. A true value is set if the flag exists, otherwise its false.
- iii. NE count is incremented, if not, no change.
- iv. Final NE count is multiplied by 1.3 as the assigned weight of a extend directive.

To count NOBL:

- i. Lexical analyzer flag is raised to indicate if a block exists at the beginning of the line.
- ii. A true value is set if the flag exists, otherwise its false.
- iii. NOBL count is incremented, if not, no change.

To measure $ABCC_{SCSS}$:

- i. Locate the variables holding current values of NRA, NO, NB, NL, NFC, NMC, NE and NOBL
- ii. Add the values of NRA, NO, NB, NL, NFC, NMC, and NE
- iii. Divide the total with NOBL

6.7.2 NF_{SCSS} Algorithm

To calculate metric value for NF_{SCSS} the following steps will be followed:

- i. Count total depth nesting level (TDNL)
- ii. Count total breadth nesting level (TBNL)

To count total depth nesting level

- i. Raise a flag in the lexical analyzer to indicate if depth of SCSS rules has been seen since the start of code
- ii. A true value is set if the flag exists, otherwise its false.
- iii. The depth of nesting value is incremented for each of the nested rule blocks, if not, no change.
- iv. The final count if its 5, then the total depth nesting level is $(5+4+3+2+1) = 15$
- v. Locate next block with nested blocks and repeat step 4
- vi. Total depth nesting level is incremented until end of code.

To count total breadth nesting level

- i. Raise a flag in the lexical analyzer to indicate if breadth of SCSS rules has been seen at the beginning of the line.
- ii. A true value is set if the flag exists, otherwise its false.
- iii. The total breadth nesting value is incremented, if not, no change.

To measure NF_{scss} :

- i. Find the TDNL and TBNL values
- ii. Get the product of TDNL and TBNL.

6.7.3 SUIL Algorithm

To calculate metric value for SUIL, the following steps will be followed:

- i. Count the number of selector instances (NSI)

- ii. Count the number of selectors (NS)

To count the number of selector instances

- i. Lexical analyzer flag is raised to indicate if an extend directive exists at the beginning of the line.
- ii. A true value is set if the flag exists, otherwise its false.
- iii. NSI count is incremented, if not, no change.

To count the number of selectors

- i. Lexical analyzer flag is raised to indicate if a selector exists at the beginning of the line.
- ii. A true value is set if the flag exists, otherwise its false.
- iii. NS count is incremented, if not, no change.

To measure SUIL:

- i. Locate the variables that hold current values of NSI and NS
- ii. Divide total NSI with total NS, i.e. $SUIL = NSI / NS$

6.7.4 CLscss Algorithm

This metric is computed following these major steps

- i. Count the number of declared mixins (NDM)
- ii. Count the number of declared variables (NDV)
- iii. Count the number of mixin calls (NMC)
- iv. Count the number of variable instances (NVI)

To count the number of declared mixins

- i. Lexical analyzer flag is raised to indicate if a mixin exists at the beginning of the line.
- ii. A true value is set if the flag exists, otherwise its false.
- iii. NDM count is incremented, if not, no change.

To count the number of declared variables

- i. Lexical analyzer flag is raised to indicate if a variable exists at the beginning of the line.
- ii. A true value is set if the flag exists, otherwise its false.
- iii. NDV count is incremented, if not, no change.

To count the number of mixin calls

- i. Lexical analyzer flag is raised to indicate if a mixin call exists at the beginning of the line.
- ii. A true value is set if the flag exists, otherwise its false.
- iii. NMC count is incremented, if not, no change.

To count the number of variable instances

- i. Lexical analyzer flag is raised to indicate if a variable instance exists at the beginning of the line.
- ii. A true value is set if the flag exists, otherwise its false.
- iii. NVI count is incremented, if not, no change.

6.8 Execution of the SCMT-SCSS Tool

The SCMT-SCSS tool functions as described:

1. The user begins by opening an SCSS source file. This is achieved by clicking Open button in the landing page, or through the menu option (File -> Open) or tool bar open icon
2. To calculate the SCSS metrics the user clicks on the analyze button
3. The user can view the metric results as presented in the textboxes.
4. The metrics results can be saved via menu option File -> Save or via tool bar icon. The metrics results are saved as text file.
5. The user can print the metrics results via menu option File -> Print or via tool bar icon
6. The user can use help function if required to do so.

The base metrics are gathered directly from the SCSS file and the computed metric values are displayed as illustrated in Figure 6.6.

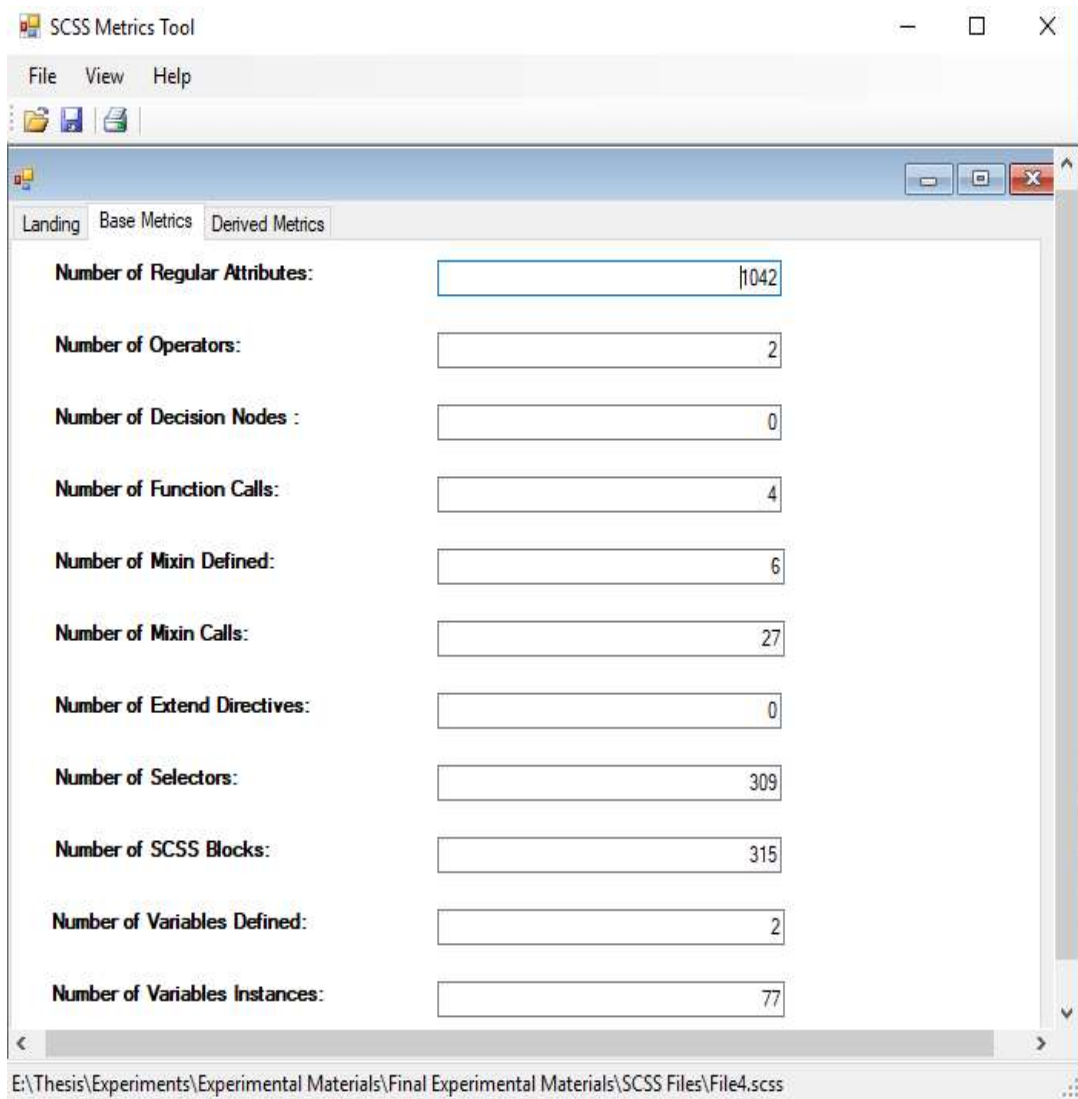


Figure 6.6: SCSS Base Metrics Values

Derived metrics are computed based on the base metrics. The metrics values are displayed in the tool as illustrated in Figure 6.7.

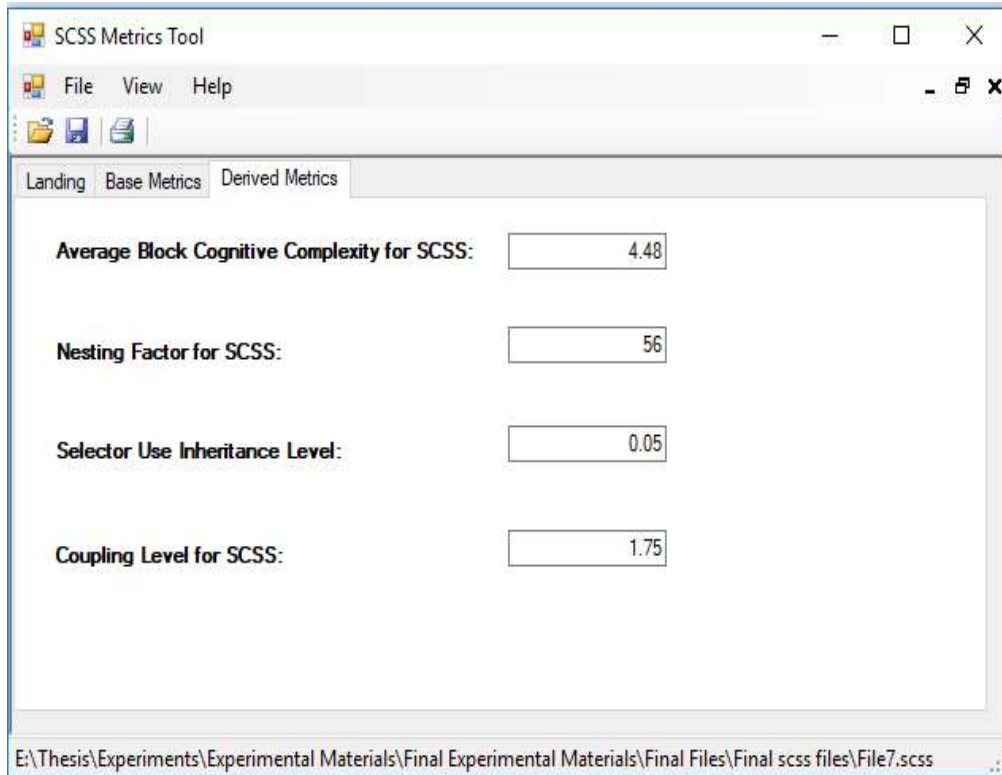


Figure 6.7: SCSS Derived Metrics Values

The saved metric values in a text file are as shown in Figure 6.8. The values are as a result of running the tool and computing the metric values of the loaded file. These results can be retrieved for future purposes.

```
SUMMARY REPORT

File Name: E:\Thesis\Experiments\Experimental Materials\Final Experimental
Materials\Final Files\Final scss files\File7.scss

BASE METRICS
1. Number of Regular Attributes:      115
2. Number of Operators:               0
3. Number of Decision Nodes:         0
4. Number of Function Calls:         0
5. Number of Mixins Defined:         7
6. Number of Mixin Calls:            4
7. Number of Extend Directives:      1
8. Number of Selectors:              21
9. Number of SCSS Blocks:            28
10. Number of Variables Defined:     7
11. Number of Variables Instances:   15

DERIVED METRICS
1. Average Block Cognitive Complexity for SCSS: 4.48
2. Nesting Factor for SCSS:          56
3. Selector Use Inheritance Level:   0.05
4. Coupling Level for SCSS:         1.75
```

Figure 6.8: SCSS Metrics Values in a Text File

6.9 Experimental Validation of the SCMT-SCSS Tool

6.9.1 Goal of the Study

The goal of this study was to evaluate the tool in terms of its effectiveness, efficiency, accuracy, suitability and operability.

6.9.2 Context Definition

The subjects involved in this study were students in the School of Computing and Information Technology from Murang’a University of Technology, Kenya. The students selected were fourth years pursuing Bachelor of Science in Information Technology, Bachelor of Science in Software Engineering, Bachelor of Business Information

Technology. Third year students pursuing Bachelor of Science in Software Engineering were also involved. A convenient sample of 21 subjects was selected.

6.9.3 Threats to Validity

6.9.3.1 Internal Validity

The threat involved was that the subjects rated some aspects of the tool such as suitability, accuracy, and operability subjectively. However, the subjects were trained on the usage of the tool before performing the tasks provided.

6.9.3.2 External Validity

The threat involved was that the subjects are not industry-based persons, however, the threat was significantly reduced by offering intensive training to the subjects on SCSS language, how to compute metrics manually and the use of the tool. Meaning that the subjects were well able to gather metrics from an SCSS file and implement the tool to automate metrics computation.

6.9.4 Experimental Design

The experimental materials used were 4 SCSS files, and were randomly distributed to 21 subjects. The subjects were guided on how to carry out the experiment. All the subjects did the experiments in a computer laboratory and were given enough time to complete the tasks. The subjects first task was to calculate the metric values for the file provided manually i.e without the tool and the second task was to calculate metric values for the same file using the tool. When performing both tasks, the subjects recorded the time taken

in terms of Minutes and seconds. Finally, the subjects were required to rate the suitability, accuracy, and operability of the SCMT-SCSS tool using a Likert scale of 1-5.

6.10 Results

6.10.1 Time to Complete Tasks

The feedback from the subjects was collected and checked for completeness. All the required responses were completed by all the subjects. Therefore, all data from the 21 subjects was analysed.

This experiment was carried out to test the tool effectiveness and efficiency. This was performed by calculating the metrics values for the provided file both manually and by use of the tool. The mean of metrics computation was calculated and provided the basis for conclusion on tool effectiveness and efficiency.

The mean time for calculating the metric values without a tool for SCSS File number 1 was 18 Minutes 3 seconds, while calculation of the metric values with the tool for the same file took an average of 42 Seconds. These results are presented in Table 6.1, and they imply that the use of the tool was far much better and saved a lot of time for the metrics values computation of SCSS File1.

