ESTIMATING COMPLEXITY OF A SOFTWARE CODE

A MASTER'S THESIS

in

Software Engineering

Atılım University

by

FERİD CAFER

JUNE 2010

GCRIS

ESTIMATING COMPLEXITY OF A SOFTWARE CODE

A THESIS SUBMITTED TO

THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES

OF

ATILIM UNIVERSITY

BY

FERİD CAFER

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

IN

THE DEPARTMENT OF SOFTWARE ENGINEERING

JUNE 2010

GCRIS

Approval of the Graduate School of Natural and Applied Sciences, Atılım University.

_____

Prof.Dr. İbrahim Akman

Director

I certify that this thesis satisfies all the requirements as a thesis for the degree of Master of Science.

_____

Prof. Dr. Ali Yazıcı

Head of Department

This is to certify that we have read the thesis Estimating Complexity of a Software Code submitted by Ferid Cafer and that in our opinion it is fully adequate, in scope and quality, as a thesis for the degree of Master of Science.

(ABROAD)

_____	_____
Assist. Prof. Dr. Sanjay Misra	Prof. Dr. K. İbrahim Akman
Co-Supervisor	Supervisor

Examining Committee Members

Prof. Dr. Ali Yazıcı	_____
Asst. Prof. Dr. Atila Bostan	_____
Asst. Prof. Dr. Nergiz Ercil Çağıltay	_____
Dr. Ali Arifoğlu	_____

Date: 18 June 2010

GCRIS

I declare and guarantee that all data, knowledge and information in this document has been obtained, processed and presented in accordance with academic rules and ethical conduct. Based on these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last name: Ferid Cafer

Signature:

GCRIS

ABSTRACT

ESTIMATING COMPLEXITY OF A SOFTWARE CODE

Cafer, Ferid

M.S., Software Engineering Department

Supervisor: Prof. Dr. K. İbrahim Akman

Co-Supervisor: Assist. Prof. Dr. Sanjay Misra

June 2010, 94 pages

This thesis study investigated the comprehensibility of a software code from a developer's point of view and proposed new metrics accordingly. The factors that affect the complexity of procedural, Object-Oriented, and multi-paradigm codes were analysed for this purpose. Addition to the investigated factors, various metrics and several aspects were combined in the proposed metrics. The proposed metrics were empirically validated in different paradigms.

Keywords: Code quality, Software complexity measurement

GCRIS

ÖZ

YAZILIM KODUNUN KARMAŞIKLIĞININ DEĞERLENDİRİLMESİ

Cafer, Ferid

Yüksek Lisans, Yazılım Mühendisliği Bölümü

Tez Yöneticisi: Prof. Dr. K. İbrahim Akman

Ortak Tez Yöneticisi: Yrd. Doç. Dr. Sanjay Misra

Haziran 2010, 94 sayfa

Bu tez calışması yazılım kodunun anlaşılırlığını programcı bakış açısıyla incelemiştir ve bu bağlamda yeni ölçevler sunmuştur. Bu amaçla, prosedürel, nesneye dayalı ve çoklu paradigmalı programlama dillerindeki karmaşıklık etkenleri araştırılmıştır. Bulunan ögelere ek olarak çeşitli ölçev ve farklı bakış açılarına dayandırılarak bir grup ölçev sunulmuştur. Sunulan ölçevlerin geçerliliği deneysel yöntemlerle test edilmiştir.

Anahtar Kelimeler: Kod kalitesi, Yazılım karmaşıklığı ölçümü

GCRIS

To My Supervisor and Co-supervisor

GCRIS

ACKNOWLEDGMENTS

I express sincere appreciation to my supervisor Prof. Dr. K. Ibrahim Akman for his guidance and insight throughout the research. Thanks also go to my co-supervisor Assist. Prof. Dr. Sanjay Misra. I am extremely grateful to my supervisor and co-supervisor for both their technical guidance and fatherly manner.

GCRIS

TABLE OF CONTENTS

ABSTRACT 5
ÖZ 6
ACKNOWLEDGMENTS 8
TABLE OF CONTENTS 9
LIST OF TABLES 11
LIST OF FIGURES 12
LIST OF ABBREVIATIONS 13
CHAPTER 1 14
INTRODUCTION 14
1.1 Introduction 14
1.2 Literature Review 17
1.2.1 Popular Metrics for Procedural Languages 18
a. Cyclomatic Complexity 18
b. Halstead Complexity Measures 19
c. Lines of Code 20
1.2.2 Some OO Metrics 20
a. The CK Metrics Suite 20
b. Weighted Class Complexity 21
1.3 Purpose and Scope of Research 22
CHAPTER 2 25
SOFTWARE QUALITY AND COMPLEXITY METRICS 25
2.1 Introduction 25
2.2 Classification of Languages 26
2.3 Quality of Software 27
2.4 Complexity Metrics 27
CHAPTER 3 29
MULTI-PARADIGM LANGUAGES 29
3.1 Introduction 29
3.2 Multi-Paradigm Languages 30
3.3 The Need for a New Metric 31
CHAPTER 4 33
PROPOSED METRICS AND THEIR IMPLEMENTATIONS 33
4.1 Introduction 33
4.2 The Proposed Metric 34
4.3 Demonstration of the Metric 40
4.4 More Examples 48
CHAPTER 5 57
EXTENDING THE METRIC 57
5.1 Introduction 57
5.2 Multi-Paradigm Complexity Measurement 57
5.3 Demonstration of the Metric 65
5.4 Comparative Study 71

5.5 Empirical Validation 73
CHAPTER 6 81
CONCLUSIONS AND FUTURE WORK 81
REFERENCES 85
APPENDICES 91
Appendix A 91
Web Sites of JavaScript Codes 91
Appendix B 93
Example Code in C 93
Appendix C 97
Empirical Validation 97
Appendix D 106
Other JavaScript Codes 106
Appendix E 107
Terms 107

GCRIS

LIST OF TABLES

Table 1: Multi-Paradigm Languages....30
Table 2: Nested Conditions 1....37
Table 3: Nested Conditions 2....37
Table 4: Nested Loops....37
Table 5: Basic Control Structures....39
Table 6: Example 1 [84]....41
Table 7: Example 2....42
Table 8: Example 3 [85]....43
Table 9: Example 4....44
Table 10: Comparison of Metrics....47
Table 11: Examples....49
Table 12: BCS for MCM....58
Table 13: Function Point....63
Table 14: Class Complexity of Shapes in Python....67
Table 15: Procedural Complexity of Shapes in Python....67
Table 16: Class Complexity of Shapes in C++....67
Table 17: Procedural Complexity of Shapes in C++....68
Table 18: Class Complexity of Shapes in Java....68
Table 19: Procedural Complexity of Shapes in Java....68
Table 20: Function Point Calculation of Shapes....70
Table 21: Comparison between Metrics....71
Table 22: Pyso – Classes....74
Table 23: Pyso – Procedural....74
Table 24: Pyso – Inherited Classes....76
Table 25: Pyso – FP....77
Table 26: Comparison of Projects....80
Table 27: Chat Application – Classes....97
Table 28: Chat Application – Cprocedural....97
Table 29: Chat Application – FP....97
Table 30: Microprocessor Simulator – Classes....99
Table 31: Microprocessor Simulator – Cprocedural....99
Table 32: Microprocessor Simulator – FP....100
Table 33: Medical System – FP....101
Table 34: NeoMem – Classes....102
Table 35: NeoMem – FP....102
Table 36: TreeMaker – Classes....104
Table 37: TreeMaker – FP....104
Table 38: Other Scripts....106

LIST OF FIGURES

Figure 1: Cyclomatic Complexity Example....18
Figure 2: Condition (1) Figure 3: Nested Condition (2) Figure 4: Nested Condition (3) 36
Figure 5: Loop (1) Figure 6: Nested Loop (2) Figure 7: Nested Loop (3) 37
Figure 8: try-catch 1 Figure 9: try-catch 2 Figure 10: try-catch 3 Figure 11: try-catch 4 38
Figure 12: Flow Graph of Example 4 47
Figure 13: Comparison between eLOC and PCCM....51
Figure 14: Comparison between CC and PCCM 52
Figure 15: Relative Graph between eLOC, CC and PCCM....53
Figure 16: Relative Graph between Time and PCCM 54
Figure 17: Relative Graph between eLOC, PCCM and Volume....54
Figure 18: Relative Graph between CC, PCCM and Difficulty....55
Figure 19: Relative Graph between eLOC, CC, PCCM, Effort and Time 56
Figure 20: Shapes – Class Diagram 66
Figure 21: Comparison of Metrics 73
Figure 22: Pyso – Inheritance 1 75
Figure 23: Pyso – Inheritance 2 76
Figure 24: Microprocessor – Inheritance 99

GCRIS

LIST OF ABBREVIATIONS

ISO	-	International Standardisation Organisation
IEEE	-	Institute of Electrical and Electronics Engineers
OO	-	Object-Oriented
BCS	-	Basic Control Structures
QA	-	Quality Assurance
FP	-	Function Point
PCCM	-	Procedural Cognitive Complexity Measure
MCM	-	Multi-paradigm Complexity Measurement
CQ	-	Code Quality
ANV	-	Arbitrarily Named Variable
MNV	-	Meaningfully Named Variable
CWU	-	Cognitive Weight Unit
CC	-	McCabe's Cyclomatic Complexity
LOC	-	Lines of Code
W	-	Weight
E	-	Halstead's Effort Estimation
T	-	Halstead's Time Estimation
D	-	Halstead's Difficulty Estimation
V	-	Halstead's Volume Estimation

GCRIS

CHAPTER 1

INTRODUCTION

1.1 Introduction

Software development involves creating a software system depending on requirements. Requirements are usually complex and this situation makes software projects change continuously. Software projects are changed or modified in order to better understand the user requirements or eliminate errors. Hence, software systems are called to be complex [58][10][53].

Software life cycle is the process of developing and changing software systems. A software life cycle consists of all the activities and products that are needed to develop a software system. Due to the fact that software systems are complex, life cycle models tend to enable developers to cope with software complexity. Life cycle models expose the software development activities and their dependencies in order to make them more visible and manageable [58][10][53].

A computer program is a set of instructions developed to perform a task, whereas a software system is a set of programs developed with an engineering discipline under consideration of quality with an aim to accomplish many tasks properly. The main distinguishing factor is quality [10]. Thereby, quality is the indispensable fact of a software system.

ISO [67] defines the concept of quality with six characteristics. If a software system is functional, reliable, usable, efficient, maintainable, and portable, then it is said to be of high quality. Another definition given by Software QA and Testing Resource Centre [66] is that quality software should have the least amount of bugs, be delivered on time and within a budget, meet requirements and be maintainable. IEEE definition is the degree to which a system component or process meets specified requirements [18].

Software systems are complex. Therefore, it is hard to attain a high level of quality. Software metrics have always been an important tool since it was realised that software development is a complex task. Due to its complexity, software quality has been a rising demand for decades and some definitions have been manifested throughout software history. A software product should carry several quality attributes, such as correctness, reliability, efficiency, integrity, usability, maintainability, testability, flexibility, portability, reusability, and interoperability [53]. According to Sommerville [69] the most necessary software quality attribute is maintainability. To efficiently be able to maintain a software system, the codes should be understandable for developers. Briefly, to achieve high quality, reduction of complexity is essential. To deal with software complexity, software metrics are used. Metrics are indicators of complexity; they expose several weaknesses of a complex software system. Therefore, by the means of software metrics, quality can be estimated. That is why metrics take an indispensable role in software development life cycle. Software complexity metrics are used to quantify a variety of software properties. Usually, it is extremely hard to build high quality software or improve the development process without using any metrics. There are a number of metrics each focusing on different complexity factors [68]. Large companies such as Hewlett-Packard, AT&T, and Nokia use several metrics to estimate the quality of their software systems [69].

The motivation for developing and using metrics is the need for quantitative and analytical approaches in general. Software metrics are amongst the measurement-based techniques which can be used to improve software development processes and software products [27]. They provide quantitative information about the development and the validation of software development processes [47]. DeMarco [20] describes the reason for using software metrics as “you cannot manage what you cannot measure.” In order to monitor and improve software quality, measurement is essential.

Software metrics tend to compare various parameters such as cost, effort, time, maintenance, understanding and reliability. Metrics are indispensable from several aspects such as measuring the comprehension of a code, testability of the software, maintainability and development processes [19].

McCabe et al. [44] define software complexity as “one branch of software metrics that is focused on direct measurement of software attributes, as opposed to indirect software measures such as project milestone status and reported system failures.”

Basili [7] defines complexity as a measure of resources used by a software system during the interaction of the parts of the software, to perform a task. If the interacting entity is a computer, then complexity is related to the execution time and hardware resources required to perform the task. If the interacting entity is a programmer, then complexity is related to the difficulty of coding, testing and modifying the software [36]. It is believed that for coding and modifying a software system, a higher comprehensibility of the code is required. If the comprehensibility is higher, then the complexity of the software is lower, and thus testing is easier.

Sommerville [69] categorises metrics as control and predictor metrics. Control metrics are related to software processes whereas predictor metrics are associated with software products. Control metrics estimate effort, time and defects. On the other hand, predictor metrics assess the number of attributes and the structures in a code. According to this definition, this thesis focuses on predictor metrics.

Related with the definitions above, the thesis proposal classifies software complexity metrics as:

1. Hardware resource allocation
2. Paradigms
    a. procedural paradigm
    b. Object-Oriented paradigm
    c. multi-paradigm
    d. other paradigms

The complexity factors and metrics that are used in this research are associated with procedural paradigm, OO paradigm, and multi-paradigm (a combination of the previous paradigms).

Procedural programming languages [81], which are also known as imperative programming languages, are based on structures. OO programming languages support hierarchical data abstractions. A problem may require various concepts to be solved practically which lead to multi-programming paradigm [75]. A multi-paradigm language may support two or more programming paradigms. One of the main benefits of a multi-paradigm language is providing easier transitions between paradigms [81]. Under the scope of the proposal, by multi-paradigm, the combination of procedural and OO paradigms is meant. However, in fact, a multi-paradigm is not restricted with only those two paradigms.

In the next chapter, software quality and complexity metrics are expressed in greater detail.

1.2 Literature Review

There are lots of metrics that have been developed until now. Each metric has its own advantages and disadvantages. Metrics may tend to measure quality, size, complexity, requirements, effort, productivity, cost and schedule, scrap and rework, and support [68].

Typically the metrics can be classified as follows [68]:

- *Technical metrics*: are used to measure the structure of the code, external characteristics, manuals and documentation.
- *Defect metrics*: measure defects of software. For instance, the number of defects found in a specific time.
- *End-user satisfaction*: is based on the value received from using the system.
- *Warranty metrics*: focus on revenues and expenditures related with correcting the software defects.
- *Reputation metrics*: are related with user satisfaction.

Although, metrics can be classified variously, this thesis categorises them according to paradigms.

1.2.1 Popular Metrics for Procedural Languages

Most of the tools used for measuring code complexity use Lines of Code, Cyclomatic Complexity and Halstead Complexity Measure [21][45].

a. Cyclomatic Complexity

McCabe's Cyclomatic Complexity formula is given below:

$$m = e - n + 2p \quad (1)$$

Where,

m is the cyclomatic complexity

e is the number of edges

n is the number of vertices

p is the connected components

For example, in Figure 1: e=5, n=4, p=1

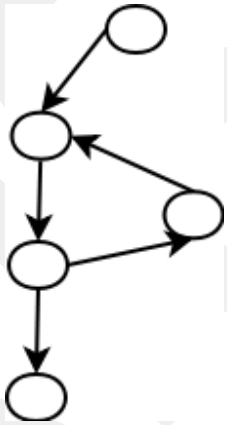


Figure 1: Cyclomatic Complexity Example

m = 5 – 4 + 1

m = 2

McCabe's Cyclomatic Complexity (CC) is older than 30 years. It was used very often in the past [43]. Measuring CC of a code is like making a basis path testing. In other words, it has an advantage of measuring the flow of a program. Thus, it can measure the complexity of an algorithm. However, it is not sufficient to measure code complexity especially of modern programming languages [42]. Today's programming languages carry functions which decrease the burden of a programmer, inside its library. There are lots of short cuts in Java, Python, and some others. For instance, a code written in C or Python will make a big difference in a point of comprehensibility. In other words, the same program written in Python will be more readable and probably simpler, and for this reason it will be easier to develop. Hence, CC is not able to go beyond the algorithm and measure its cognitive aspect. Moreover, the factors that affect the code complexity does

not merely consist of algorithm, but also variables, structures, classes, coupling, cohesion.

b. Halstead Complexity Measures

This metric was presented by Halstead in 1977 [76][78]. The method includes

- n1: the number of distinct operators,
- n2: the number of distinct operands,
- N1: the total number of operators, and
- N2: the total number of operands.

Program Length	=>	$N = N1 + N2$
Vocabulary Size	=>	$n = n1 + n2$
Program Volume	=>	$V = N * \log 2(n)$
Difficulty Level	=>	$D = (n1/2) * (N2/n2)$
Program Level	=>	$L = 1/D$
Effort to Implement	=>	$E = V * D$
Time to Implement	=>	$T = E/18$
Number of Delivered Bugs	=>	$B = E^{(2/3)}/3000$

By using n1, n2, N1, N2, several outputs are generated, namely, program length, vocabulary size, program volume, difficulty level, program level, effort to implement, time to implement, and number of delivered bugs.

This metric is an easy way of measuring a code from many different angles. However, later some questions were aroused such as what was an operator and operand. It was difficult to differentiate them. There is no a sharp distinction between operators and operands. That was one of the major problems. Another issue to be considered is the

structure, inheritance, objects, and so on. The Halstead method is not capable of measuring the structure of a code, inheritance, interactions between modules, and so on [33]. Moreover, as mentioned before, it is based on some psychological assumptions. This fact breeds haze about the objectivity of the metric. However, limited amount of subjectivity is particular to cognitive aspect, since cognitive measurement considers comprehensibility of human being.

c. Lines of Code

This metric considers on the number of lines of code inside a program. It has some types [57]:

- Lines of Code (LOC): counts every line including comments and blank lines.
- Kilo Lines of Code (KLOC): it is LOC divided by 1000.
- Effective Lines of Code (eLOC): estimates effective line of code excluding parenthesis, blanks and comments.
- Logical Lines of Code (lLOC): estimates only the lines which form statements of a code. For example, in C, the statements which end with semi-colon are counted to be lLOC.