Table 6.1: Time to Complete Tasks for SCSS File 1

Subject	Metrics Values Calculation time without Tool (MM:SS)	Metrics Values Calculation time with Tool (MM:SS)
1	24.52	0.27
2	10.07	1.00
3	30.24	0.28
4	11.19	0.36
5	11.19	0.24
6	18.54	0.38
Mean	18.03	0.42

The computed metrics for SCSS File number 2 are shown in Table 6.2. There were five subjects involved and the mean of calculating the metric values without tool was 26 Minutes 28 seconds while calculation of the metric values with the tool took an average of 47 Seconds. This proves that the tool achieves results with far much less time in comparison to manual calculation.

Table 6.2: Time to Complete Tasks for SCSS File 2

Subject	Metrics Values Calculation time without Tool (MM:SS)	Metrics Values Calculation time with Tool (MM:SS)
1	25.31	0.28
2	27.33	0.26
3	25.5	1.2
4	23.07	0.25
5	28.22	0.37
Mean	26.28	0.47

The SCSS File number 3 was assessed by 5 subjects and as shown in Table 6.3 the mean of calculating the metric values without a tool is 15 Minutes 34 seconds while calculation of the metric values with tool took an average of 27 Seconds. This shows that the

automation of metrics calculation achieves results in a shorter time as compared to manual calculation metrics values for the same SCSS file.

Table 6.3: Time to Complete Tasks for SCSS File 3

Subject	Metrics Values Calculation time without Tool (MM:SS)	Metrics Values Calculation time with Tool (MM:SS)
1	20	0.21
2	15.55	0.29
3	9.35	0.36
4	11.42	0.23
5	18.38	0.27
Mean	15.34	0.27

The computed metrics for SCSS File number 4 are shown in Table 6.4. There were five subjects involved to compute metrics values for the file both manually and with the use of a tool. The mean of calculating the metric values without tool was 16 Minutes 25 seconds while calculation of the metric values with the tool took an average of 29 Seconds. This proves that the tool achieves results with far much less time in comparison to manual calculation.

Table 6.4: Time to Complete Tasks for SCSS File 4

Subject	Metrics Values Calculation time without Tool (MM:SS)	Metrics Values Calculation time with Tool (MM:SS)
1	15.56	0.26
2	13.13	0.27
3	21.06	0.44
4	21.38	0.32
5	8.13	0.16
Mean	16.25	0.29

6.10.2 Suitability, Accuracy and Operability Rating

The subjects were asked questions concerning the suitability, accuracy, and operability of the SCMT-SCSS Metrics tool. The subjects were required to give a rating using a Likert scale of 1 to 5 (see Appendix 7). The subjects' ratings were averaged, and they all acquired a mean greater than 4. The standard deviation for all the responses were less than 1.0, meaning that the subjects' responses did not differ too much.

Suitability is the capability of the tool to provide an adequate set of functions for the tasks to be carried out, while accuracy is the capability of the tool to provide correct results and operability is defined as the capability of the tool to allow the user to operate it (ISO, 2001).

The subjects were asked the question, how do you rate the suitability of SCMT-SCSS Metrics tool? The average of the responses is 4.52 as shown in Table 6.5 and is a high rating on the suitability of the tool. This means that the subjects were able to use the menu bar, toolbar and buttons provided to load an SCSS file and compute SCSS metrics values. Therefore, the researcher concluded that the SCMT-SCSS tool provides a set of functions which enable SCSS metrics values computation.

Table 6.5: Average Rating on Suitability

	Mean	Std. Deviation
Subject Rating on Tool Suitability	4.52	0.60

The subjects were asked the question, how do you rate the accuracy of SCMT-SCSS Metrics tool? The average of the responses is 4.43 as shown in Table 6.6 and is a high rating on the accuracy of the tool. This implies that the subjects were able to obtain results which were accurate. Therefore, the research concluded that the SCMT-SCSS tool can be used to provide correct SCSS metrics values for any SCSS file.

Table 6.6: Average Rating on Accuracy

	Mean	Std. Deviation
Subject Rating on Tool Accuracy	4.43	0.60

The subjects were asked the question, how do you rate the operability of the SCMT-SCSS Metrics tool? The average of the responses is 4.76 as shown in Table 6.7 and is a high rating on the operability of the tool. This implies that the subjects were able to easily operate or use the tool to perform the task assigned. Therefore, the researcher concluded that the SCMT-SCSS tool is operable.

Table 6.7: Average Rating on Operability

	Mean	Std. Deviation
Subject Rating on Tool Operability	4.76	0.44

6.11 Chapter Summary

The development of a metrics tool is a basic requirement for the defined software metrics to be acceptable in the software industry. There are several metrics proposed over the years without tool support and this trend cannot be tolerated. Therefore, a metrics tool to

automate the collection and analyze the defined SCSS metrics was developed and is herein referred to as SCMT-SCSS Metrics Tool. The tool was validated by using 4 SCSS files with a convenient sample of 21 students who were trained in SCSS language and on the usage of the SCMT-SCSS metrics tool. The results indicated that the tool computes metrics in much less time than manual computation. The subjects rated the tool as suitable for the tasks assigned, and in comparison, of the metrics values results obtained from the tool and without the tool, the subjects rated the tool as accurate. In addition, the subjects rated the tool as easy to operate.

This study proposed an SCMT-SCSS metrics tool and was validated using an experiment. The tool was found to be very effective and efficient because as observed it took far less time to complete a similar task. The manual computation of metrics is an inefficient process in terms of time taken and the software industry may not appreciate the metrics in the absence of a tool. The results from the subjects rating suggested strongly that the tool provided a required set of functions to compute SCSS metrics values and therefore it's suitable for the tasks. In addition, the subjects strongly agreed that the tool provides accurate results. The results by different subjects differed in terms of metrics values obtained manually, but the tool provided consistent and accurate results. Lastly, the SCMT-SCSS tool proved operable and users were able to easily navigate through the tool to execute the tasks provided.

CHAPTER SEVEN

AN EXPERIMENTAL VALIDATION OF STRUCTURAL COMPLEXITY METRICS FOR SASSY CASCADING STYLE SHEETS

7.1 Introduction

The two phases of the experiment are described in this chapter. In the first phase subjective experiment was carried out and second phase objective experiment was conducted. The purpose of the experiment was to check if the proposed metrics can predict the maintainability of SCSS code. Pilot study was carried out to identify any important omissions, clarifications and corrections.

7.2 Context Definition

Thirty SCSS files were availed for the experiment to 30 students from Murang'a University of Technology in the School of Computing and Information Technology. The students involved were fourth year students pursuing Bachelor of Science in Software Engineering, Bachelor of Information Technology and Bachelor of Business Information Technology because the researchers believed they were more knowledgeable in software engineering processes in comparison with other students. In addition, third year students pursuing Bachelor of Science in Software Engineering were also involved because when they studied the web development unit, they were introduced to CSS pre-processors specifically SASS using .scss syntax. Therefore, the researcher believed they were well placed to participate in this study. All the subjects who participated in the experiment were trained on SCSS language intensively for 27 hours.

7.3 Strategy for Conducting the Experiment

The experiment was conducted in two phases, where in the first phase it involved subjects ranking maintainability sub-characteristics, while in the second phase, involved subjects indicating the understanding time, modification time and testing time of each SCSS file. A between subject design was used where ten groups were involved and each group worked on same files. The subjects worked individually for a period of two hours. The total experimental objects provided for the experiment was 30 SCSS files which were assumed to have correct syntax. Each subject was provided with three SCSS files. The files had different complexity values as evidenced by the metrics values gathered (see Appendix 3). The SCMT-SCSS tool was included as a material to automate gathering and calculation of metrics. The subjects did not use the metrics tool, and so only the researcher used the tool to collect metric values of the files provided to the subjects.

Before the experiment was conducted a pilot study was carried out. Through the pilot study, lessons were learned and were used to inform the process of carrying out the final experiment.

7.4 Pilot Study

A pilot study was carried out using a between-subject experimental design. The design was chosen because SCSS files are normally long files and this ensured that the subjects don't take a lot of time working on the files which could lead to boredom. Therefore, each subject only worked on three SCSS files of the 10 files available. The study aimed at finding if the proposed SCSS metrics correlate with the subjects rating of

understandability, modifiability and testability. The study was also conducted to find if the proposed SCSS metrics are valid measures of SCSS maintainability. Finally, the pilot study was performed to test and validate the questionnaire. The questionnaire used for both subjective and objective phases is shown in Appendix 2.

A convenient sample of 10 subjects were involved in pilot study. They were required to rate the SCSS files provided in Likert scale of 1-5 i.e. Very Difficult to Very Easy. The rating for each file was in terms of understandability, modifiability and testability. The metrics values collected with SCMT-SCSS tool were correlated with the mean of subject ratings for each file. A number of questions were availed to the subjects on understandability, modifiability and testability aspects of maintainability. While the subjects responded to the questions provided, the time to understand, time to modify and time to test each of the SCSS file was recorded and their means were computed for each file. The metrics values for each SCSS file was computed using the SCMT-SCSS tool and correlation of the means for understanding time, modification time and testability time was done.

In the pilot study it was learnt that the subjects took a lot of time on each file and so they were unable to finish the tasks within the expected two hours. As a result of much time taken to perform all tasks, the subjects became too exhausted and hurriedly finished the last SCSS file provided, thus affecting the results. Therefore, the questions were reduced from three to two for each of the understandability, modifiability and testability sections.

It was also learnt that a clear guideline should be provided before performing an experiment so that the experiment is carried out smoothly.

7.5 Subjects' Background

The information about subjects' knowledge on programming languages, software engineering, and SCSS features was established to ascertain the subjects readiness to perform the experimental tasks.

The subjects were asked about the programming languages they have knowledge in. This was to establish their grounding in various programming concepts such as inheritance, declaring variables, nesting, control structures e.t.c. Fifteen subjects which account for 50% said they had taken between five to six programming languages, and other fifteen subjects, which is 50% of the total responses said they had taken above 6 programming languages, as indicated in Table 7.1.

Table 7.1: Programming languages taken

Programming Languages pursued	Number of Subjects	Percent (%)
5-6	15	50
Above 6 courses	15	50

The subjects were asked about the number of software engineering courses they had taken. This was to establish their knowledge of the software engineering processes and software

engineering concepts such as quality of software. The responses were as shown in Table 7.2, where eight subjects, which is 26.7% of the responses had taken one to four software engineering courses while a majority of twenty-two of them (73.3 %), had taken more than four software engineering courses.

Table 7.2: Software Engineering courses pursued

Software Engineering Courses pursued	Number of Subjects	Percent (%)
1-4	8	26.7
>4	22	73.3

The subjects were asked about the number of SCSS features they can comfortably implement in an SCSS code. This was to establish the level of SCSS knowledge. As shown in Table 7.3, fifteen subjects indicated they could comfortably use four to six features of the eight main features of SCSS, while fifteen subjects said they could comfortably implement more than six features of SCSS language. These results imply that all the subjects had at least moderate level knowledge of SCSS.

Table 7.3: Knowledge of SCSS

SCSS features	Number of Subjects	Percent (%)
4-6	15	50
>6	15	50

7.6 Subjective Data

The subjective part of the experiment was intended to investigate the existence of a relationship between SCSS metrics and the rating of SCSS files by subjects in terms of understandability, modifiability and testability of SCSS files. To achieve this, the SCSS metrics values and subjects' rating values were captured and analysis was performed on the data.

7.6.1 Experimental Planning

The materials to be used in the experiment were distributed to the subjects in the computer laboratory, that is, the SCSS files and a questionnaire.

7.6.1.1 Effect of SCSS Metrics on Subjects Rating of Understandability

The independent variables in this study refer to the collected SCSS metrics values while the dependent variable is the subjects rating on the understandability of the SCSS files.

The hypotheses under investigation in the experiment were for the purpose of establishing if the SCSS metrics correlate with the subjects rating of understandability of SCSS files. The hypotheses were as follows:

- Null Hypothesis (H_{0-u}): There exists no significant correlation between the SCSS metrics and subjects rating of understandability of SCSS files.
- Alternative Hypothesis (H_{1-u}): There exists significant correlation between the SCSS metrics and subjects rating of understandability of SCSS files.

7.6.1.2 Effect of SCSS Metrics on Subjects Rating of Modifiability

The subjective phase of the experiment intention was to investigate whether the proposed SCSS metrics correlate with modifiability of SCSS files. The independent variables in this study refers to the collected SCSS metrics values and dependent variable is the modifiability of the SCSS files based on subjects rating.

The hypotheses under investigation in the experiment were to establish if the SCSS metrics correlate with the subjects rating of modifiability of SCSS files. The hypotheses were as follows:

- Null Hypothesis (H_0 -m): There exists no significant correlation between the SCSS metrics and subjects rating of modifiability of SCSS files.
- Alternative Hypothesis (H_1 -m): There exists significant correlation between the SCSS metrics and subjects rating of modifiability of SCSS files.

7.6.1.3 Effect of SCSS Metrics on Subjects Rating of Testability

The subjective phase of the experiment intention was to investigate whether the proposed SCSS metrics correlate with the subjects rating of SCSS files testability. The independent variables in this study refer to the collected SCSS metrics values and dependent variable is the subjects rating on testability.

The hypotheses under investigation in the experiment were to establish if the SCSS metrics correlate with the subjects rating of testability of SCSS files. The hypotheses were as follows:

- Null Hypothesis (H_0): There exists no significant correlation between the SCSS metrics and subjects rating of testability of SCSS files.
- Alternative Hypothesis (H_1): There exists significant correlation between the SCSS metrics and subjects rating of testability of SCSS files.

7.6.2 Threats to validity

7.6.2.1 Internal validity

This type of validity ensures that we can trust the cause and effect relationship and is achieved by controlling the factors that can affect the dependent variable. The subjects provided the rating on understandability, modifiability and testability level of SCSS files based on their perceptions. The subjectivity of the experiment was a threat and therefore to lessen it, an intensive training to the subjects on SCSS language was carried out. This training reduced the skills gap in SCSS, thus improving on the subject's validity ratings. In addition, the subjects can be considered as having moderate level experience based on Table 7.4 results. The Subjects mean on the number of programming languages they have done in their course of study is 3.50 on a Likert scale of 1-4. This means that the subjects are knowledgeable in programming concepts. The subjects mean of the number of software engineering courses they have pursued is 2.73 in a Likert scale of 1-3, this means that the subjects have taken at least four courses. The subjects mean of the number of SCSS features they can comfortably implement is 2.5 in a Likert scale of 1-3. This tends to mean of 3, meaning that the subjects have moderate experience in SCSS.

To further ensure internal validity the independent variables were measured via SCSS metrics which were theoretically validated.

Table 7.4: Subjects Background Knowledge

Scale	Mean	Std. Deviation
#Programming Languages	3.50	0.508
#Software Engineering Courses	2.73	0.449
#SCSS Features	2.50	0.508

7.6.2.2 External Validity

External validity implies that the results from the study can be generalized. Though the subjects in the study were students, the researchers selected fourth year students and third year students because they had accrued knowledge in area of programming and software engineering as supported by results in Table 7.4. In addition, all the subjects had been involved in the development of a web-based project, implying that they had an exposure to real world projects. The threat to external validity was significantly reduced by the moderate experience of subjects in software development.

7.7 Objective Data

The objective phase of the experiment was intended to establish if any relationship exists between the SCSS metrics and time to understand, time to modify and time to test SCSS files. To achieve this, the SCSS metrics values and time taken to perform the tasks provided were captured and analysis was performed on the data. Further analysis were conducted with principle component analysis (PCA) to establish which variables significantly contribute to the understandability, modifiability and testability models at 80% variance

A Kaiser-Meyer-Olkin (KMO) and Bartlett's tests were conducted before performing PCA to indicate the proportion of variance in the variables that may be caused by underlying factors. The tests establish the suitability of data for structure detection. It was established that the KMO measure was 0.725, as shown in Table 7.5 which is greater than the recommended value of >0.5 , meaning that the Bartlett's Test of Sphericity is significant

Table 7.5: KMO and Bartlett's Test

KOM and Bartlett's Tests		Coefficients
Kaiser-Meyer-Olkin Measure of Sampling Adequacy.		.725
Bartlett's Test of Sphericity	Approx. Chi-Square	46.841
	Df	6
	Sig.	.000

7.7.1 Experimental Planning

The SCSS files to be used in the experiment were the same as distributed during subjective experiment.to the subjects.

7.7.1.1 Effect of SCSS Metrics on Subjects Understanding time

The objective phase of the experiment intention was to investigate if any correlation exists between the SCSS metrics and time to understand SCSS files. This kind of experiment is done to reduce the shortcomings of results obtained due to the subjective nature of data.

The hypotheses under investigation in the experiment were for the purpose of checking if SCSS metrics correlate with understanding time of SCSS files. The hypotheses were as follows:

- Null Hypothesis (H_0 -ut): There exists no significant correlation between the SCSS metrics and understanding time of SCSS files.
- Alternative Hypothesis (H_1 -ut): There exists significant correlation between the SCSS metrics and understanding time of SCSS files.

7.7.1.2 Effect of SCSS Metrics on Subjects Modifying Time

The objective phase of the experiment intention was to investigate if the proposed SCSS metrics correlate with modifying time of SCSS files. This kind of experiment is done to reduce the shortcomings of results obtained due to the subjective nature of data.

The hypotheses under investigation in the experiment were for checking if SCSS metrics correlate with the modifying time of SCSS files. The hypotheses were as follows:

- Null Hypothesis (H_0 -mt): There exists no significant correlation between the SCSS metrics and modifying time of SCSS files.
- Alternative Hypothesis (H_1 -mt): There exists significant correlation between the SCSS metrics and modifying time of SCSS files.

7.7.1.3 Effect of SCSS Metrics on Subjects Testing Time

The objective phase of the experiment intention was to investigate if the proposed exists SCSS metrics correlate with the testing time of SCSS files. This kind of experiment is done to reduce the shortcomings of results obtained due to the subjective nature of data.

The hypotheses under investigation in the experiment were for the purpose of establishing if the proposed SCSS metrics correlate with the testing time of SCSS files.

The hypotheses were as follows:

- Null Hypothesis (H_0 .tt): There exists no significant correlation between the SCSS metrics and testing time of SCSS files.
- Alternative Hypothesis (H_1 .tt): There exists significant correlation between the SCSS metrics and testing time of SCSS files.

7.7.2 Threats to Validity

7.7.2.1 Internal Validity

The metric values acquired via the dependent variables were objective, that is, the time to understand, modify and test the SCSS files was recorded. This means that the values obtained for the dependent variable are valid. The choice of between subject design over the within subject design ensured that the subjects' fatigue is reduced significantly. In addition, the subjects volunteered to participate in the experiment, meaning that they had high self-drive.

7.7.2.2 External Validity

The use of students as subjects in the experiment introduced external threat, but the students selected were fourth year students and third year students. This means that they have advanced knowledge in programming and software engineering as shown in Table 7.10. These students had participated in a web-based project and during the SCSS training every subject developed a simple SCSS based project.

7.8 Results

The completeness of questionnaire was checked and a threshold of 70% was set for inclusion of questionnaire in the analysis stage. All subjects attained the threshold; however, three questionnaires were not filled completely and so they were rejected for inclusion in data analysis.

The Spearman Rank Order Correlation coefficient (r_s) is a non-parametric measure of the strength and direction of association that exists between two variables on a scale that is at least ordinal. This method of correlation was chosen after Shapiro-Wilk test was performed and the results showed that the data was non-normal.

7.8.1 Subjective Results

The first data set was for metrics values and was collected via SCMT-SCSS tool and they represent the independent variables while the subject's ratings of understandability, modifiability and testability represents the dependent variables.

7.8.1.1 Relationship between Metrics and Understandability

The correlation of SCSS metrics values with understandability is shown in Table 7.6. All the metrics are significantly correlated to the subjects rating of the SCSS code understandability. The $ABCC_{SCSS}$ metric is correlated with understandability as shown by the correlation coefficient value of 0.383 at 95% confidence level. The NF_{SCSS} has a correlation coefficient of -0.684, $SUIL$ is -0.560, and CL_{SCSS} is -0.550 and are all at 99% confidence level.

Table 7.6: Correlation with Understandability

SCSS Metrics	Correlation Coefficient	Sig. (2-tailed)
$ABCC_{SCSS}$	0.383*	0.049
NF_{SCSS}	-0.684**	0.000
$SUIL$	-0.560**	0.002
CL_{SCSS}	-0.550**	0.003

**=99% confidence, *=95% confidence

The Analysis of Variance (ANOVA) test resulted to a P-value (“Sig” for significance) of 0.006 as shown in Table 7.7. This value is below the prescribed $P < 0.05$, meaning it is statistically significant, and that SCSS metrics means are not equal, therefore, there is a strong evidence against null hypothesis. This study concludes that the proposed metrics are good indicators of understandability of SCSS code.

Table 7.7: Understandability Significance with ANOVA

Model	Sum of Squares	df	Mean Square	F	Sig.
Regression	6.415	4	1.604	4.877	.006a
Residual	7.234	22	.329		
Total	13.649	26			

a. Predictors: (Constant), Coupling Level, Average Block Cognitive Complexity, Selector Use inheritance Level, Nesting Factor

b. Dependent Variable: Understandability

7.8.1.2 Relationship between Metrics and Modifiability

The correlation of SCSS metrics values with modifiability is shown in Table 7.8. All the metrics are significantly correlated to the subjects rating of the SCSS code modifiability. The $ABCC_{SCSS}$ metric is correlated with modifiability as shown by the correlation coefficient value of 0.409 at 95% confidence level. The NF_{SCSS} has a correlation coefficient of -0.686, $SUIL$ is -0.644, and CL_{SCSS} is -0.574 and are all at 99% confidence level.

Table 7.8: Correlation with Modifiability

SCSS Metrics	Correlation Coefficient	Sig. (2-tailed)
$ABCC_{SCSS}$	0.409*	0.034
NF_{SCSS}	-0.686**	0.000
$SUIL$	-0.644**	0.000
CL_{SCSS}	-0.574**	0.002

**=99% confidence, *=95% confidence

The ANOVA results for modifiability resulted to a P-value (“Sig” for significance) of 0.005 as shown in Table 7.9. This value is below the prescribed $P < 0.05$, and this is a strong indication against the null hypothesis, in addition, it shows that the SCSS metrics means are not equal, therefore, the study concludes that the proposed metrics are good indicators of modifiability of SCSS code.

Table 7.9: Modifiability Significance with ANOVA

Model		Sum of Squares	df	Mean Square	F	Sig.
1	Regression	7.758	4	1.940	4.960	.005 ^a
	Residual	8.602	22	.391		
	Total	16.360	26			

a. Predictors: (Constant), Coupling Level, Average Block Cognitive Complexity, Selector Use inheritance Level, Nesting Factor

b. Dependent Variable: Modifiability

7.8.1.3 Relationship between Metrics and Testability

The correlation of SCSS metrics values with testability is shown in Table 7.10. All the metrics are significantly correlated to the subjects rating of the SCSS code testability. The $ABCC_{SCSS}$ metric is correlated with testability as shown by the correlation coefficient value of 0.385 at 95% confidence level. The NF_{SCSS} has a correlation coefficient of -0.703, $SUIL$ is -0.572, and CL_{SCSS} is -0.734 and are all at 99% confidence level.

Table 7.10: Correlation with Testability

SCSS Metrics	Correlation Coefficient	Sig. (2-tailed)
ABCC _{SCSS}	0.385*	0.048
NF _{SCSS}	-0.703**	0.000
SUIL	-0.572**	0.002
CL _{SCSS}	-0.734**	0.000

**=99% confidence, *=95% confidence

The ANOVA results for testability showed that the P-value (“Sig” for significance) was 0.003 as shown in Table 7.11. This value is below the prescribed $P < 0.05$, meaning the SCSS metrics means are not equal and its statistically significant. The null hypothesis is therefore ruled out and the study concludes that the proposed metrics are good indicators of testability of SCSS code.

Table 7.11: Testability Significance with ANOVA

Model		Sum of Squares	Df	Mean Square	F	Sig.
1	Regression	7.737	4	1.934	5.517	.003 ^a
	Residual	7.712	22	.351		
	Total	15.449	26			

a. Predictors: (Constant), Coupling Level, Average Block Cognitive Complexity, Selector Use inheritance Level, Nesting Factor

b. Dependent Variable: Testability

The correlation analysis results showed that $ABCC_{SCSS}$ metric is positively correlated with understandability, modifiability and testability which is contrary to the results of NF_{SCSS} , $SUIL$ and CL_{SCSS} which are negatively correlated with subjects rating of understandability, modifiability and testability.

7.8.2 Objective Results

This objective part of the experiment was performed by the subjects responding to the questions on understandability, modifiability and testability sections. The subjects were required to indicate the starting time and ending time i.e. indicate time before tackling questions on each section and ending time after completing the tasks on each section.

The understanding time, modifying time and testing time for each SCSS file was recorded. Three data sets were generated by computing the means for understanding time (see Appendix 4), modifying time (see Appendix 5) and testing time (see Appendix 6) for each SCSS file. These data sets represented the dependent variable, while the independent variable i.e metrics values was acquired through SCMT-SCSS tool.

7.8.2.1 Relationship between Metrics and Time to Understand

The correlation of SCSS metrics values with time taken to understand SCSS files is shown in Table 7.12. All the metrics are significantly correlated to the subjects understanding time of the SCSS code. The $ABCC_{SCSS}$ metric is negatively correlated with understandability as shown by the correlation coefficient value of -0.611, while NF_{SCSS} has a correlation coefficient value of 0.687 and $SUIL$ has a correlation coefficient value

of 0.611 at 99% confidence level. The CL_{SCSS} has a correlation coefficient of 0.386 at 95% confidence level.

Table 7.12: Correlation Results with Time to Understand

SCSS Metrics	Correlation Coefficient	Sig. (2-tailed)
ABCC _{SCSS}	-0.611**	0.001
NF _{SCSS}	0.687**	0.000
SUIL	0.611**	0.001
CL _{SCSS}	0.386*	0.047

**=99% confidence, *=95% confidence

The ANOVA results as shown in Table 7.13 indicate that the P-value (“Sig” for significance) is 0.000. This value is below the prescribed $P < 0.05$, meaning its statistically significant and the SCSS metrics means are not equal. Therefore, this is a strong indication against the null hypothesis and the study concludes that in overall the model can predict the understandability of the SCSS code.

Table 7.13: Understanding time significance with ANOVA

Model		Sum of Squares	df	Mean Square	F	Sig.
1	Regression	3756261.696	4	939065.424	23.684	.000 ^a
	Residual	872277.353	22	39648.971		
	Total	4628539.049	26			

a. Predictors: (Constant), Coupling Level, Average Block Cognitive Complexity, Selector Use Inheritance Level, Nesting Factor

b. Dependent Variable: Understandability with Time

The importance of components is shown in Table 7.14 , Component 1 comprises 70.6% of the proportion data variance while component 2 comprises 13.7% of the proportionate variance and cumulatively component 1 and 2 account for 84.3% of variance. Component 3 comprises 8.6% of the proportionate variance and cumulatively component 1,2 and 3 account for 92.9% of model variance. Component 4 comprises 5.1% of the proportionate variance and cumulatively component 1,2,3 and 4 account for 98.0% of variance. Component 5 comprises 2.0% of the proportionate variance and cumulatively component 1,2,3,4 and 5 account for 100% of the model variance. The selected level of model variation for acceptance of factors contributing to the model was 80%. The first and second component which achieved 84.3%, were chosen for understandability model variance.

Table 7.14: PCA for Understandability

Components	Comp.1	Comp.2	Comp.3	Comp.4	Comp.5
Proportion of Variance	0.7059	0.1367	0.0864	0.0510	0.0198
Cumulative Proportion	0.7059	0.8426	0.9291	0.9801	1

The researcher conducted PCA for understandability to determine which of the variables would be reduced or dropped in the model while retaining as much of the information in the model as possible. However, from the PCA extraction generated in component 1 and component 2, as shown in Table 7.15, none of the variables had value of zero. This implies that all the variables namely, ABCC_{scss}, NF_{scss}, SUIL and CL_{scss} are significant understandability model predictors.