This type of measurement is highly dependent on programming languages. A code written in Java may be much more effective than C. Two programs that give the same functionalities written in two different languages may have very different LOC values. The advantage of LOC is its ease of calculation, though it neglects all other factors that affect the complexity of software, such as the name of variables, classes, structures, coupling, cohesion, inheritance, and so on.

1.2.2 Some OO Metrics

a. The CK Metrics Suite

As mentioned above it is developed by Chidamber and Kemerer [58] as a suite of class-oriented metrics. There are six class-based metric for OO codes.

Weighted methods per class (WMC): It is the sum of the complexities of all methods of a class.

Depth of the inheritance tree (DIT): It is the maximum length in between the node and the root.

Number of Children (NOC): It is the number of subclasses.

Coupling between object classes (CBO): It is the number of coupled classes.

Response for a class (RFC): It is the number of methods that can be triggered by a message sent to an object.

Lack of cohesion in methods (LCOM): It is the number of methods that use one or more of the same attributes [58].

These metrics are used for comparison purpose with the proposed measure.

b. Weighted Class Complexity

Misra and Akman proposed two metrics [51][48] for inheritance and class features of the OO code. Both metrics are based on cognitive weights. For including the inheritance property of the OO code, the authors first suggested calculating the weight of individual method in a class by associating a number (weight) with each member function (method), and then we simply add all the weights of all the methods. This gives the complexity (weight) of a single class/object. There are two cases for calculating the whole complexity of the entire system (if the system consists of more than one class or object), depending on the architecture:

- If the classes/objects are in the same level then their weights are added.
- If they are subclasses or children of their parent then their weights are multiplied.

If there are m levels of depth in the object oriented code and level j has n classes then the cognitive code complexity (CCC) of the system is given by

$$CCC = \prod_{j=1}^{m}\left[\sum_{k=1}^{n} CC_{jk}\right] \qquad (1)$$

The second metric proposed by Misra and Akman is based on the theme that complexity of a single class depends on attributes and as well as on the complexity of the methods. Accordingly, the authors suggested Weighted Class Complexity (WCC) as

$$WCC = N_a + \sum_{p=1}^{s} MC_p \quad (2)$$

Where Na is the total number of attributes and MCp is the complexity of pth method of the class.

If there are *y* classes in an object oriented code, then the total complexity of the code is given by the sum of weights of individual classes.

$$TotalWeightedClassComplexity = \sum_{x=1}^{y} WCC_x \quad (3)$$

Both of these metrics are used in a modified and enhanced form in the proposal.

1.3 Purpose and Scope of Research

Most of the existing metrics may yield conflicting results [47]. Some researchers emphasise that Halstead metrics are based on assumptions and those assumptions may mislead developers. For instance, to measure 'Time to Implement' in Halstead, 'Effort to Implement' should be divided by 18. Here, 18 is an example for some of the assumptions. Even though its psychological validation had been proved, it may be different with modern programming languages, for Halstead is out-dated. CC is usually used for testing rather than measuring the code complexity [41]. Furthermore, some researchers say that McCabe's Cyclomatic Complexity is based on poor theoretical foundations and an inconvenient model of software development [42].

The existing metrics consider only a fact or a few facts that affect the software complexity. Most of them do not consider the characteristics of multi-paradigm languages. They are for either procedural or OO languages.

This thesis study is based on measuring complexity per paradigms. The research begins with investigating the factors of procedural paradigm codes, because procedural

paradigm is one of the oldest ways of computer programming. By combining those factors, a metric is proposed to measure the cognitive complexity of a procedural code. Thereafter, with a more modern approach, the research is extended further by adding OO features so that the newer metric could be used not only for procedural, but also either for OO codes or in more general meaning for multi-paradigm codes. Under scope of this research it is believed that multi-paradigm is highly popular in modern days because developers need the simplicity of procedural languages and maturity of OO languages. Due to the fact that multi-paradigm concept covers the features of two or more paradigms; it has become a necessity among software developers. To meet the developers' needs, an extended metric was proposed which can be used in most of the popular programming languages such as Python, Java, C, and also in web applications such as JavaScript, JSP, PHP, etc.

The widely used old metrics mentioned above give numerical values as outputs. Those numbers may not always connote the quality of software. By using the available metrics, it is highly possible to reach vague and conflicting results. For example, ordinary metrics give a value such that one code is more complex than another one. It is not clear enough the notion of 'being more complex'. In other words, being more complex may imply that the first code is a great software system developed for banking systems, and the other one is a simple calculator. Another possibility is that both of the software products have the same functionalities, but the second code was written in a more efficient way. That may be the reason for the second one being less complex. Due to not measuring functional aspect, this type of metrics makes developers fall into such ambiguous results. Hence, an exact comment cannot be made by looking at the results.

According to Kearney et al. [36] the large majority of software complexity metrics have been developed with little consideration of the understanding of programmers. Software metrics should be developed with regard for the understanding process.

On the other hand, this thesis deals with cognitive complexity aspect. Some metrics are proposed in this research to measure the cognitive complexity of a program. For example, Multi-paradigm Complexity Measurement method is combined with Function Point as an attempt to assess code quality. So, through measuring cognitive complexity

of a code, which is designed as the internal aspect in this study, and functional complexity of the software, also known as the external aspect, code quality value is reached.

Furthermore, with the proposed metric in this study due to combining cognitive code complexity with functional complexity of the program, the quality of the code can better be assessed cognitively and functionally. If a code also contains redundancy it can be evaluated with the proposed metric.

In Chapter 2, software quality and complexity metrics are discussed. In Chapter 3, a description is given about multi-paradigm languages and also the need for a new metric is explained. The procedural paradigm based metric is proposed in Chapter 4. The fourth chapter also covers complexity factors of procedural languages. In Chapter 5, the previous metric is extended by adding OO factors of complexity in order to propose a metric for multi-paradigm codes. Chapter 6 consists of conclusions and suggestions for future work.

GCRIS

CHAPTER 2

SOFTWARE QUALITY AND COMPLEXITY METRICS

2.1 Introduction

Software quality is closely related with testing and measurement. Fenton [22] defines measurement as follows; “Measurement is the process by which numbers or symbols are assigned to attributes of entities in the real world in such a way as to describe them according to clearly defined unambiguous rules.” Testing techniques tend to find defects, bottlenecks and weaknesses of a software system. Measurement aims to find the complexity in order to understand the effectiveness of the software’s code.

Requirement to improve the software quality is the prime objective, which promotes research projects on software metrics technology. It is always hard to control the quality if the code is complex [3][23]. Complex codes are hard to review, test, maintain and manage. As a consequence, those handicaps increase the maintenance cost and the cost of the product. Due to these reasons, it is strongly recommended that the complexity of the code should be controlled from the beginning of the software development process. Software metrics help to achieve this goal.

In order to increase the quality, the complexity should be decreased [3][23], because complexity increases risks of having defects, difficulty of maintenance and integration. Since this research is focused on cognitive complexity, it should be mentioned that complexity decreases the comprehensibility. To decrease the complexity of software, the factors that affect the complexity should be considered. Some of the factors that affect the procedural complexity are variables and structures. Some of the factors that affect the OO complexity are attributes, structures, and classes. Thus, in order to conceive the complexity of multi-paradigm code, the complexity factors of both of the paradigms should be considered, since multi-paradigm includes the features of both procedural and OO paradigms.

2.2 Classification of Languages

Programming languages are based on programming paradigms. Although there are vast amount of programming paradigms the scope of this research encompasses procedural paradigm, OO paradigm, and multi-paradigm. Multi-paradigm means a paradigm which carries features of two or more programming paradigms. Under the scope of the research the programming languages are classified as;

Procedural programming: it is based on structures and structural flow of algorithms. Programming language C is a good example to procedural languages.

In the past, researchers proposed their methodologies for evaluating codes, which were written in procedural languages [50], such as C. Later, studies focused on OO programming languages, e.g. Java [43, 44, and 45].

OO programming: it provides data abstractions of hierarchical classes for programmers. Java is a popular [73] example for programming languages in this category.

Some of the benefits of OO are faster development, higher quality, easier maintenance, reduced costs, increased scalability, improved structures, and higher level of adaptability [19].

Multi-paradigm programming: it encompasses both procedural and OO paradigms. A programmer may choose to use either procedural code or OO, or even use both of them in the same code. For example: Python. Even though Python is not as popular as Java or C++ it is used by some software giants such as Google and YouTube [56]. Additionally, several conferences and workshops devoted to Python including SciPy India 2009 [61], RuPy '09 [59], Poland, FRUncon 09 [24], USA, ConFoo.ca 2010 [14], Canada, are proving the importance of Python. These facts show the need for modern multi-paradigm programming languages. C++ is known as the most used programming language in the world [73] which is another multi-paradigm language.