The weights or loading of the various variables are shown in Table 7.15. Each of the weights are attributed to each of the component. Therefore, the principle component value for understandability was computed as follows:

$$PC1 = -0.371(ABCC) + 0.502(NF) + 0.418(SUIL) + 0.438(CL)$$

$$PC2 = -0.827(ABCC) + 0.203(NF) - 0.406(SUIL) - 0.374(CL)$$

Table 7.15: PCA Loadings for Understandability

Components	Comp.1	Comp.2	Comp.3	Comp.4	Comp.5
ABCC	-0.371	-0.827	0.415		
NF	0.502	0.203	0.373	-0.752	
SUIL	0.418	-0.406	-0.756	-0.294	
CL	0.438	-0.374	0.615	-0.482	0.241

7.8.2.2 Relationship between Metrics and Time to Modify

The correlation of SCSS metrics values with time taken to modify SCSS files is shown in Table 7.16. All the metrics are significantly correlated to the subjects modifying time of the SCSS files. The $ABCC_{SCSS}$ metric is negatively correlated with modifiability as shown by the correlation coefficient value of -0.415 at 95% confidence level, NF_{SCSS} has a correlation value of 0.633 at 99% confidence level, $SUIL$ has a correlation coefficient value of 0.472 at 95% confidence level and CL_{SCSS} has a correlation coefficient of 0.385 at 95% confidence level.

Table 7.16: Correlation Results with Time to Modify

SCSS Metrics	Correlation Coefficient	Sig. (2-tailed)
ABCC _{SCSS}	-0.415*	0.032
NF _{SCSS}	0.633**	0.000
SUIL	0.472*	0.013
CL _{SCSS}	0.385*	0.047

**=99% confidence, *=95% confidence

The ANOVA results are shown in Table 7.17, where P-value (“Sig” for significance) is 0.000. This value is below the prescribed $P < 0.05$, meaning its statistically significant and is a strong indication against null hypothesis. This study therefore concludes that in overall the model can predict the modifiability of the SCSS code.

Table 7.17: Modifying Time Significance with ANOVA

Model		Sum of Squares	Df	Mean Square	F	Sig.
1	Regression	1365953.575	4	341488.394	10.141	.000 ^a
	Residual	740837.406	22	33674.428		
	Total	2106790.981	26			

a. Predictors: (Constant), Coupling Level, Average Block Cognitive Complexity, Selector Use Inheritance Level, Nesting Factor

b. Dependent Variable: Modifiability with Time

The importance of components is shown in Table 7.18, Component 1 comprises 66.7% of the proportion data variance while component 2 comprises 13.8% of the proportionate variance and cumulatively component 1 and 2 account for 80.4% of variance. Component 3 comprises 9.1% of the proportionate variance and cumulatively component 1,2 and 3 account for 89.5% of model variance. Component 4 comprises 8.2% of the proportionate variance and cumulatively component 1,2,3 and 4 account for 97.7% of variance. Component 5 comprises 2.3% of the proportionate variance and cumulatively component 1,2,3,4 and 5 account for 100% of the model variance. Based on the selected level of model variation which is 80%, the first and second component were chosen for modifiability model.

Table 7.18: PCA for Modifiability

Components	Comp.1	Comp.2	Comp.3	Comp.4	Comp.5
Proportion of Variance	0.6665	0.1379	0.0905	0.0818	0.0231
Cumulative Proportion	0.6665	0.8044	0.8949	0.9768	1

The researcher conducted PCA for modifiability to determine which of the variables would be reduced or dropped in the model while retaining as much of the information in the model as possible. However, from the PCA extraction generated in component 1 and component 2, as shown in Table 7.19 none of the variables had zero result. This implies that all the variables namely, ABCC_{scss}, NF_{scss}, SUIL and CL_{scss} are significant modifiability model predictors.

The weights or loading of the various variables attributed to each of the component are shown in Table 7.19. Therefore, the principle component value for modifiability was computed as follows:

$$PC1 = -0.377(ABCC) + 0.518(NF) + 0.428(SUIL) + 0.443(CL)$$

$$PC2 = -0.79(ABCC) + 0.244(NF) - 0.413(SUIL) - 0.412(CL)$$

Table 7.19: PCA Loadings for Modifiability

Components	Comp.1	Comp.2	Comp.3	Comp.4	Comp.5
ABCC	-0.377	-0.79	0.254	0.397	-0.106
NF	0.518	0.244	-0.814		
SUIL	0.428	-0.413	0.43	-0.677	
CL	0.443	-0.412	-0.667	0.136	0.412

7.8.2.3 Relationship between Metrics and Time to Test

The correlation of SCSS metrics values with time taken to test SCSS files is shown in Table 7.20. All the metrics are significantly correlated to the subjects testing time of the SCSS files. The $ABCC_{SCSS}$ metric is negatively correlated with testability as shown by the correlation coefficient value of -0.584, NF_{SCSS} has a correlation value of 0.789 at 99% confidence level, $SUIL$ has a correlation coefficient value of 0.494, CL_{SCSS} has a correlation coefficient of 0.688. All the metrics were found to be correlating at 99% confidence level.

Table 7.20: Correlation Results with Time to Test

SCSS Metrics	Correlation Coefficient	Sig. (2-tailed)
ABCC _{SCSS}	-0.584**	0.001
NF _{SCSS}	0.789**	0.000
SUIL	0.494**	0.009
CL _{SCSS}	0.688**	0.000

**=99% confidence

The ANOVA results for testability of SCSS code indicates that the P-value (“Sig” for significance) was 0.000 as shown in Table 7.21. This value is below the prescribed $P < 0.05$, meaning that the defined metrics have no equal means, therefore forming a strong evidence against the null hypothesis. This study concludes that the metrics can predict the testability of the SCSS code.

Table 7.21: Testing Time Significance with ANOVA

Model		Sum of squares	Df	Mean Square	F	Sig.
1	Regression	1375713.471	4	343928.368	14.146	.000 ^a
	Residual	534887.009	22	24313.046		
	Total	1910600.480	26			

a. Predictors: (Constant), Coupling Level, Average Block Cognitive Complexity, Selector Use Inheritance Level, Nesting Factor

b. Dependent Variable: Testability with Time

The importance of components is shown in Table 7.22, Component 1 comprises 69.5% of the proportion data variance while component 2 comprises 14.0% of the proportionate variance and cumulatively component 1 and 2 account for 83.5% of variance. Component 3 comprises 6.8% of the proportionate variance and cumulatively component 1,2 and 3 account for 92.3% of model variance. Component 4 comprises 4.2% of the proportionate variance and cumulatively component 1,2,3 and 4 account for 96.5% of variance. Component 5 comprises 3.5% of the proportionate variance and cumulatively component 1,2,3,4 and 5 account for 100% of the model variance. Based on the selected level of model variation of 80%, the first and second component were chosen for testability model variance.

Table 7.22: PCA for Testability

Components	Comp.1	Comp.2	Comp.3	Comp.4	Comp.5
Proportion of variance	0.6950	0.1397	0.0884	0.0422	0.0346
Cumulative proportion	0.6950	0.8347	0.9231	0.9653	1

The researcher conducted PCA for testability to determine which of the variables would be reduced or dropped in the model while retaining as much of the information in the model as possible. However, from the PCA extraction generated in component 1 and component 2, as shown in Table 7.23 none of the variables had zero result. This implies that all the variables namely, ABCC_{scss}, NF_{scss}, SUIL and CL_{scss} are significant testability model predictors.

The weights or loading of the various variables attributed to each of the component are shown in Table 7.23. Therefore, the principle component value for modifiability can be computed as follows:

$$PC1 = -0.382(ABCC) + 0.498(NF) + 0.41(SUIL) + 0.451(CL)$$

$$PC2 = -0.787(ABCC) + 0.143(NF) - 0.459(SUIL) - 0.361(CL)$$

Table 7.23: PCA Loadings for Testability

Components	Comp.1	Comp.2	Comp.3	Comp.4	Comp.5
ABCC	-0.382	-0.787	0.227	0.405	-0.137
NF	0.498	0.143	-0.851		
SUIL	0.41	-0.459	-0.773	0.152	
CL	0.451	-0.361	0.547	-0.445	0.411

7.9 Discussion

The researcher investigated the relationship between SCSS metrics and the ratings by subjects for understandability, modifiability and testability. The researcher further investigated the relationship between the SCSS metrics and the subjects time to understand, time to modify, and time to test. The results indicated a strong correlation between the independent variables and dependent variables in the experiment. This implies that all the four proposed metrics can be regarded as good predictors of SCSS code maintainability.

The researcher found out that when $ABCC_{SCSS}$ metric value increases, the subjects rating of understandability, modifiability, testability, and subjects understanding time, modification time and testing time decreases. On the contrary, for the other three metrics namely NF_{SCSS} , $SUIL$ and CL_{SCSS} when their values increase, the subjects rating of understandability, modifiability, testability, and subjects understanding time, modification time and testing time increases. This means that when $ABCC_{SCSS}$ metric value increases, the complexity of the code decreases, implying that the time required to understand, modify and test SCSS code reduces, while when the NF_{SCSS} , $SUIL$ and CL_{SCSS} metric value increases the complexity of the SCSS code increase, thus increase in time required to understand, modify and test SCSS code..

The uniqueness of the $ABCC_{SCSS}$ metric means that its high value is desirable, hence making the SCSS code more understandable, modifiable and testable. However higher values of NF_{SCSS} , $SUIL$ and CL_{SCSS} are undesirable, because they make SCSS code more difficult to understand, modify and test.

7.9.1 Implications of Understandability Results

This section discusses the results based on subjects rating of understandability of SCSS code and subjects understanding time of SCSS code.

7.9.1.1 Relationship between Metrics and Understandability

Results showed strong correlation between the proposed metrics and subjects rating of understandability. Therefore, the metrics can be used as indicators of the understandability of SCSS code. The null hypothesis that there is no significant

correlation between the two variables was rejected and the alternative hypothesis that there is significant correlation between the metrics and subjects rating of understandability of SCSS code was accepted.

The ANOVA results based on the subjects rating of understandability of SCSS files confirmed that all the proposed metrics can be used to predict the understandability of SCSS code, meaning that all the proposed metrics didn't happen by chance.

7.9.1.2 Relationship between Metrics and Time to Understand

Results indicated a strong correlation between the metrics and time to understand. This experiment confirmed the results of the first experiment which had natural weakness introduced due to its subjectivity. The null hypothesis that there is no significant correlation between the two variables was rejected and the alternative hypothesis that there is significant correlation between the metrics and understanding time of SCSS code was accepted.

The ANOVA results based on the subjects understanding time of SCSS files further confirmed that all the proposed metrics can be used to predict the understandability of SCSS code, meaning that the metrics actually influence understandability as hypothesized. In addition, the principle component analysis results strongly indicated that all the proposed metrics are required to fully measure the understandability of SCSS code.

7.9.2 Implications of Modifiability Results

This section discusses the results based on subjects rating of modifiability of SCSS code and subjects modifying time of SCSS code.

7.9.2.1 Relationship between Metrics and Modifiability

The relationship between the SCSS metrics and modifiability was investigated. Results showed that all the metrics can serve as SCSS code modifiability predictors. The correlation coefficients showed that there is strong correlation between the metrics and modifiability. The null hypothesis that there is no significant correlation between the two variables was rejected and the alternative hypothesis accepted.

The ANOVA results based on the subjects rating of modifiability of SCSS files confirmed that all the proposed metrics can be used to predict the modifiability of SCSS code, meaning that all the proposed metrics didn't happen by accident.

7.9.2.2 Relationship between Metrics and Time to Modify

Results showed a strong correlation between the metrics and time to modify. This experiment confirmed the results of subjective data. The null hypothesis that there is no significant correlation between the two variables was rejected and the alternative hypothesis accepted.

The ANOVA results based on the subjects modifying time of SCSS files further confirmed that all the proposed metrics can be used to predict the modifiability of SCSS

code, meaning that the metrics actually influence the modifiability of SCSS code as hypothesized. Finally, the principle component analysis results indicate that the SCSS metrics $ABCC_{SCSS}$, NF_{SCSS} , $SUIL$ and CL_{SCSS} are necessary to fully measure or predict SCSS code modifiability.

7.9.3 Implications of Testability Results

This section discusses the results based on subjects rating of testability of SCSS code and subjects testing time of SCSS code.

7.9.3.1 Relationship between Metrics and Testability

The relationship between the SCSS metrics and testability was tested and results showed that the metrics can strongly predict the testability of SCSS code. The correlation coefficients showed that there is strong correlation between the metrics and testability. The null hypothesis that there is no significant correlation between the two variables was rejected and the alternative hypothesis accepted.

The ANOVA results based on the subjects testing time of SCSS files further confirmed that all the proposed metrics can be used to predict the testability of SCSS code, meaning that the metrics didn't happen by chance.

7.9.3.2 Relationship between Metrics and Time to Test

Results indicated a strong correlation between the metrics and time to test. This corroborated with the subjective rating of testability. The null hypothesis that there is no

significant correlation between the two variables was rejected and the alternative hypothesis accepted.

The ANOVA results based on the subjects testing time of SCSS files further confirmed that all the proposed metrics can be used to predict the testability of SCSS code, meaning that the metrics actually influence testability of SCSS code as hypothesized. In addition, the principle component analysis results indicate that all the proposed SCSS metrics $ABCC_{SCSS}$, NF_{SCSS} , $SUIL$ and CL_{SCSS} are key requirements to predict SCSS testability.

7.10 Effect of Moderating Variables on the Complexity-Maintainability Relationship

This study identified two moderating variables which are, number of years of experience and level of education. These variables were controlled, because all the subjects were undergraduate students in their fourth year and third year students. This means that all the subjects have relatively same level of experience in programming and software engineering field and have similar level of education. Therefore, no further analysis was done based on these variables.

7.11 Chapter Summary

This chapter presented an experiment that was carried out to investigate the relationship between the metrics and understandability, modifiability and testability (subjective data), and the relationship between the metrics and time to understand, modify and test (objective data). In the subjective part of the experiment, subjects were required to rate

the understandability, modifiability and testability of the provided SCSS file. The three data sets on understandability, modifiability and testability were generated and checked for correlation with the metrics values collected via metrics tool. Results for subjective experiment showed a strong correlation at 99% confidence level for NF_{SCSS} , $SUIL$ and CL_{SCSS} metrics and 95% confidence level for $ABCC_{SCSS}$ metric. The objective experiment also showed a strong correlation of $ABCC_{SCSS}$ metric, NF_{SCSS} , $SUIL$ metrics with understanding time at 99% confidence level and CL_{SCSS} metric at 95% confidence level. There was a strong correlation of $ABCC_{SCSS}$ metric and NF_{SCSS} with modifying time at 99% confidence level and $SUIL$ and CL_{SCSS} metrics at 95% confidence level. Results showed a strong correlation relationship between the metrics and testing time at 99% confidence level. This implies that the proposed metrics are strong SCSS code maintainability predictors. The ANOVA tests were carried out for both subjective and objective data. The results showed that the objective data is more reliable to predict the understandability, modifiability and testability of SCSS code in comparison to the subjective data. The PCA results strongly indicated that all the proposed metrics are required to predict the maintainability of SCSS code at 80% model variance.