All stages in the development life-cycle need to be evaluated from the quality point of view. It is usually expected that the most important one amongst these stages is the

quality of the code which is highly affected by the programming paradigm used for development.

2.3 Quality of Software

There are several quality attributes such as security, performance, reusability, availability, testability, correctness, maintainability, reliability, integrity and many others [4][25]. To achieve some of those quality attributes, complexity should be reduced. For example, to be able to test software easily it is necessary that the software is not complex. Otherwise, the testing process will be harder and thus the cost will be higher.

What makes software quality assurance unique is product complexity, visibility, and development process. Actually, complexity of software products has been observed for decades. Complexity of software product is much higher than that of other industrial products. Visibility is another difficulty of software quality assurance, since other industrial products are visible but software products are not visible until the end is reached. Software development process differs with its development methodologies and difficulties in finding and removing defects [25]. Similarly, Hughes and Cotterell [29] state that intangibility, increasing criticality of software, and accumulating defects during development process make the software quality unique. Furthermore, software needs to be measured in order to understand its quality. Otherwise, it may not be possible to make an effective project management.

One of the most important and effective tools in assessing software quality is to use complexity metrics explained in the following section.

2.4 Complexity Metrics

Complexity is defined as [30] “the degree to which a system or component has a design or implementation that is difficult to understand and verify”.

Metrics that concern the complexity of software can be classified as procedural metrics, OO metrics, and multi-paradigm metrics. In addition to those, there are also other metrics which are widely used, such as LOC, Halstead, and CC. Although, it is clear that

there are more metric types, only the above mentioned metrics are under the scope of this research, because it is not practically possible and significant to analyse all the existing metrics in a research.

It is well known that the maintainability is one of the important factors that affect the quality of any kind of software. JavaScript also requires modelling, measurement, and quantification for the ease of maintainability purpose. In addition, software metrics play an important role since they provide useful feedback to the designers to impact the decisions that are made during design, coding, architecture, or specification phases. Without such feedback, many decisions are made in ad hoc manner.

Number of researchers has proposed variety of complexity metrics [16][12] for different types of software, software languages [50], software products and related technologies [5][6]. All the reported complexity measures are supposed to cover the correctness, effectiveness and clarity of a system and to provide good estimate of these parameters. With the emergence of the new technologies, also new measurement techniques evolve. There is an ongoing effort to find such a comprehensive measure, which addresses most of the parameters for evaluating quality of the system. In addition, the quality objectives may be listed as performance, reliability, availability and maintainability [4][25] that are all closely related with software complexity.

GCRIS

CHAPTER 3

MULTI-PARADIGM LANGUAGES

3.1 Introduction

Multi-paradigm programming languages are the languages which carry the features of two or more paradigms. The multi-paradigm concept is taken as the combination of procedural and OO paradigms in this thesis. Tim Budd's [11] definition for multi-paradigm is that it is a framework in which various constructs are obtained from different paradigms. In other words, it is a software development style that supports a number of different language paradigms which provides different problem solving styles. Therefore, one of the greatest advantages of using a multi-paradigm language is that it provides programmers a wider aspect of programming styles. That is to say, a programmer may prefer using developing code with very few classes or even without any class. Hence, multi-paradigm decreases the constraints for developers.

In the past, procedural languages gained popularity for developing programs. Those languages help developers to reduce a problem in its composite parts. Later, OO languages took the lead in popularity due to providing features of class hierarchies with data and methods encapsulated in classes. A need arose for a new paradigm considering the disadvantages of the former ones. Hence, multi-paradigm languages gained popularity by merging elements of various programming paradigms into a cohesive language which utilises programming and conceptual aspects from different paradigms [38].

According to Coplien [15], "Multi-paradigm design becomes an audit for that intuition and provides techniques and vocabulary to regularise the design." Multi-paradigm programming makes developers think about the nature of complexity [46]. Therefore, it is one of the effective ways of coping with complexity.

Ierusalimschy [31] uses Lua scripting language which is similar to Scheme, in order to benefit from the effectiveness of multi-paradigm design. Instead, this study used JavaScript as a multi-paradigm scripting language. Though, the study does not contain validation of the proposed metrics only with JavaScript codes, but also C as a procedural language, Java as an OO language, and C++ and Python as multi-paradigm languages.

3.2 Multi-Paradigm Languages

Some of the most popular multi-paradigm languages are C++, Python, JavaScript, Perl, Ruby and PHP [73]. For validation, JavaScript, C++ and Python are used as multi-paradigm languages in this research. One of the advantages of Python is being platform independent since it can be used in Windows, Linux, BSD, Macintosh and even in cell phones. Another important advantage is its readability. According to Python official web site, Python additionally provides easy integration and lower maintenance cost [56]. One of the strongest advantages of C++ is its performance. However it is not platform-independent. JavaScript is a scripting language used to embed on HTML files. One of the valuable advantages of JavaScript is that it provides an interaction between the web page and client without using any extra networking resources.

Table 1 shows Scriptol's [62] descriptions of popular multi-paradigm languages.

Table 1: Multi-Paradigm Languages

Multi-paradigm Language	Description
C++	It is a combination of C and objects. It provides an extended library and templates. System programming is possible in C++ as C, but C++ allows larger projects and applications.
Perl	A scripting interpreted language. Readability and ease of use are not the goals. It is usually used by network administrators and for small CGI scripts.
PHP	Designed to be embedded inside HTML to build dynamic web pages or update them from databases. It is possible to produce HTML pages by using PHP.

Python	A modern interpreted language with powerful built-in features and a unique indentation feature to shorten coding. It provides developers programming very fast. It is powerful and easy to learn.
Ruby	Designed with simplicity in mind. It is interpreted, and has a proprietary but extensible library. Writing scripts are easy.
JavaScript	JavaScript has been invented to build dynamic client-side HTML pages. It is used for interactivity in web pages.

3.3 The Need for a New Metric

There are too many popular and simple metrics that do not include the most important complexity factors [34]. Popular metrics that are used inside tools are simplistic [72]. As already noted before, various old metrics are under several criticisms. These criticisms are mainly based on lacking a theoretical basis [35][77], lacking in desirable measurement properties [82], being insufficiently generalized or too dependent on implementation technology [79], being too labour-intensive to collect [37] and only confined to the features of procedural languages.

Most of the available metrics cover only certain features of a language. For example, if Lines of Code (LOC) is applied, then only size will be considered; if McCabe's Complexity metric is applied, the control flow of the program will be covered. In addition, the metrics applicable to the procedural languages do not fit to the modern languages such as Ruby or Python [12]. Metrics that are developed specifically for OO languages still do not satisfy the requirements for multi-paradigm since multi-paradigm does not cover merely OO features.

Moreover, most of the available metrics do not consider the cognitive characteristics in calculating the complexity of a code, which directly affects the cognitive complexity. Complexity of a code directly affects comprehension. The understanding of a code is known as program comprehension and is a cognitive process and related to cognitive complexity. The cognitive complexity is defined as the mental burden on the user who deals with the code, for instance, the developers, the testers and the maintenance staff. In

the proposal, cognitive complexity is calculated in terms of cognitive weights [80]. Cognitive weights are defined as the extent of difficulty or the relative time and effort required for comprehending the given software, and measure the complexity of the logical structure of the software. A higher weight indicates a higher level of effort required to understand the software. A high cognitive complexity is undesirable for several reasons, such as increased fault-proneness and reduced maintainability. Moreover, one of the programmers may leave the project and another one may come to sustain the project. In such a case, the code should have a low complexity so that the latter programmer can easily grasp the code without wasting too much time. Additionally, cognitive complexity also provides valuable information for the design of systems. High cognitive complexity indicates poor design, which sometimes can be unmanageable [9]. In such cases, maintenance effort increases drastically.

In this research, the factors that affect the cognitive complexity of a procedural language are investigated. Next, the metric is extended by adding OO factors so that the research can be used for multi-paradigm languages.

GCRIS

CHAPTER 4

PROPOSED METRICS AND THEIR IMPLEMENTATIONS

4.1 Introduction

Multi-paradigm programming is widely used, as mentioned by TIOBE [73] and LangPop [40] that C++, Python, Ruby, JavaScript and some other multi-paradigm programming and scripting languages are highly popular. Initially, in this section, procedural part of multi-paradigm is studied. The study investigates the factors that affect the complexity of a procedural code and then proposed a metric for procedural languages. For validation of the metric, the metric is applied on some sample codes which are written in JavaScript scripting language. Some of the reasons for choosing JavaScript are:

- It is a popularly used scripting language.
- There are not many researches which use JavaScript.
- JavaScript has lots of skills such as providing a programming tool for HTML, making an HTML code dynamic, give response to events, validate data, and get client side information [83].
- Even though having OO features, it is widely used for especially writing shorter codes [2].
- According to TIOBE Programming Community Index for January 2010, JavaScript is the ninth most popular language among all types of programming/scripting languages [73].

JavaScript is a simple client-side web programming language [60][17]. Despite JavaScript is used for validation of the metric, the proposed metric can also be applied for other procedural languages, for the metric covers most of the factors that affect the

complexity of procedural languages generally. Detailed explanations of the metric and empirical validations are sequentially given in 4.1.2 and 4.1.3.

4.2 The Proposed Metric

Definitions of complexity [30] imply that all the factors which make code difficult to understand are responsible for complexity. Accordingly, the factors which are responsible for the complexity of a procedural code should be identified. When procedural codes are analysed it is found that the following factors are responsible for the cognitive complexity:

1. Number of Arbitrarily Named Variables (ANV), [39]
2. Number of Meaningfully Named Variables (MNV), [39]
3. Number of operators; [49] and
4. Cognitive weights of basic control structures (BCS) [80].

Number of Arbitrarily Named Variables (ANV): The names of variables used in the code play a very important role in increasing or decreasing the understanding of the code. Although, it is suggested that the name of the variables should be chosen in such a way which is meaningful in programming, most of the developers do not follow it very strictly. If the variable names are taken arbitrarily, there is no problem if the developer himself is evaluating the code. However, it is not the case in real life implementations. After the system is developed, especially during maintenance time, arbitrarily named variables increase the difficulty of understanding four times more [39] than the meaningful names. In the formulation of the proposed metrics, the weights of the arbitrarily named variables are considered to be four times greater than the meaningfully named variables.

Number of Meaningfully Named Variables (MNV): From the discussion part taken in the above section, it is clear that meaningful named variables are more understandable than arbitrary named variables. The weight of meaningfully named variables is assigned as one unit.

Constants: Constants are out of the scope of the research as proposed by Kushwaha and Misra [39]. It is possible to assume that constants have similar comprehensibility with MNV, because it supposed that constants make a similar effect on human understanding. For this reason, it is supposed that constants should be counted similar as MNV.

Words or sentences written in double quotations or single quotations are not assumed to be constants, because in the case of treating them as MNVs, ambiguity occurs. Because any character, word or sentence may be a string. Moreover, a string may be divided into many strings or characters. This ambiguity was realised during empirical validations. For this reason, they are exempt of being treated either as ANV or MNV. This fact is the same for characters. Only if the string or the character is a variable will it be enquired whether it is ANV or MNV.

It should be noted that discriminating the MNVs and the ANVs is subject to developers' choice. A standard should be defined among a software team and the style of MNVs and ANVs should be defined by their cognitive choice.

Number of operators: Software in cognitive informatics is perceived as formally described design information and implemented instructions of a computing application [80]. In other words, complexity of any software is in the form of difficulty in understanding the information contained within. By keeping this point in mind, in formulation of Procedural Cognitive Complexity Measure (PCCM), the contribution of information contents is considered in terms of occurrences of operators.

Cognitive weights of Basic Control Structures (BCS): The complexity of a program is directly proportional to the cognitive weights of Basic Control Structures (BSC). The cognitive weight of software is the extent of difficulty or the relative time and effort for comprehending the given software modelled by a number of BCS. BCS are basic building blocks of any software and their weights are one, two and three respectively. These weights are assigned on the classification of cognitive phenomenon as discussed by Wang [80]. He proved and assigned the weights for sub-conscious function (sequence), meta-cognitive function (selection) and higher cognitive function (looping) as 1, 2 and 3 respectively. Although the thesis followed a similar approach as Wang [80], there are some modifications in the weights of some BCS (Table 5). For example,

try-catch is included inside BCS (a special feature of JavaScript codes) in the list and its weight is assigned as 1, based on its structure. As a result the identified BCS and their corresponding weights are given in Table 5.

From the table it is clear that sequence, condition and loops in JavaScript have similar structures with other programming languages. The differences lie in functional activity and exceptions, where alert/prompt/throw, event, and try-catch are new basic control structures. However, try-catch, is common for most of the modern programming languages. The new basic control structures are represented by the corresponding flow graph notations in Table 5.

The weight of the structures depends on their flow diagram. The number of categories and structures can be either reduced or increased accordingly, based on the same logic. For example, alert/prompt/throw is a specific feature of JavaScript which can be removed while measuring codes that are written in other languages.

For example, try-catch has two variations; either try or catch (or one of the couple of catches) will be executed. Therefore, its Cognitive Weight Unit (CWU) is assigned as 2. In this logic, the number of 'catch's should be counted. 'alert', 'prompt', 'throw' and 'event' are kinds of function calls. Since Wang gives value 2 for function calls, in this study, the same weight is assigned to them. These types of functions differ from other functions by changing the flow of a program. Although the study considers the fact that there are also other such types of functionalities in JavaScript, the specified ones are the most commonly used types. For nested conditions value 1 and for nested loops value 2 is assigned to each sub-condition and sub-loop. The logical reason is shown in the figures below.

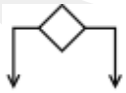


Figure 2: Condition (1)
(CWU=2)

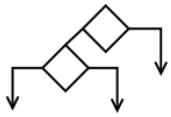


Figure 3: Nested Condition (2)
(CWU=3)

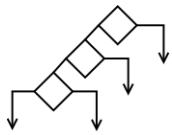


Figure 4: Nested Condition (3)
(CWU=4)



Figure 5: Loop (1)

(CWU=3)

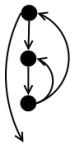


Figure 6: Nested Loop (2)

(CWU=5)

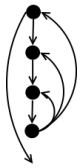


Figure 7: Nested Loop (3)

(CWU=7)

Table 2: Nested Conditions 1

Figure 1	Figure 2	Figure 3

if (condition)
statement;
else
statement;
	if (condition1)
	statement;
	else if (condition2)
	statement;
	else
	statement;
		if (condition1)
		statement;
		else if (condition2)
		statement;
		else if (condition3)
		statement;
		else
		statement;

Table 3: Nested Conditions 2

Figure 1	Figure 2	Figure 3

if (condition)
statement;
	if (condition1)
	if (condition2)
	statement;
		if (condition1)
		if (condition2)
		if (condition3)
		statement;

Table 4: Nested Loops

Figure 4	Figure 5	Figure 6

for (content)
statement;
	for (content1)
	for (content2)
	statement;
		for (content1)
		for (content2)
		for (content3)
		statement;

In Figure 2, CWU is 2 as demonstrated by Wang [80]. Figure 3 shows that there are three possibilities of flow. So, CWU is 3. In Figure 4, 3-leveled of conditional hierarchy is given. There are 3 variations in the flow. For this reason CWU value is assigned to 4.

Figure 5, 6, and 7 are the demonstration of loops. According to Wang, loop's CWU should be 3 based on its flow diagram. Based on the same logic, it is proposed that each nested loop increases the complexity by 2 CWU. For example, in Figure 5, in a nested loop there are 5 variations, and thus CWU is 5. Figure 7, shows that three nested loops make 7 variations. Hence its value should be 7 CWU. The pseudo codes of the conditions and loops (Figure 2-7) are given in Table 2, 3, and 4. Conditions and nested conditions are written in two ways which are shown in Table 2 and 3. Table 4 shows the codes for loops and nested loops.



Figure 8: try-catch 1



Figure 9: try-catch 2



Figure 10: try-catch 3

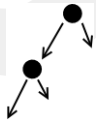
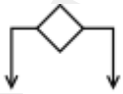


Figure 11: try-catch 4

While calculating try-catch statement, only 'catch's are counted. Because try-catch directs a program into possibilities similar to conditional structures, its weight is assigned to be 1. In a code, there may be more than one catches. In that case each catch will add 1 to the CWU value, because each catch increases the number of variations by 1. For example; if there are 1 try and 2 catches, then the first catch is counted as 1, and the second catch as 1. Therefore the total weight is 2. Of course the structures inside try-catch should also be considered. If there is no catch, then there cannot be a 'try'. The same goes for try; if there is no try, then there cannot be catch. However, for one try there can be many 'catch's. For example for one try, there may be 5 'catch's. In other words, the first catch is the initiator of try-catch variations. Each upcoming catch increases the number of variations only by one. In try-catch statements try does not have a weight. This has two reasons. First, try does not contain any variable or operator. Second, try is the expected flow of a program rather than being a condition. The exceptional cases are held by catches. Therefore, it seems more logical to count catches and eliminate try. Figure 8 is an example for a try with one catch. The weight will be as

a variable multiplied by 1 for catch. The weight of try will be as 0. Figure 9 is an example for two catch's. The weight will be as each error-variable multiplied by each catch's weight which is 1. That totally makes 2. Of course, if inside catch instead of MNV, ANV is used then it would be as 4x1. If the catch does not contain any variable, then it would be as 0x1. Figure 10 is an example of three catch's. Similar approach should be applied here, too. Figure 11 shows a try-catch where there is another try-catch inside. In this case still the weight totally makes 2. In short, each catch has a value of 1 which should be multiplied by the variable used for catching errors.