CHAPTER EIGHT

SUMMARY, CONCLUSION AND RECOMMENDATIONS

8.1 Summary

This research aimed at defining SCSS structural complexity metrics that can be used to analyze the maintainability of SCSS code. Maintainability is an important characteristic of software and it determines the extent to which a software artifact can be understood, modified and tested. The maintainability of a code becomes difficult as software complexity increases. Therefore, to control SCSS complexity, four SCSS metrics were defined. These metrics are Average Block Cognitive Complexity for SCSS ($ABCC_{SCSS}$), Nesting Factor for SCSS (NF_{SCSS}), Selector Use Inheritance Level (SUIL) and Coupling Level for SCSS (CL_{SCSS}). The metrics were derived from SCSS attributes identified through the SCSS Structural Complexity Attributes Classification Framework. The SCSS metrics were theoretically and empirically validated. The results showed that there is a strong relationship between the metrics and subjective rating of SCSS code understandability, understandability with time, subjective rating of SCSS code modifiability, modifiability with time, subjective rating of SCSS code testability and testability with time. This implies that the four SCSS metrics are good predictors of maintainability of SCSS code.

8.2 Conclusion

This research aimed at defining valid SCSS structural complexity metrics that can then be useful in determining the maintainability of SCSS code. In order to achieve this, the research identified four specific objectives.

The first specific objective was to determine the attributes of SCSS code that affect its structural complexity. A literature review study was conducted and based on the gap identified which is lack of comprehensive framework to identify SCSS structural attributes, an SCSS structural complexity attributes framework was developed. The proposed framework was an extension of Muketha's structural complexity framework, which identified three types of attributes namely, intra-module attribute, inter-module attribute and hybrid attribute. The new framework identified a new category of attribute known as extra-module attribute. In addition, the framework extended every attribute category. The intra-module attribute was divided into size and control-flow complexity, the intra-module attribute was categorized into nesting and inheritance complexity, hybrid complexity identified one category known as association complexity, while the extra-module attribute has one category under it referred to as information flow complexity. This framework was presented to several SCSS experts to study it and provide feedback. The results proved that the proposed framework is relevant to SCSS complexity and that its able to identify all the SCSS structural complexity attributes

The second specific objective was to define metrics for measuring the structural complexity of SCSS code. It was found out that there are no existing metrics in literature that are suitable for SCSS complexity measurement. Therefore, four SCSS metrics were defined following the SCSS attributes identified in the proposed framework. These four metrics were, $ABCC_{SCSS}$, NF_{SCSS} , $SUIL$ and CL_{SCSS} . The $ABCC_{SCSS}$ metric measures the average cognitive complexity of all SCSS rule blocks in an SCSS file. NF_{SCSS} measures the extent to which rule nesting has been implemented in the SCSS code. $SUIL$ metric

measures the extent to which inheritance has been used in an SCSS code. Lastly, the CL_{SCSS} metric measures the extent to which rule blocks are coupled with each other. These metrics were theoretically validated with Weyuker's properties, and satisfied 7 out of 9 properties. Though not all properties were satisfied, it has been argued that for software metrics to be regarded as valid, they just need to satisfy most of the Weyuker's properties and not necessarily all. This implies that the proposed SCSS metrics are mathematically sound. The researcher further used Kaner framework to confirm the practicality of the metrics. This framework requires a response to its 11 questions, and all the questions were responded to positively and is a proof that the metrics can be applied to a real-life scenario.

The specific objective three was to develop a functional and usable metrics analysis tool for SCSS metrics computation. In the literature it was found that the development of a metrics tool is necessary to make the metrics to be appreciated by the software industry. When there are metrics without tool support, they end up forgotten and not useful for the software community. Therefore, a metrics tool referred to as SCMT-SCSS was developed and validated through an experiment. The results indicate that the tool is efficient, that is, it computes metrics values in much shorter time as compared to manual computation. This implies that the software designers and programmers will attain metrics values results in a shorter time and enable them make decisions regarding maintainability of code almost instantly. The results also indicate that the tool is effective and accurate, meaning the computed metric values are correct and can be relied on to make conclusions regarding the complexity of SCSS code. Further results showed that the tool was suitable for the tasks provided, meaning that the functions provided were good enough to aid in the

calculation of SCSS metrics values. Lastly, the results indicated that the tool is operable, meaning that the tool can be easily operated on to execute the tasks provided. This implies that the users of the tool can comfortably use it to compute metrics values.

The fourth and final specific objective in this study was to validate the structural complexity metrics for SCSS. A controlled laboratory experiment was carried out using between subjects design. The experiment was carried out in two phases to validate the SCSS metrics namely subjective phase and objective phase. From the experiment conducted, the results indicate that all the SCSS metrics highly correlate with maintainability sub-attributes of understandability, modifiability and testability. This means that all the SCSS metrics can be used to analyze SCSS maintainability. The ANOVA results in both subjective and objective parts of the experiment strongly indicate that the SCSS metrics (independent variables) influence the understandability, modifiability and testability (dependent variables) in the ANOVA analysis. The results obtained in both the objective and subjective ratings imply that the metrics can be taken to be good maintainability predictors for SCSS code. Therefore, the SCSS designers and programmers can use these metrics to measure and control SCSS complexity to achieve maintainable SCSS code.

8.3 Recommendations for Future Work

This study did not cover some aspects, though desirable because they didn't form part of the objectives of this research., this means that more research is required in future.

8.3.1 Define Metrics for other CSS Pre-Processors

In this research, four metrics were defined, and they are limited to SCSS language which is one of the SASS preprocessor syntaxes. This study didn't cover the .sass syntax, in addition there are many other preprocessors being used in the software industry such as less and stylus and there are no complexity metrics defined to measure them. Therefore, this research proposes that more new metrics to be defined to measure the structural complexity of CSS and its extensions. This field remains largely unexplored, as the literature showed that only few CSS metrics exist, and they have not been validated via theoretical framework to prove their compliance to principles of measurement theory. In addition, prior to this work there was no single metric proposed for CSS preprocessors.

8.3.2 Extending Structural Complexity Framework

This study proposes an extension of the SCACF-SCSS framework to accommodate the structural features of .sass syntax of SASS pre-processors and other CSS pre-processors languages.

8.3.3 Metrics Tool Extension to Recognize Multiple Languages

The researchers propose that the SCMT-SCSS tool to be upgraded to recognize the two syntaxes of SASS preprocessor i.e. .sass and .scss. This improvement of the tool will make it useful to all SASS programmers. Further improvements of the metrics tool could also accommodate CSS syntax and all its preprocessors.

8.3.4 Further Metrics Experimentation

This study proved that the proposed SCSS metrics are good for determination of SCSS maintainability. However, more experimental work will be required to be conducted with software industry experts. This will enhance the acceptability of the metrics by the SCSS experts and enable the establishment of credible threshold for maintainability of SCSS code.

REFERENCES

- Abreu, F., Goulo, M., & Esteves, R. (1995). Toward the design quality evaluation of object-oriented software systems. *Fifth International Conference on Software Quality*, (pp. 44-57). Austin, Texas, USA.
- Abreu, F.B. and Carapuca, R. (1994). Candidate Metrics for Object-Oriented Software within a Taxonomy Framework. *Journal of System Software*, Vol. 26, pp. 87–96.
- Abreu, Melo, & Abreu, F. B. (1996). Evaluating the impact of Object-Oriented Design on Software Quality. *Proceedings of 3rd International Software Metrics Symp.* Berlin.
- Adeyemi, A., Emebo, O., Misra, S., & Fernandez, L. (2015). Tool Support for Cascading Style sheets' Complexity Metrics. *Communications in Computer and Information Science*, 551-560.
- Adeyemi, A., Misra, S., & Ikhu-Omoregbe, N. (2012). Complexity Metrics for Cascading Style Sheets. In B. Murgante (Ed.), *Lecture Notes in Computer Science* (Vol. 7336, pp. 248-257). Springer.
- Albrecht, A. J. (1979). Measuring Application Development Productivity. *Proceedings of the Joint IBM/SHARE/GUIDE Application Development Symposium*, (pp. 83-92).
- Alghamdi, J. S., Rufai, R. A., & Khan, S. M. (2005, March). OOMeter: A software quality assurance tool. In *Ninth European Conference on Software Maintenance and Reengineering* (pp. 190-191). IEEE.
- Almugrin, S., Albattah, W., & Melton, A. (2016). Using indirect coupling metrics to predict package maintainability and testability. *Journal of systems and software*, 121, 298-310.
- Arar, Ö. F., & Ayan, K. (2016). Deriving thresholds of software metrics to predict faults on open source software: Replicated case studies. *Expert Systems with Applications*, 61, 106-121.
- Awode, T. R., Olatinwo, D. D., Shoewu, O., Olatinwo, S. O., Omitola, O. O., & Adedoyin, M. (2017). Halstead Complexity Analysis of Bubble and Insertion Sorting Algorithms.
- Babbie, E., & Rubin, A. (2008). *Research methods for social work*. California, USA: Thomson Brooks/Cole.

- Babu, P. C., Prasad, A. N., & Sudhakar, D. (2013, August). Software Complexity Metrics: A Survey. *International Journal of Advanced Research in Computer Science and Software Engineering*, 3(8), 1359-1362.
- Bagheri, E., & Gasevic, D. (2011). Assessing the maintainability of software product line feature models using structural metrics. *Software Quality Journal*, 19(3), 579-612.
- Bandi, R.K., Vaishnavi, V.K. and Turk, D.E. 2003. Predicting maintenance performance using object-oriented design complexity metrics, *IEEE Transactions on Software Engineering* 29: 77-87.
- Baroni, A. L., & Abreu, F. B. (2003, July). A formal library for aiding metrics extraction. In International Workshop on Object-Oriented Re-Engineering at ECOOP.
- Basci, D., & Misra, S. (2008). Entropy Metric for XML DTD Documents. *ACM SIGSOFT Software Engineering Notes*, 33(4).
- Basci, D., & Misra, S. (2009). Data Complexity Metrics for XML Web Services. *Advances in Electrical and Computer Engineering*, 9(2).
- Basci, D., & Misra, S. (2011a). Entropy as a Measure of Quality of XML Schema Document. *The International Arab Journal of Information Technology*, 8(1), 16-24.
- Basci, D., & Misra, S. (2011b). Metrics Suite for Maintainability of XML Web-Services. *IET Software*, 5(3), 320-341.
- 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.
- Basili, V. R.,(1992) Software modeling and measurement: the Goal/Question/Metric paradigm.
- Bhattacharjee, A. (2012). *Social Science Research: Principles, Methods, and Practices* (2nd ed.). Florida, USA: Textbooks Collection.
- Bieman, J. M., & Ott, L. M. (1994). Measuring functional cohesion. *IEEE transactions on Software Engineering*, 20(8), 644-657.
- Boehm, B. W., Brown, J. R., Kaspar, H., Lipow, M., McLeod, G., & Merritt, M. (1978). Characteristics of Software Quality. North Holland.

- Borade, J. G., & Khalkar, V. R. (2013). Software project effort and cost estimation techniques. *International Journal of Advanced Research in Computer Science and Software Engineering*, 3(8).
- Briand, L.C., Morasca, S., & Basili, V.R., (1996) "Property-Based Software Engineering Measurement".
- Bryaman, A. and Bell, E.(2007). *Business research methods* (15th ed.). Oxford: Oxford University Press.
- Bukhari, Z., Yahaya, J., and Deraman, A. (2015, August.)"Software metric selection methods: A review". In *Electrical Engineering and Informatics (ICEEI), 2015 International Conference on*, IEEE, pp. 433-438,
- Canfora, G., García, F., Piattini, M., Ruiz, F. & Visaggio, C.A.(2005,August).A family of experiments to validate metrics for software process models.*Journal of Systems and Software*.Volume 77, Issue 2, Pages 113-129.
- Cardoso, J. (2006). Complexity analysis of BPEL Web processes. *Software process: Improvement and Practice Journal*, 35-49.
- Catlin, H., & Catlin, M. L. (2011). *Pragmatic Guide to Sass*. (K. Keppler, Ed.) USA: The Pragmatic Programmers, LLC.
- Cederholm, D. (2013). *A BOOK APART: Sass for Web Designers*. (M. Brown, E. Kissane, J. Bolton, & T. Lee, Eds.) New York, USA: Jeffrey Zeldman.
- Charpentier, A., Falleri, J. R., & Réveillère, L., (2016) October. Automated Extraction of Mixins in Cascading Style Sheets. In *IEEE International Conference on Software Maintenance and Evolution (ICSME)*. pp56-66.
- Chawla, S. and Nath, R.(2013, July), "Evaluating Inheritance and Coupling Metrics", *International Journal of Engineering Trends and Technology (IJETT)*, vol.4, no.7, pp. 2903-2908.
- Cherniavsky, J. C., & Smith, C. H. (1991). On Weyuker's axioms for software complexity measures. *IEEE Transactions on Software Engineering*, 17(6), 636-638.
- Chhillar, U., & Bhasin, S. (2011). A New Weighted Composite Complexity Measure for Object-Oriented Systems. *International Journal of Information and Communication Technology Research*, 1(3), 101-108.
- Chidamber, S. R., & Kemerer, C. F. (1991, November). Towards a metrics suite for object-oriented design,. In *Object Oriented Programming Systems Languages and Applications*, 197-211.