Table 5: Basic Control Structures

Category	BCS	CWU	Flow Graph
Sequence	sequence	1	
Condition	if-else	2	
	switch	2	
	sub-if (in nested conditions)	1	[Figure 2, 3]
Loop	for	3	
	for...in	3	
	while/do...while	3	
	sub-loop (in nested loops)	2	[Figure 5, 6]
Functional Activity	function-call	2	
	alert/prompt/ throw	2	
	event	2	
	recursion	3	
Exception	try...catch	1	

Accordingly the total complexity of a JavaScript is given by the following formula:

Procedural Cognitive Complexity Measure

$$PCCM = \sum_{i=1}^{n} \sum_{j=1}^{m_i} (((4 * ANV + MNV) + operators) * CWU_{ij}) \qquad (1)$$

Here, the complexity measure of a procedural code (PCCM) is defined as the sum of complexity of its n modules (if exists) and module *I* consists of m_i lines of code. In the context of formula 1, the concept of cognitive weights is used as an integer multiplier. Therefore, the unit of the PCCM is: CWU which is always a positive integer number. This implies achievement of scale compatibility. This logic was derived from Unified Complexity Measure [49]. Cognitive differences of variables were added inside the metric.

By looking at the formula and the methodology of reaching to it, it can be realised that the proposed metric can be thought as a dynamic metric since its structures can be changed due to the needs of a programming language or even of a scripting language. For instance, the BSC is edited according to the features of JavaScript. It could be modified further to be used with another language.

4.3 Demonstration of the Metric

For demonstration of PCCM, 3 different types of codes written in JavaScript taken from the web are considered. These programs are different from each other in their architecture. The calculations of PCCM for these examples are given in Tables 6-8. The structures of all the three programs in tables are as follows: The second column of the tables shows the JavaScript codes. The sum of Arbitrarily Named Variables (ANV), the Meaningfully Named Variables (MNV) and the operators in the line is given in the third column of the table. The cognitive weights of each JavaScript lines are presented in the forth column. The JavaScript complexity calculation measure for each line is shown in the last column of Table 6-8.

Table 6: Example 1 [84]

Line no.	JavaScript Code	ANV+MNV+ operator+constant	CWU	PCCM
1	var i=0;	6	1	6
2	for (i=0;i<=5;i++)	17	3	51
3	{	0	0	0
4	document.write("The number is " + i);	5	1	5
5	document.write(" ");	0	1	0
6	}	0	0	0
	Total	-	-	62

Description of Example 1:

Line 1: 'i' is an ANV, '0' is a constant, '=' is an operator. For this reason 4x1+1+1=6

Line 2: '=', '<=', '++' are operators. 'i' is used three times. There are two constants '0' and '5'. As a result, 4x3+3+2=17

Line 3: '{' is neither a variable, nor an operator, nor a structure.

Line 4: 'i' as an ANV. Therefore, 4x1, '+' 1 => 4x1+1=5

Line 5: There is not either a variable or an operator.

Line 6: There is no any structure, variable or operator.

Table 7: Example 2

Line no.	JavaScript Code	ANV+MNV+ operator+constant	CWU	PCCM
1	var i, j;	8	1	8
2	for (i=0;i<=5;i++){	17	3	51
3	for (j=0; j<=i; j++)	20	5	100
4	document.write(“*”);	0	1	0
5	document.write(“ ”);	0	1	0
6	}	0	0	0
	Total	-	-	159

The important thing is to calculate the most realistic value which really represents the complexity of the script. If the complexity values of all related complexity measures given in Table 7 are compared, it will be found that PCCM values are higher than the lines of code, cyclomatic complexity [43], difficulty and time [28]. Its reason is that, PCCM represents the complexity values due to all parameters responsible for complexity; however, all these parameters are independently evaluated by different metrics.

Line 1: ‘i’ and ‘j’ are ANVs. 4+4=8

Line 2: ‘i’ is used three times. After adding also the operators and constants, the total makes 4x3+3+2=17

Line 3: There are 4 MNVs and a constant. 4x4+3+1=20. Due to being a nested loop 20x5=100

Line 4: However, for being a statement its structure value is 1, there is not any kind of ANV, MNV or operator. Therefore, 0x1=0

Line 5: It is similar to Line 4.

Line 6: There is no any structure, variable or operator.

Table 8: Example 3 [85]

Line no.	JavaScript Code	ANV+MNV+ operator+constant	CWU	PCCM
1	var txt=””;	2	1	2
2	function message(){	0	1	0
3	try{	0	0	0
4	adddlert(“Welcome guest!”);}	0	1	0
5	catch(err){	1	1	1
6	txt=”There was an error on this page.\n\n”;	2	1	2
7	txt+=”Error description: “ + err.description + “\n\n”;	5	1	5
8	txt+=”Click OK to continue.\n\n”;	2	1	2
9	alert(txt);}	1	2	2
10	}	0	0	0
	Total	-	-	14

Line 1: It is obvious that ‘txt’ means text. It is counted as 1x1, for it is MNV. ‘=’ is an operator. So, totally that line’s weight is 2.

Line 2: It does not have any variable or operator. The value of statement is 1. 0x1=0

Line 3: It does not have any variable or operator.

Line 4: It is similar to Line 2. Its structure’s weight is 1 due to being a statement.

Line 5: ‘err’ can be counted as an MNV. The weight of the structure is 1. Therefore, 1x1=1

Line 6: ‘txt’ and ‘=’ totally makes 2.

Line 7: ‘txt’, ‘+=’, and 2 times ‘+’ totally make 4. For ‘err’ is accepted as an MNV, the total weight of the line is 5.

Line 8: It is similar to Line 6.

Line 9: ‘alert’ is a kind of function call. So, the weight of the structure is 2.

Line 10: It is obviously 0.

Table 9: Example 4

Line no.	JavaScript Code	ANV+MNV+ operator+constant	CWU	PCCM
1	var sum=0, min=100, max=0;	9	1	9
2	var grade, arrayNumber, average, studNo;	4	1	4
3	var studNo=prompt(“Number of Students:”,””);	2	2	4
4	var grade=new Array();	2	1	2
5	for (arrayNumber=0; arrayNumber<studNo; arrayNumber++){	8	3	24
6	grade[arrayNumber]=prompt(“Grade:”,””);	3	2	6
7	sum=sum+parseInt(grade[arrayNumber]);	6	1	6
8	if (grade[arrayNumber]>max)	4	2	8
9	max=grade[arrayNumber];	4	1	4
10	if (grade[arrayNumber]<min)	4	2	8
11	min=grade[arrayNumber]; }	4	1	4
12	try{	0	0	0
13	if (studNo==0)	3	2	6
14	throw “DivZero”;	0	2	0
15	else if (studNo<0)	3	2	6

16	throw “Minus”;	0	2	0
17	average=sum/studNo;	5	1	5
18	document.write(“Maximum grade is “+max+” ”);	3	1	3
19	document.write(“Average is “+average+” ”);	3	1	3
20	document.write(“Minimum grade is “+min+” ”);	3	1	3
21	catch(er){	1	1	1
22	if (er==”DivZero”)	2	1	2
23	alert(“There should be some students”);	0	2	0
24	else if (er==”Minus”)	2	2	4
25	alert(“Student number cannot be negative”); }	0	2	0
	Total	-	-	112

Line 1: There are 3 MNVs, 3 operators and 3 constants.

Line 2: There are 4 MNVs.

Line 3: There is a prompt. Thus, MNV+operator, which makes 2 is multiplied by prompt’s weight 2.

Line 4: 1 MNV and 1 operator make up 2.

Line 5: Operators, constants and variables totally make 8. Due to being ‘for’ loop, the value is multiplied by 3.

Line 6: Both ‘grade’ and ‘arrayNumber’ are MNVs. There is also an operator. Thus, 3 is multiplied by prompt’s weight.

Line 7: Similar to Line 6, but this time it is a simple sequence.

Line 8: For being ‘if condition’ the total weight of MNVs and operators are multiplied by 2.

Line 9: It is a simple sequence.

Line 10: Similar to Line 8.

Line 11: Similar to Line 9.

Line 12: The start of try-catch.

Line 13: Similar to Line 8.

Line 14: ‘throw’ has a cognitive weight of 2, but there is no any operator or a variable. Thus, it is 0x2.

Line 15: Similar to Line 13.

Line 16: Similar to Line 14.

Line 17: Similar to Line 9.

Line 18, 19, and 20: ‘+’ operators and an MNV make up 3.

Line 21: ‘er’ (error) is counted as MNV. ‘catch’s value is 1.

Line 22: Similar to Line 8.

Line 23: ‘alert’ has a weight of 2, but there is no any constant, operator or variable.

Line 24: Similar to Line 8.

Line 25: Similar to Line 23.

GCRIS

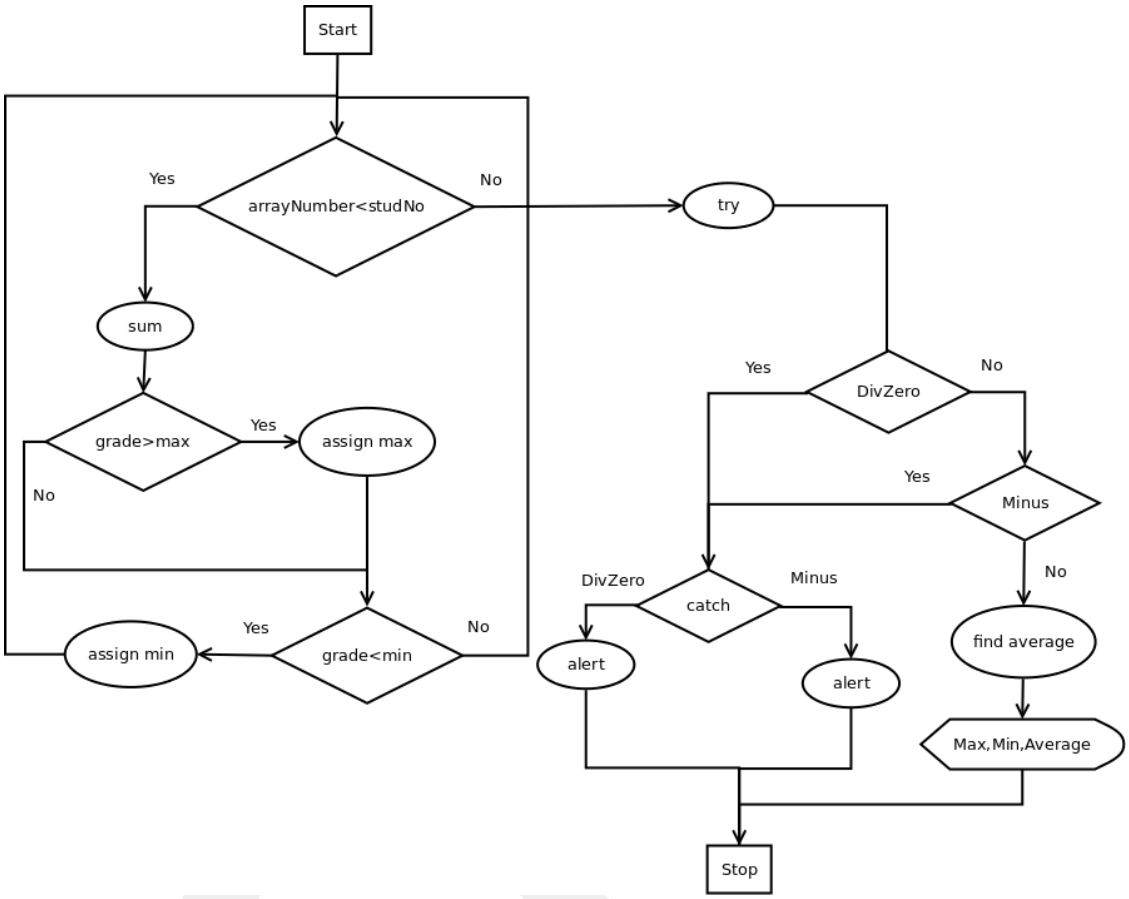


Figure 12: Flow Graph of Example 4

The flow graph of example 4 is given in Figure 12.

Table 10: Comparison of Metrics

Example number	PCCM	eLOC	CC	Halstead			
				V	D	E	T
1	62	4	3	33	4	132	7
2	159	5	5	33	3	99	5
3	14	9	4	13	3	39	2
4	112	25	8	263	12	3156	175

Table 10 shows different values depending on the metric. Example 2 was obviously more complex than example 1. This is because, although both of their LOC value is 6, example 2 has a nested loop, whereas example 1 has only one loop. Example 2 has two

arbitrarily named variables, but example 1 has only one. In human understanding, the second example is clearly more difficult to grasp than the first one. The similar difference is also observed by eLOC and CC. However, none of Halstead results could measure the difference. Yet more, the Halstead data show that example 1 is more complex than example 2. On the other hand, PCCM could realise that the second example was more complex than the first one and measured the difference in a much more sensitive way than eLOC and CC.

Example 3 has a simplistic code which consists of only sequences, except try-catch. Even though CC, and Halstead's V, E, and T consider example 3 more complex than example 2, those metrics were not able to show that example 3 is even simpler than example 1. Whereas, PCCM was the only metric that is closer to human understanding.

Example 4 has a less cognitive complexity than example 2, due to its readability. Almost with a glance, the fourth example's purpose is comprehensible. To understand the second example, more thought process is required, even though the fourth example has more lines of code and a longer flow of process. This fact was recognised only by PCCM among the specified metrics, because only PCCM is capable of measuring also the cognitive aspect of a code.

All the above results have shown that PCCM performs better in reflecting the comparative complexities. This also means PCCM is capable of assessing the quality of the code and hence is a valuable addition to the literature.

4.4 More Examples

The web sites of the examples in this section can be found in Appendix A. The proposed metric is compared with some popular metrics which are developed to be used for most of the programming languages. For not being developed specifically for JavaScript code and not even for generally procedural languages, their deficiencies are obvious in comparison with PCCM. PCCM is a dynamic metric, since its structure can be changed due to the features of any specific language. Though JavaScript carries also OO features, for procedural part of the research only procedural examples of JavaScript are used.

Table 11: Examples

Program	eLOC	CC	PCCM	Halstead			
				V	D	E	T
[1]	2	2	7	13	1	13	0
[2]	14	5	39	148	3	244	24
[3]	8	4	35	148	4	592	32
[4]	3	2	33	79	6	474	26
[5]	6	4	44	114	8	912	50
[6]	7	3	35	93	2	186	10
[7]	9	3	31	167	3	501	27
[8]	15	9	72	237	4	948	52
[9]	12	3	68	212	6	1274	70
[10]	32	3	163	212	12	2544	141
[11]	20	5	40	237	4	948	52
[12]	13	3	62	220	4	880	48
[13]	4	2	27	129	4	516	28
[14]	21	5	120	748	6	4488	249
[15]	4	2	16	148	3	444	24
[16]	10	7	153	152	16	2432	135
[17]	10	4	42	366	12	4392	244
[18]	8	3	34	93	2	186	10
[19]	7	3	10	63	1	63	3
[20]	3	2	9	48	2	96	5
[21]	5	3	10	23	1	13	1
[22]	6	3	24	76	2	152	8
[23]	5	4	30	93	2	186	10
[24]	4	2	19	76	2	152	8
[25]	4	2	19	48	1	48	2
[26]	5	4	28	171	3	513	28
[27]	6	5	54	259	4	1039	57
[28]	13	10	107	514	12	6168	342
[29]	6	4	38	192	4	768	42
[30]	20	31	128	696	4	2784	154

For empirical validation of the PCCM metric, thirty JavaScript codes are analysed. It is believed that the selected 30 scripts are significant in number for comparison since they include different structures and, therefore, contain most of the characteristics of a system required for the validation of the proposed measure. The complexity values of different measures for the cases are summarised in Table 11. All of the analysed scripts were extracted from the web. Table 11 contains the statistics that are collected after analysing those JavaScript codes to evaluate the PCCM measures. Actually, the agenda of empirical validation is two-fold. First, the well known metrics like effective Lines of Code (eLOC), Cyclomatic Complexity (CC) [43], and volume, effort, difficulty and time estimations from Halstead metrics [28] are all applied. Since, those well known metrics have not been tested in JavaScript; the study evaluates the applicability of these metrics to JavaScript codes. Second, the statistics that are collected from those metrics is compared with the values obtained from PCCM to investigate the usefulness and effectiveness of the proposal. More sample scripts are given in Appendix D.

The variables (arbitrary names and meaningful names), basic control structures, i.e. sequence, branch, iteration and function call are directly related with the complexity of a code. If a structure does not contain any variable, operator and constant, its PCCM value becomes zero. All these prove that the proposed measure considers several factors of complexity, and does not directly count the numbers of lines. The graph depicted in Figure 13 shows the comparison results between the effective Lines of Code (eLOC) and PCCM. It is clear from the graph that PCCM values are normally higher than eLOC. It is because that PCCM consists of complexity values due to other parameters/factors responsible for complexity also. In other words, PCCM calculates more factors than eLOC does. However, there is not a conflict or opposition between PCCM and eLOC. In Figure 13, when eLOC value increases so does PCCM value, and vice versa.

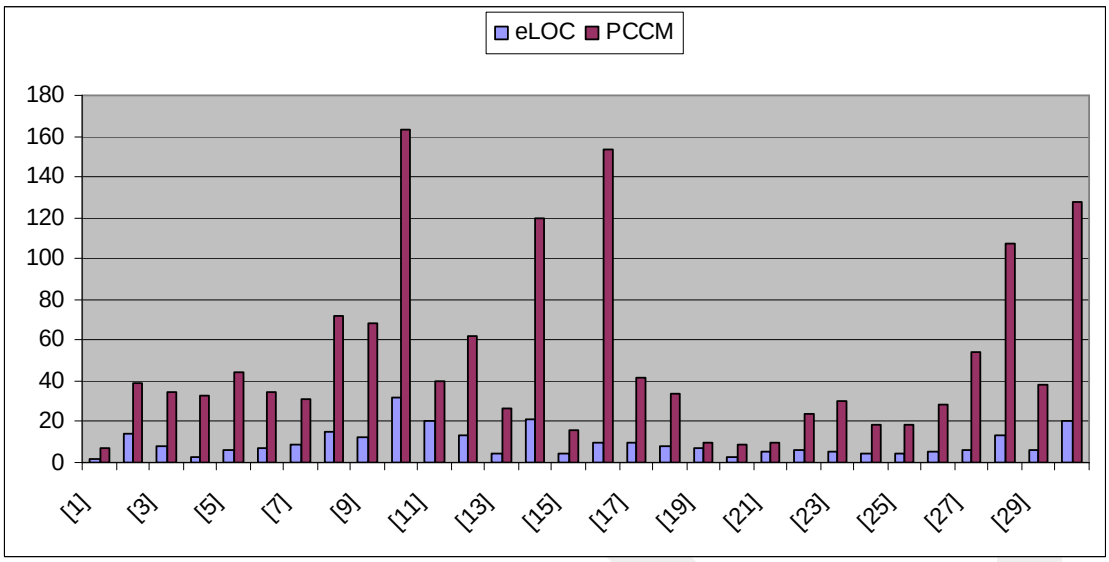


Figure 13: Comparison between eLOC and PCCM

CC attempts to determine the number of execution paths in a program. Therefore, on the contrary to the proposed measure, it does not consider variables, difference between variables, operators or constants. For this reason, for example, CC values for programs 1, 4, 13, 15, 20, 24 and 25, are equal (*CC*=2) (in Figure 14) and are minimum since these programs have no extra modules. However, PCCM does not only consider the factors related with variables and operators but also complexity due to internal structures. The PCCM values for the above mentioned programs are 7, 33, 27, 16, 9, 19, and 19 respectively, which is able to indicate the complexity differences between programs and therefore provide more information.