- Chidamber, S. R., & Kemerer, C. F. (1994, June). A metrics suite for object-oriented design. *IEEE Transactions on Software Engineering*, 20(6), 467-493.
- Chung, C. and Lee, M.(1992), "Inheritance based Object-Oriented Software Metrics:," *IEEE Region 10 Conference*. Melbourne, Australia.
- Cohen, L., Manion. L., & Morrison, K. (2011). *Research methods in education*. London: Routledge.
- Creswell, J. W. (2014). The selection of a research approach. Research design: Qualitative, quantitative, and mixed methods approaches, 3-24.
- Curtis, B., Sheppard, S. B., Milliman, P., Borst, M. A., & Love, T. (1979). Measuring the psychological complexity of software maintenance tasks with the Halstead and McCabe metrics. *IEEE Transactions on software engineering*, (2), 96-104.
- Darcy, D. P., Slaughter, S., Kemerer, C. F.(2005). "The structural complexity of software: An experimental test". *IEEE Transactions on software engineering*, vol. 31, no. 11.
- Daud N. M. and Kadir W. M.(2014, September). "Static and Dynamic Classifications for SOA Structural Attributes Metrics", *Software Engineering Conference (MySEC), 2014 8th Malaysian*. IEEE, pp. 130-135.
- Debbarma, M. K., Debbarma, S., Debbarma, N., Chakma, K., & Jamatia, A. (2013, March). A Review and Analysis of Software Complexity Metrics in Structural Testing. *International Journal of Computer and Communication Engineering*, 2(2), 129-133.
- Dempsey, B. (2003) Target your brand. *Library journal*, 129(13):32.
- Denaro, G., Lavazza, L., & Pezze, M. (2003, November). An empirical evaluation of object oriented metrics in industrial setting. In The 5th CaberNet Plenary Workshop, Porto Santo, Madeira Archipelago, Portugal.
- Dhawan, S., & Kiran. (2012). Software Metrics – A Tool for Measuring Complexity. *International Journal of Software and Web Sciences (IJSWS)*, 2(1), 4-7.
- Dromey, R. G. (1995). "A model for software product quality," *IEEE Transactions on Software Engineering*, 21:146-162
- Dubey, S.K & Soumi Ghosh & Ajay Rana. (2012). "Comparison of Software Quality Models: An Analytical Approach," *International Journal of Emerging Technology and Advanced Engineering*, Volume 2, Issue 2, pp 111-119

- Dufour, B., Driesen, K., Hendren, L., & Verbrugge, C. (2003, October). Dynamic metrics for Java. *In ACM SIGPLAN Notices*, 38(11), 149-168.
- eAbreau, F.B., Carapuça, R.(1994,October) "Object-orientated software engineering: measuring and controlling the developmentprocess", Proc. 4th Int. Conf. On Software Quality, McLean, VA, USA.
- Easterbrook, S., Singer, J., Storey, M. A., & Damian, D. (2008). Selecting empirical methods for software engineering research. In *Guide to advanced empirical software engineering* (pp. 285-311). Springer, London.
- El Emam, K., Melo, W., & Machado, J. C. (2001). The prediction of faulty classes using object-oriented design metrics. *Journal of Systems and Software*, 56(1), 63-75.
- Falah, B., and Magel, K (2015). "Taxonomy Dimensions of Complexity Metrics". *Int'l Conf. Software Eng. Research and Practice*.
- Falessi, D., Juristo, N., Wohlin, C., Turhan, B., Münch, J., Jedlitschka, A., & Oivo, M. (2018). Empirical software engineering experts on the use of students and professionals in experiments. *Empirical Software Engineering*, 23(1), 452-489.
- Fenton N., and Pfleeger, S. L.(1997). "Software Metrics: A Rigorous and Practical Approach", *2nd Edition*, IT Publishing Company,.
- Fenton, N. (1994). Software measurement: a necessary scientific basis. *IEEE Transactions on Software Engineering*, 199-206.
- Fenton, N. and Bieman, J. (2014). "Software Metrics: A Rigorous and Practical Approach", *3rd Edition*, Chapman & Hall/CRC Innovations in Software Engineering and Software Development Series,.
- Frain, B. (2013). *Sass and Compass for Designers*. Birmingham, UK: Packt Publishing.
- Genero, M., Manso, E., & Cantone, G. (2003). Building UML Class Diagram Maintainability Prediction Models Based on Early Metrics. *Proc. 9th International Symposium on Software Metrics(METRICS '03)*, (pp. 263-275).
- Genero, M., Manso, E., Visaggio, A., Canfora, G., & Piattini, M. (2007). Building measure-based prediction models for UML class diagram maintainability. *Empirical Software Engineering*, 12(5), 517-549.
- Ghosheh, E., Black, S., & Qaddour, J. (2008). Design metrics for web application maintainability measurement. *IEEE/ACS International Conference on Computer Systems and Applications* (pp. 778-784). Doha: IEEE.

- Gill, N. S., & Sikka, S. (2011). Correlating Dimensions of Inheritance Hierarchy with Complexity & Reuse. *International Journal on Computer Science and Engineering (IJCSE)*, 3(9), 3250-3253.
- Gupta, R. (2015). LI metrics which predicts maintainability. *International Journal of Engineering Technology and Computer Research*, 3(3). Retrieved from <http://ijetcr.org/index.php/ijetcr/article/view/171>
- Gursaran. (2001). Viewpoint representation validation: a case study on two metrics from the Chidamber and Kemerer suite. *Journal of Systems and Software*, 59(1), 83-97.
- Halstead, M. H. (1977). *Elements of Software Science*. New York: Elsevier North-Holland.
- Hammersley, M. (2013). *What is Qualitative Research?* London and New York: Bloomsbury.
- Harrison, R., Counsell, S., & Nithi, R. (1997, July). An overview of object-oriented design metrics. In *Proceedings Eighth IEEE International Workshop on Software Technology and Engineering Practice incorporating Computer Aided Software Engineering* (pp. 230-235). IEEE.
- Harrison, W., Magel, K., Kluczny, R., & Dekok, A. (1982). Applying Software Complexity Metrics to Program Maintenance. *Communications of the ACM*, 15, 65-79.
- Henderson-Sellers, B.(1996).“Object Oriented Metrics: Measures of Complexity”, *Prentice Hall, Upper Saddle River, NJ*.
- Henley, C., (2015) *Better CSS with Sass. UK: Five Simple Steps. ISBN: 978-3-863730-81-9*
- Henry,S., and Kafura, D.(1984). The evaluation of software systems structure using quantitative software metrics. *Software:Practice and Experience*, 14(6), 561-573.
- Hissom, A. (2011). (MIT) Retrieved September 13, 2016, from <http://amyhissom.com/HTML5-CSS3/history.html>
- IEEE . (1998). *Std. 1061-1998 IEEE Computer Society: Standard for Software Quality Metrics Methodology*.
- IEEE Std. 610.12-1990. (1993). *Standard Glossary of Software Engineering Terminology*. Los Alamito, CA: IEEE Computer Society Press.
- Imenda, S. (2014). Is there a conceptual difference between theoretical and conceptual frameworks?. *Journal of Social Sciences*, 38(2), 185-195.ISBN: 013179292X.

ISO, International Organization for Standardization (2001) "ISO 9126-1:2001, Software engineering-Product quality, Part 1: Quality model",

ISO/ IEC CD 25010. (2008). Software Engineering: Software Product Quality Requirements and Evaluation (SQuaRE) Quality Model and guide. International Organization for Standardization, Geneva, Switzerland.

Kandpal, M., & Kandpal, A. (2012). Critical Analysis of Traditional Size Estimation Metrics for Object Oriented Programming. *International Journal of Computer Applications*, 58(13).

Kaner, C. (2004). Software Engineering Metrics:what do they measure and how do we know? *In:Proc. Tenth Int. Software Metrics Symp.,Metrics*, (pp. 1-10).

Kaur, S., & Maini, R. (2016). Analysis of various software metrics used to detect bad smells. *Int J Eng Sci (IJES)*, 5(6), 14-20.

Khan, A. A., Mahmood, A., Amralla, M. S., & Mirza, T. H. (2016, January). Comparison of Software Complexity Metrics. *International Journal of Computing and Network Technology*, 4(1), 19-26.

Kiewkanya, M., Jindasawat, N., & Muenchaisri, P. (2004). A Methodology for Constructing Maintainability Model of Object-Oriented Design. *Proc. 4th International Conference on Quality Software* (pp. 206-213). IEEE Computer Society.

Ko, A. J., Latoza, T. D., & Burnett, M. M. (2015). A practical guide to controlled experiments of software engineering tools with human participants. *Empirical Software Engineering*, 20(1), 110-141.

Kocaguneli, E., Tosun, A., Bener, A. B., Turhan, B., & Caglayan, B. (2009, July). Prest: An Intelligent Software Metrics Extraction, Analysis and Defect Prediction Tool. *In SEKE* (pp. 637-642).

Koh, T. W., Selamat, M. H., Ghani, A. A. A., & Abdullah, R. (2008). Review of complexity metrics for object oriented software products. *International Journal of Computer Science and Network Security*, 8(11), 314-320.

Kothari, C. R. (2004). *Research methodology: Methods and techniques*. New Age International.

Kumar, L., Naik, D.K. and Rath, S.K. (2015) 'Validating the effectiveness of object-oriented metrics for predicting maintainability', *Procedia Computer Science*, Vol. 57, pp.798–806

- Kumar, R. (2011). *Research methodology: A step-by-step guide for beginners*. Los Angeles: SAGE.
- Kushwaha, D. S., & Misra, A. K. (2006, September). Improved Cognitive Information Complexity Measure: A Metric that establishes Program Comprehension Effort. *SIGSOFT Software Engineering Notes*, 31(5), 1-7.
- Laird, L. M., & Brennan, M. C. (2006). *Software measurement and estimation: a practical approach* (Vol. 2). John Wiley & Sons.
- Landis, J. R., Koch, G. G., 1977. The measurement of observer agreement for categorical data. *Biometrics* 33(1), 159–174
- Li, E. Y. (1987). “A measure of program nesting complexity” *National Computer Conference*, San Luis Obispo, California, pp. 531-538.
- Li, W. (1998). Another metric suite for object-oriented programming. *Journal of Systems and Software*, 44(2), 155-162.
- Li, W., & Henry, S. (1993). Object-oriented metrics that predict maintainability. *The Journal of Systems and Software*, 23(2), 111-122.
- Libakova, N. M., & Sertakova, E. A. (2015). The method of expert interview as an effective research procedure of studying the indigenous peoples of the north.
- Lie, H. W., & Bos, B. (2005). *Cascading Style Sheets: Designing for the Web* (3rd ed.). Boston, MA, USA: Addison-Wesley Professional.
- Liehr P, Smith MJ 1999. Middle range theory: Spinning research and practice to create knowledge for the new millennium. *Advances in Nursing Science*, 21(4): 81-91.
- Lincke, R., Lundberg, J., & Löwe, W. (2008). Comparing software metrics tool. *In proceedings of the 2008 international symposium on Software testing and analysis* (pp. 131-142). ACM.
- Linos, P., Lucas, W., Myers, S., & Maier, E. (2007, November). A metrics tool for multi-language software. In *Proceedings of the 11th IASTED International Conference on Software Engineering and Applications* (pp. 324-329). ACTA Press.
- Littlefair, T. (2001). *An investigation into the use of software code metrics in the industrial software development environment*. Retrieved from <https://ro.ecu.edu.au/theses/1508>
- Lorenz, M. & Kidd, J. (1994). *Object-Oriented Software Metrics*. Prentice Hall.

- Lu, Y., Mao, X., & Li, Z. (2016). Assessing software maintainability based on class diagram design: A preliminary case study. *Lecture Notes on Software Engineering*, 4(1), 53.
- Madi, A., Zein, O. K., & Kadry, S. (2013). On the improvement of cyclomatic complexity metric. *International Journal of Software Engineering and Its Applications*, 7(2), 67-82.
- Maheswaran, K., & Aloysius, A. (2018a). Cognitive weighted inherited class complexity metric. *Procedia Computer Science*, 125, 297-304.
- Maheswaran, K., & Aloysius, A. (2018b). An Interface based Cognitive Weighted Class Complexity Measure for Object Oriented Design. *International Journal of Pure and Applied Mathematics*, 118(18), 2771-2778.
- Manso, M. E., Cruz-Lemus, J. A., Genero, M., & Piattini, M. (2008, September). Empirical validation of measures for UML class diagrams: A meta-analysis study. In *International Conference on Model Driven Engineering Languages and Systems* (pp. 303-313). Springer, Berlin, Heidelberg.
- Marco, T. D. (1982). *Controlling software projects*. New York: Prentice Hall.
- Marden, P. M., & Munson, E. V. (1999). Today's Style Sheet Standards: The Great Vision Binded. *Computer*.
- Martinsons, M., Davison, R., & Tse, D.,(1999) The balanced scorecard: a foundation for the strategic management of information systems. *Decision support systems*, Vol 25, No.1, pp71-88.
- Mazinanian, D. and Tsantalis, N. (2016, March). "An empirical study on the use of CSS preprocessors". In *2016 IEEE 23rd international conference on Software Analysis, Evolution, and Reengineering (SANER)*, pp.168-178.
- McCABE, T. J. (1976, December). A Complexity Measure. *IEEE Transactions on software engineering*, se-2(4), 308-320.
- McCall, J.A., Richards,P.K., and Walters, G.F.(1977) "Factors in Software Quality", Nat'l Tech Information Service, no. Vol.1,2 and 3,
- McGarry, J., Card, D., Jones, C., Layman, B., Clark, E., Dean, J., & Hall, F. (2002). *Practical software measurement: objective information for decision makers*. Boston: Addison-Wesley.
- Mens, T. (2016) "Research trends in structural software complexity". arXiv preprint arXiv:1608.01533.

- Mesbah, A., & Mirshokraie, S. (2012, June). Automated analysis of CSS rules to support style maintenance. In Proceedings of the 34th International Conference on Software Engineering (pp. 408-418). IEEE Press.
- Miguel, J. P., Mauricio, D., & Rodríguez, G. (2014). A review of software quality models for the evaluation of software products. arXiv preprint arXiv:1412.2977.
- Mishra, D. (2012). New Inheritance Complexity Metrics for Object-Oriented Software Systems: An Evaluation with Weyuker's Properties. *Computing and Informatics*, 30(2), 267-293.
- Mishra, S. and Sharma, A. (2015) 'Maintainability prediction of object oriented software by using adaptive network based fuzzy system technique', *International Journal of Computer Applications*, Vol. 119, No. 9
- Misra S. and Cafer, F.(2012, November) "Estimating Quality of JavaScript" *The International Arab Journal of Information Technology*, vol. 9, no.6, pp. 535-543.
- Misra, S. , Adewumi, A. , Fernandez-Sanz, L. & Damasevicius, R .,(2018) A Suite of Object Oriented Cognitive Complexity Metrics. *IEEE Access*, Vol. 6, pp8782-8796.
- Misra, S., & Cafer, F. (2012, November). Estimating Quality of JavaScript. *The International Arab Journal of Information Technology*, 9(6), 535-543.
- Misra, S., Akman, I., & Koyuncu, M. (2011, June). An inheritance complexity metric for object-oriented code: A cognitive approach. *Indian Academy of Sciences*, 36(3), 317-337.
- Mohajan, H. K. (2017). Two criteria for good measurements in research: Validity and reliability. *Annals of Spiru Haret University. Economic Series*, 17(4), 59-82.
- Morasca, S.(2015). "Rethinking Software Attribute Categorization". *6th International Workshop on Emerging Trends in Software Metrics*. IEEE. pp. 31-34,. DOI 10.1109/WETSoM.2015.8
- Morasca, S., and Briand, L. C.(1997, November) "Towards a theoretical framework for measuring software attributes", *In proceedings Fourth International Software Metrics Symposium*, IEEE, pp. 119-126.
- Mugenda, O. M., & Mugenda, A. G. (2003). *Research methods. Quantitative and Qualitative*.
- Muketha, G. M. (2011). "Size and Complexity Metrics as Indicators of Maintainability of Business Process Execution Language Process Models", *Doctoral dissertation*,

- Muketha, G.M., Ghani, A.A.A., Selamat, M.H. & Atan, R, (2010a) Complexity Metrics for Executable Business Processes. *Information Technology Journal*, Vol. 9, No. 7, pp1317-1326.
- Muketha, G. M., Ghani, A. A. A., Selamat, M. H., & Atan, R, (2010b) A Survey of Business Process Complexity Metrics. *Information Technology Journal*, Vol 9, No. 7, pp1336-1344.
- Myers, M.D. and Avison, D. (2002). Qualitative Research in Information Systems. *MIS Quarterly*. 22(2), 241-242.
- Naderifar, M., Goli, H., & Ghaljaie, F. (2017). Snowball sampling: A purposeful method of sampling in qualitative research. *Sdmej*, 14(3).
- Neelamegam, C., & Punithavalli, M. (2009). A survey-object oriented quality metrics. *Global Journal of Computer Science and Technology*, 9(4), 183-186.
- Netherland, W., Eppstein, C., Weizenbaum, N., & Mathis, B. (2013). *Sass and Compass in Action*. (S. Stirling, & A. Carroll, Eds.) New York, USA: Manning Publications Co.
- Nunnally, J. C. (2008)., *Psychometric theory* (2nd ed.). New York: McGrawHill
- Ogheneovo, E. E. (2014, December). On the Relationship between Software Complexity and Maintenance Costs. *Journal of Computer and Communications*, 2, 1-16.
- Parthasarathy, S., and Anbazhagan, N. (2006). Analyzing the software quality metrics for object oriented technology. *Inform. Technol. J*, 5, 1053-1057.
- Praveen, S., Agarwal, D., & Srivastava, A. (2018). Literature survey on object oriented function point: a reusability metrics. *International Journal of Advanced Research in Computer Science*, 9(Special Issue 2), 152.
- Ramasubbu, N. and Kemerer, C. F.(2012) “Structural Complexity and Programmer Team Strategy: An Experimental Test”, *IEEE Transactions on software engineering*, vol. 38, no. 5.
- Rea, A., & Rea, W. (2016). How Many Components should be Retained from a Multivariate Time Series PCA?. arXiv preprint arXiv:1610.03588.
- Remenyi D., SAGE Publications Ltd; 1 edition. *Doing Research in Business and Management* 1st Edition September 14, 1998
- Rietveld, T., & Van Hout, R. (2011). *Statistical techniques for the study of language and language behaviour*. Walter de Gruyter.

- Riguzzi, F.(1996). *A survey of software metrics*. Università degli Studi di Bologna.
- Rizvi, S. W., & Khan, R. A. (2010, April). Maintainability Estimation Model for Object-Oriented Software in Design Phase(MEMOOD). *Journal of Computing*, 2(4).
- Saini, S., Sharma, S., & Singh, R. (2015, December). Better utilization of correlation between metrics using Principal Component Analysis (PCA). In 2015 Annual IEEE India Conference (INDICON) (pp. 1-6). IEEE.
- Salman, I., Misirli, A. T., & Juristo, N. (2015, May). Are students representatives of professionals in software engineering experiments?. In 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (Vol. 1, pp. 666-676). IEEE.
- Saunders, M., Lewis, P. and Thornhill, A., 2012. *Research methods for business students*.6th edition, Pearson Education Limited.
- Serrano, M., Calero, C., Trujillo, J., Luján-Mora, S., & Piattini, M. (2004, June). Empirical validation of metrics for conceptual models of data warehouses. In International Conference on Advanced Information Systems Engineering (pp. 506-520). Springer, Berlin, Heidelberg.
- Shaik, A., Reddy, C. R. K., & Damodaram, A. (2012). Object oriented software metrics and quality assessment: Current state of the art. *International Journal of Computer Applications*, 37(11), 6-15.
- Sharma, N., Joshi, P., & Joshi, R. K. (2006). Applicability of Weyuker's Property 9 to object oriented metrics. *IEEE Transactions on Software Engineering*, 32(3), 209-211.
- Spinellis, D.: Tool writing: a forgotten art?, *IEEE Software*, vol. 22, (2005) 9-11
- Srinivasan, K.P. and Devi, T.(2014, November). Software metrics validation methodologies in software engineering. *International Journal of Software Engineering & Applications (IJSEA)*, Vol.5, No.6.
- Stevens, W. P., Myers, G. J., & Constantine, L. L. (1974). Structured design. *IBM Systems Journal*, 13(2), 115-139.
- Sullivan M.(2015), *Fundamentals of Statistics*, Upper Saddle River, NJ: Pearson Education, Inc., pp. 382-383.

- Tamayo, A., Granell, C., & Huerta, J., (2011) Analysing complexity of XML schemas in geospatial web services. *In Proceedings of the 2nd international conference on computing for geospatial research & applications*. ACM. pp17.
- Thaw, T., & Misra, S. (2013) Measuring the Reusable Quality for XML Schema Documents. *Acta Polytechnica Hungarica*, Vol. 10, No.4, pp87-106.
- Tomar, A. B., & Thakare, V. M. (2011). A systematic study of software quality models. *International Journal of Software Engineering & Applications*, 2(4), 61.
- Törn, A., Andersson, T. and Enholm, K. 1999. A complexity metrics model for software, *South African Computer Journal* 24: 40-48.
- Verner, J., & Tate, G. (1992). A software size model. *IEEE Transaction on Software Engineering*, 18(4).
- Vinobha, A., Velan, S., & Babu, C. (2014, May). Evaluation of reusability in aspect oriented software using inheritance metrics. In *2014 IEEE International Conference on Advanced Communications, Control and Computing Technologies* (pp. 1715-1722). IEEE.
- Weyuker, E. J. (1988). Evaluating software complexity measure. *IEEE Transaction on Software Engineering*, 14(9), 1357-1365.
- Whitfield, D., Ruddock M. and Bullman, R.(2008) “Expert opinion as a tool for quantifying bird tolerance to human disturbance”. *Journal of Biological Conservation*, vol. 141, pp 2708-2717.
- Wohlin, C., Runeson, P., Höst, M., Olsson, M.C., Regnell, B. and Wesslén, A. 2000. *Experimentation in Software Engineering: An Introduction*, Kluwer Academic Publishers.
- Wolf, A. (2009). Subjectivity, the researcher and the researched. *Operant Subjectivity: The International Journal of Q Methodology*, 32, 6-28.
- Yamane, T. (1967). *Statistics: An Introductory Analysis* (2nd ed.). New York: Harper and Row.

APPENDICES

**APPENDIX 1: EXPERT OPINION QUESTIONNAIRE FOR VALIDATING THE
STRUCTURAL COMPLEXITY ATTRIBUTES FRAMEWORK**

JOHN GICHUKI NDIA
P.O. BOX 75-10200
MURANG'A, KENYA.
Email: *ndiajg@gmail.com*

RE: LETTER OF INTRODUCTION

I am a PhD (Information Technology) student at **Masinde Muliro University of Science and Technology** Kenya carrying out a study on **Structural Complexity Attributes for Sassy Cascading Style Sheets**.

The aim of this questionnaire is to seek for your opinion as an expert concerning the **Relevance** and **Comprehensiveness** of the *attached* SASSY CSS COMPLEXITY ATTRIBUTE CLASSIFICATION FRAMEWORK. The attributes classification scheme is informed by the internal structure of Sassy CSS code (i.e. the elements within a rule-block and how they are related to each other).

NB: The data collected in this exercise is for research purposes only and will therefore be treated with strict confidentiality.

Kindly follow the link below to provide your responses.

Your participation in this study as a respondent is highly appreciated.

John Gichuki Ndia
Student Registration Number:
SIT/LH/004/2015

[Survey Link](#)

A. Personal Information

i) Please state your highest academic qualification.

- a) Bachelor’s Degree b) Master’s Degree c) PhD
 d) Other

ii) Kindly indicate your years of industry experience.

0 – 1	2 – 3	4- 5	6 – 7	Above 7

iii) How do you rank your knowledge of Software Engineering processes? Please tick appropriately.

Very Low	Low	Moderate	High	Very high

iv) How do you rank your knowledge of Sassy cascading style sheets (SCSS)? Please tick appropriately.

Very low	Low	Moderate	High	Very high

B. Relevance of the Classification Framework

i) It has been argued that SCSS code is more complex (i.e. it requires more time to understand and implement it) as compared to CSS because it has more features. This necessitates development of an attribute classification framework that will enable researchers to measure and control SCSS code complexity. Please indicate your level of agreement by ticking (✓) in the appropriate boxes

Don't agree	Slightly agree	Agree	Strongly agree	Very Strongly agree

- ii) The framework attached has been developed for the purpose of identifying the factors that contribute to Sassy CSS complexity. Please rate the extent to which you agree the framework is useful in identifying these factors.

Don't agree	Slightly agree	Agree	Strongly agree	Very Strongly agree

C. Comprehensiveness of the Classification Framework

- i) It has been argued that the following eight (8) SCSS features as presented in the table below and in the attached framework could increase SCSS code complexity (difficult to understand and modify) if overly and improperly used. Please indicate your level of agreement by ticking (✓) in the appropriate boxes.

	SCSS features	Don't agree	Slightly agree	Agree	Strongly agree	Very Strongly agree
1.	Global variables					
2.	Declarations /Attributes					
3.	Operators					
4.	Control directives					
5.	Functions					
6.	Mixins					
7.	Extend directives					
8.	Nesting					

ii) In your opinion, do you think the eight features captured in the table above have sufficiently covered all the possible complexity-causing factors in Sassy CSS?

a) Yes b) No

iii) If no, list any other features that in your opinion has not been covered.

NB: The SCSS Structural Complexity Attributes Classification Framework was attached with the guideline on interpretation of the framework

Thank you for your time and responses

APPENDIX 2: METRICS VALIDATION EXPERIMENT QUESTIONNAIRE

Purpose:

The purpose of this exercise is to investigate the relationship between your rating of complexity of each of the SCSS file provided and the SCSS metrics values. In addition the relationship between the understanding time, modification time and testing time of the files provided and the SCSS metrics values will be investigated.

Please answer ALL questions. There is no right or wrong answers. If you are unsure of some question, simply indicate your best from the provided options. You are required to tick (✓) the appropriate box where applicable. You will also be required to record certain measurements in the spaces provided.

Please read all questions carefully before answering. You are given two hours to complete your task. Please return the completed forms to me when you are through.

Note: The data collected in this exercise is for research purposes only, and will therefore be treated with strict confidentiality. The returned dully completed forms will be destroyed upon completion of the research project.

Thank you very much for participating in this study.

John Gichuki Ndia
PhD student
Department of Information Technology
Masinde Muliro University of Science and Technology

Please fill up the information below:

Name:

Programme of study:

Year of study:

Cell Phone No:

Email Address:

Background knowledge on Sassy Cascading Style Sheet Evaluation

Please complete this section by ticking (✓) as appropriate.

1. How many programming languages have you covered in your course of study?

- 0 – 2
 3 – 4
 5 – 6 years
 Above 7

2. How many software engineering courses have you taken?

- None
 1-4 courses
 More than 4 courses

3. Which features of Sassy Cascading Style Sheets (SCSS) can you comfortably use.

Global variables
 Mixins
 Nesting

Extends/Inheritance
 Functions
 Control directives

Operators
 Use of Declarations/attributes

Opinion on the understandability of each of the SCSS file provided

Write the time before you start to observe the SCSS Code File (starting time) in *hh:mm:ss*, and the time after you rate the SCSS Code File (ending time) in *hh:mm:ss*.

1. You are required to enter the name of the SCSS file attached to this question and then rate its ***understandability*** by ticking (✓) as appropriate.

Definition: *Understandability* is how easy it is to comprehend an SCSS code.

SCSS File No.	Very difficult (1)	Difficult (2)	Moderately difficult (3)	Easy (4)	Very easy (5)

Opinion on the modifiability of each of the SCSS file provided

2. You are required to enter the name of the SCSS file attached to this question and then rate its ***modifiability*** by ticking (✓) as appropriate.

Definition: *Modifiability* is how easy it is to incorporate changes to an SCSS code.

SCSS File No.	Very difficult (1)	Difficult (2)	Moderately difficult (3)	Easy (4)	Very easy (5)

Opinion on the testability of each of the SCSS file provided

3. You are required to enter the name of the SCSS file attached to this question and then rate its *testability* by ticking (✓) as appropriate.

Definition: *Testability* is how easy it is to identify errors or faults in an SCSS code.

SCSS File No.	Very difficult (1)	Difficult (2)	Moderately difficult (3)	Easy (4)	Very easy (5)

Understandability questions

Definition: *Understandability* is how easy it is to comprehend an SCSS code.

1. Write the time before you start to observe the SCSS Code File (starting time) in *hh:mm:ss*, and the time after you answer the questions (ending time) in *hh:mm:ss*.

Starting time (hh:mm:ss) _____

Answer the following questions in the space provided

- Identify one of the mixins defined and indicate how many times it has been used in the SCSS file?-----
- In which element(s) and/or selectors has inheritance been implemented?-----

Ending time (hh:mm:ss) _____

Modifiability questions

Definition: *Modifiability* is how easy it is to incorporate changes to an SCSS code.

2. Write the time before you start to observe the SCSS Code File (starting time) in *hh:mm:ss*, and the time after you answer the questions (ending time) in *hh:mm:ss*.

Starting time (hh:mm:ss) _____

Make the necessary changes to the SCSS file provided based on the following requirements.

You should write the SCSS codes in the space provided

- i. You are required to ensure all the links change to color green and are underlined when you hover on them.
- ii. You are required to define a mixin named ***Common*** that has three declarations/attributes i.e padding:2px, margin-left:5px and position:relative. The mixin should be included in all the existing body, heading 4, table, form and list element selectors in the SCSS file.

Ending time (hh:mm:ss) _____

Testability questions

Definition: *Testability* is how easy it is to identify errors or faults in an SCSS code.

3. Write the time before you start to observe the SCSS Code File (starting time) in *hh:mm:ss*, and the time after you answer the questions (ending time) in *hh:mm:ss*.

Starting time (hh:mm:ss) _____

Indicate the errors identified in the SCSS file.

You should write the SCSS codes in the space provided

- i. Identify the errors in the SCSS file based on the first global variable declared.
- ii. In the nesting of selectors, some situations require one to select parent selectors. The selection uses ampersand symbol (&). Identify in the SCSS file where the parent selector has not been well implemented.

Ending time (hh:mm:ss) _____

APPENDIX 3: SCSS Complexity Metrics Values

SCSS File No.	ABCC_{scss}	NF_{scss}	SUIL	CL_{scss}
1	3.01	1794	0.01	0.44
2	3.3	702	0	0
3	3.26	6466	0.04	0.29
4	2.17	15125	0	0.05
5	4.48	56	0.05	0.17
6	3.38	9890	0.09	0.12
7	2.04	33600	0.05	1.4
8	3.69	152	0	0.13
9	2.49	14097	0.06	0.22
10	3.18	2535	0.03	0.26
11	2.76	323	0	0
12	2.29	6958	0.06	0.2
13	2.33	36411	0.1	1.5
14	3.05	930	0	0.16
15	2.89	2511	0.03	0.07
16	1.89	44384	0.13	1.75
17	2.99	984	0	0.83
18	2.86	6348	0	0.16
19	2.36	11152	0.03	0.53
20	4.19	126	0	0
21	3.1	2482	0.06	1.33
22	2.46	7200	0.04	0.36
23	2.47	2050	0	0
24	2.69	5624	0.06	0.15
25	2.28	64	0.07	0.06
26	3.27	702	0	0
27	2.27	6958	0.06	0.2

APPENDIX 4: TIME TO UNDERSTAND SCSS FILE

SCSS File No.	Time to Understand SCSS File (Seconds)
1	417
2	474.5
3	955
4	1238
5	232.5
6	621
7	1306.5
8	416.67
9	1069.5
10	662.33
11	539.67
12	641
13	1504
14	365.67
15	832.67
16	1925
17	377
18	536.33
19	482
20	126
21	710.67
22	366
23	417
24	775
25	713
26	130
27	650

APPENDIX 5: TIME TO MODIFY SCSS FILE

SCSS File No.	Time to Modify SCSS File(Seconds)
1	567
2	636.5
3	967
4	1302.5
5	613
6	636.5
7	1024.5
8	658.67
9	903.33
10	813
11	539.67
12	903.33
13	1289
14	640
15	385.33
16	1410
17	354.67
18	567
19	567
20	232.5
21	640
22	715
23	420
24	760
25	620
26	823
27	758

APPENDIX 6: TIME TO TEST SCSS FILE

SCSS File No.	Time to Test SCSS File (Seconds)
1	571
2	670
3	953
4	1141.5
5	468.33
6	648.5
7	1197
8	689.67
9	955
10	775
11	485.33
12	775
13	1302.5
14	658.67
15	670
16	1306.5
17	847
18	518
19	934
20	222
21	867
22	866.33
23	420
24	760
25	613
26	583.33
27	1069.33

APPENDIX 7: EXPERIMENT QUESTIONNAIRE FOR VALIDATING THE METRICS TOOL

Purpose:

The purpose of this exercise is to investigate the efficiency, accuracy, functionality and usability of the Structural Complexity Metrics Tool (SCMT) for SCSS.

Please answer ALL questions. There is no right or wrong answers. If you are unsure of some question, simply indicate your best from the provided options. You are required to tick (✓) the appropriate box where applicable. You will also be required to record time taken to compute metrics values for SCSS files in the spaces provided.

Please read all questions carefully before answering. You are given one hour to complete your task. Please return the completed forms to me when you are through.

Note: The data collected in this exercise is for research purposes only, and will therefore be treated with strict confidentiality. The returned dully completed forms will be destroyed upon completion of the research project.

Thank you very much for participating in this study.

John Gichuki Ndia
PhD student
Department of Information Technology
Masinde Muliro University of Science and Technology

Please fill up the information below:

Name:

Programme of study:

Year of study:

Cell Phone No:

Email Address:

A. Manual Collection of SCSS metrics in the SCSS file provided

- Write the time before you start to count the metrics in SCSS File (starting time) in *mm:ss*, and the time after you count the metrics in the SCSS File (ending time) in *mm:ss*.

Starting time (mm:ss) _____

- Fill in the metrics value in the table based on the manual count of the metrics

FILE NO:		
Base Metrics		
S.No.	Metrics	Metrics Value
1	Number of Regular Attributes	
2	Number of Operators	
3	Number of Decision Nodes	
4	Number of function calls	
5	Number of Mixins Defined	
6	Number of Mixin calls	
7	Number of extend directives	
8	Number of selectors	
9	Number of SCSS Blocks	
10	Number of Variables Defined	
11	Number of Variables Instances	
Derived Metrics		
1	Average Block Cognitive Complexity	
2	Nesting Factor	
3	Selector Use Inheritance Level	
4	Coupling Level	

Ending time (mm:ss) _____

B. Automated Collection of SCSS metrics in the SCSS file provided

- Write the time before you start to count the metrics in SCSS File (starting time) in *mm:ss*, and the time after you count the metrics in the SCSS File (ending time) in *mm:ss*.

Starting time (mm:ss) _____

- Fill in the metrics value in the table based on the automated count of the metrics

FILE NO:		
Base Metrics		
S.No.	Metrics	Metrics Value
1	Number of Regular Attributes	
2	Number of Operators	
3	Number of Decision Nodes	
4	Number of function calls	
5	Number of Mixins Defined	
6	Number of Mixin calls	
7	Number of extend directives	
8	Number of selectors	
9	Number of SCSS Blocks	
10	Number of Variables Defined	
11	Number of Variables Instances	
Derived Metrics		
1	Average Block Cognitive Complexity	
2	Nesting Factor	
3	Selector Use Inheritance Level	
4	Coupling Level	

Ending time (mm:ss) _____

**OPINION ON SUITABILITY, ACCURACY AND OPERABILITY OF THE
SCSS METRICS TOOL**

a) How do you rate the Suitability of the SCSS-Metrics Tool?

Definition: Suitability is the capability of the tool to provide adequate set of functions for the tasks to be carried out.

Not Suitable	Slightly Suitable	Moderately Suitable	Suitable	Very Suitable
(1)	(2)	(3)	(4)	(5)

b) How do you rate the Accuracy of the SCSS Metrics Tool?

Definition: Accuracy is the capability of the tool to provide correct results

Not Accurate	Slightly Accurate	Moderately Accurate	Accurate	Very Accurate
(1)	(2)	(3)	(4)	(5)

c) How do you rate the Operability of the SCSS Metrics Tool?

Definition: The capability of the tool to allow the user to operate it

Not Operable	Slightly Operable	Moderately Operable	Operable	Very Operable
(1)	(2)	(3)	(4)	(5)

APPENDIX 8: MMUST RESEARCH AUTHORIZATION LETTER



MASINDE MULIRO UNIVERSITY OF SCIENCE AND TECHNOLOGY (MMUST)

Tel: 0702597360/61
: 0733120020/22
E-mail: deansgs@mmust.ac.ke
Website: www.mmust.ac.ke

P.O Box 190
50100 Kakamega
KENYA

Directorate of Postgraduate Studies

Ref: MMU/COR: 509079

16th February, 2018

John Gichuki Ndia
SIT/LH/004/2015
MMUST
P.O. Box 190-50100
KAKAMEGA

Dear Mr. Ndia ,

RE: APPROVAL OF PROPOSAL

I am pleased to inform you that the Directorate of Postgraduate Studies has considered and approved your Ph.D proposal entitled: *“Complexity Metrics For Analyzing The Maintainability of Sassy Cascading Style Sheets ”* and appointed the following as supervisors:

1. Prof. Geoffrey Muchiri Muketha - Department of Information Technology – Murang’a Uni.
2. Dr. Kelvin Kabeti Omieno - Department of Computer Science- MMUST

You are required to submit through your supervisor(s) progress reports every three months to the Director of Postgraduate Studies. Such reports should be copied to the following: Chairman, School of Computing Graduate Studies Committee and Chairman, Department of Information Science. Kindly adhere to research ethics consideration in conducting research.

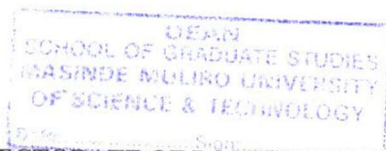
It is the policy and regulations of the University that you observe a deadline of three years from the date of registration to complete your Ph.D thesis. Do not hesitate to consult this office in case of any problem encountered in the course of your work.

We wish you the best in your research and hope the study will make original contribution to knowledge.

Yours Sincerely,

Prof: John Obiri

DIRECTOR DIRECTORATE OF POSTGRADUATE STUDIES



APPENDIX 9: NACOSTI RESEARCH LICENSE

THE SCIENCE, TECHNOLOGY AND INNOVATION ACT, 2013

The Grant of Research Licenses is guided by the Science, Technology and Innovation (Research Licensing) Regulations, 2014.



REPUBLIC OF KENYA

CONDITIONS

1. The License is valid for the proposed research, location and specified period.
2. The License and any rights thereunder are non-transferable.
3. The Licensee shall inform the County Governor before commencement of the research.
4. Excavation, filming and collection of specimens are subject to further necessary clearance from relevant Government Agencies.
5. The License does not give authority to transfer research materials.
6. NACOSTI may monitor and evaluate the licensed research project.
7. The Licensee shall submit one hard copy and upload a soft copy of their final report within one year of completion of the research.
8. NACOSTI reserves the right to modify the conditions of the License including cancellation without prior notice.



National Commission for Science, Technology and Innovation
RESEARCH LICENSE

National Commission for Science, Technology and Innovation
P.O. Box 30623 - 00100, Nairobi, Kenya

TEL: 020 400 7000, 0713 788787, 0735 404245

Email: dg@nacosti.go.ke, registry@nacosti.go.ke

Website: www.nacosti.go.ke

Serial No.A 23493

CONDITIONS: see back page

APPENDIX 10: NACOSTI RESEARCH AUTHORIZATION LETTER



NATIONAL COMMISSION FOR SCIENCE, TECHNOLOGY AND INNOVATION

Telephone: +254-20-2213471,
2241349,3310571,2219420
Fax: +254-20-318245,318249
Email: dg@nacosti.go.ke
Website : www.nacosti.go.ke
When replying please quote

NACOSTI, Upper Kabete
Off Waiyaki Way
P.O. Box 30623-00100
NAIROBI-KENYA

Ref. No. **NACOSTI/P/19/79922/28378**

Date: **12th March, 2019**

John Gichuki Ndia
Masinde Muliro University of Science and Technology
P. O Box 190-50100
KAKAMEGA

RE: RESEARCH AUTHORIZATION

Following your application for authority to carry out research on "*Complexity metrics for analyzing the maintainability of Sassy Cascading Style Sheets*" I am pleased to inform you that you have been authorized to undertake research in **Murang'a and Nairobi Counties** for the period ending **11th March, 2020**.

You are advised to report to **the County Commissioners and the County Directors of Education, Murang'a and Nairobi Counties** before embarking on the research project.

Kindly note that, as an applicant who has been licensed under the Science, Technology and Innovation Act, 2013 to conduct research in Kenya, you shall deposit **a copy** of the final research report to the Commission within **one year** of completion. The soft copy of the same should be submitted through the Online Research Information System.


DR. STEPHEN K. KIBIRU, PhD.
FOR: DIRECTOR-GENERAL/CEO

Copy to:

The County Commissioner
Murang'a County.

The County Director of Education
Murang'a County.

National Commission for Science, Technology and Innovation is ISO9001:2008 Certified

APPENDIX 11: MUT DATA COLLECTION RESEARCH PERMIT



**MURANG'A UNIVERSITY OF
TECHNOLOGY**
Office of the Registrar
(Administration & Planning)

Cell: 254-0772454938
E-mail: registrar@mut.ac.ke,
Website: www.mut.ac.ke

P.O.Box 75 - 10200,
Murang'a, Kenya,

Ref: MUT/GC/REG.AP/45/2016/VOL.1

18th April, 2019

John Ndia
C/o School of Computing and Technology
PO Box 75-10200
Murang'a

Dear Mr. Ndia,


RE: PERMISSION TO COLLECT RESEARCH DATA

Your letter dated 13th March, 2019 refers.

Your request to collect data at Muranga University of Technology on "Complexity Metrics for Anlayzing the Maintainability of Sassy Cascading Style Sheets" has been granted.

Kindly treat the information you will obtain in strict confidence and thereafter share the findings of your study with us for information purposes.

Yours sincerely,


JOSEPH GACHANJA
Ag. REGISTRAR (A&P)

Copy to: Vice Chancellor
Deputy Vice Chancellor (F&D)
Deputy Vice Chancellor (ASA)
Registrar (ASA)



MUT IS ISO 9001:2015 CERTIFIED

APPENDIX 12: PUBLICATIONS

The following papers have been published from this thesis

1. Ndia, J. G., Muketha, G. M., & Omieno, K. K. (2019, May). A Survey of Cascading Style Sheets Complexity Metrics. *International Journal of Software Engineering & Applications (IJSEA)*, 10(3), 21-33. <http://dx.doi.org/10.2139/ssrn.3405783>
2. Ndia, J. G., Muketha, G. M., & Omieno, K. K. (2019). Complexity Metrics for Sassy Cascading Style Sheets. *Baltic Journal of Modern Computing*, 7(4), 454-474. <https://doi.org/10.22364/bjmc.2019.7.4.01>