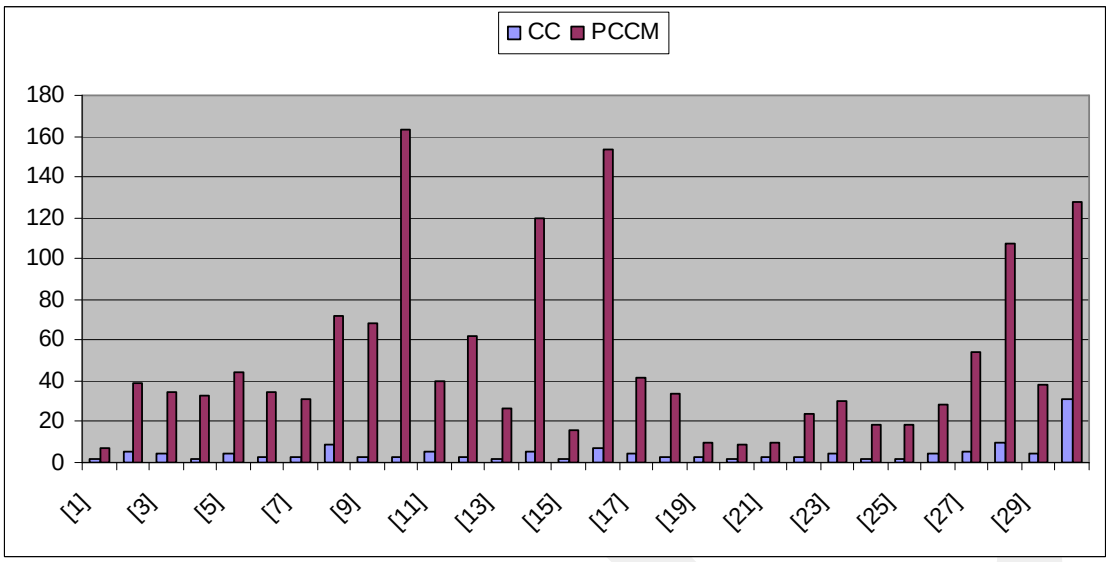


Figure 14: Comparison between CC and PCCM

A graph which covers the comparison between CC, eLOC and PCCM is also plotted in Figure 15, to observe similarities and differences between them. A close inspection of this graph (Figure 15) shows that PCCM has close relations with CC and eLOC. This can easily be seen in the figure, in which PCCM, CC and eLOC reflect similar trends. In other words, higher PCCM values are due to a large number of variables, arbitrarily named variables, a number of iterations, branching structures, function calls or a combination of the previously mentioned features. For example, PCCM has the highest value for script 10 (163), which is due to having the maximum lines of code (32), variables and complex control structures. The similarity that lies between them is the harmony between their increase and decrease. The difference is that some of the programs are showed by eLOC or CC as having almost the same complexity. On the other hand, PCCM is able to catch the differences even if they are hidden in details, and therefore, give more prominent reactions.

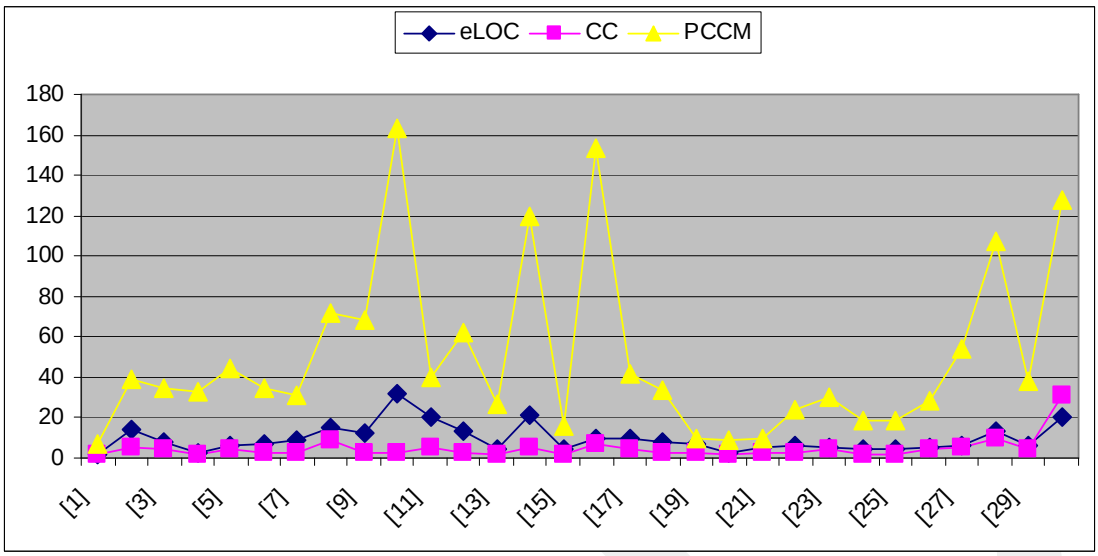


Figure 15: Relative Graph between eLOC, CC and PCCM

PCCM is compared also with the Halstead metrics. The graphs between PCCM, volume, difficulty, effort and time have been demonstrated in figures 16, 17, 18, and 19. It is observed that PCCM has similar trends with volume, difficulty and time. Further, PCCM values are less than the volume measurement of Halstead but are almost similar with the time measurement of Halstead in most of the scripts (except 14, 17 and 28). Actually, the time measurement of Halstead is approximately the time spent to understand a program and PCCM reflects the similar values to this measurement. This proves that the proposed metric is a strong predictor of comprehensibility. Despite the similarity of PCCM and Time (Figure 16), none of Halstead's metrics were capable of measuring the structural complexity of a program [26].

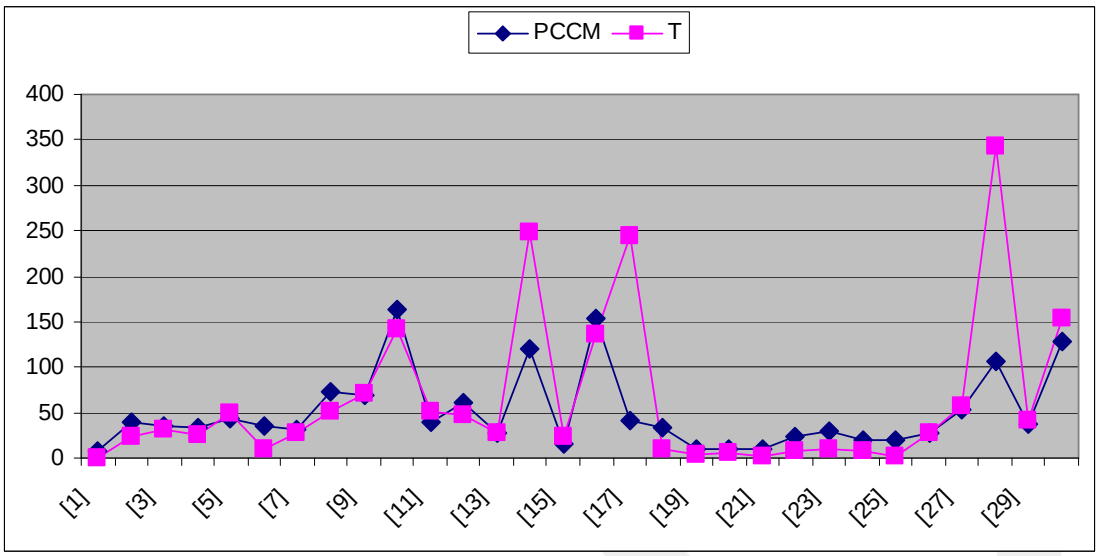


Figure 16: Relative Graph between Time and PCCM

In Figure 17, comparison between eLOC, PCCM and Volume is given. Logically, there should be similarities between eLOC and Volume. However, here PCCM and eLOC values are more similar to each other. Volume has exaggerated values as shown in the graph. The 30 programs are not extremely different in their size and understanding, but according to Volume they are.

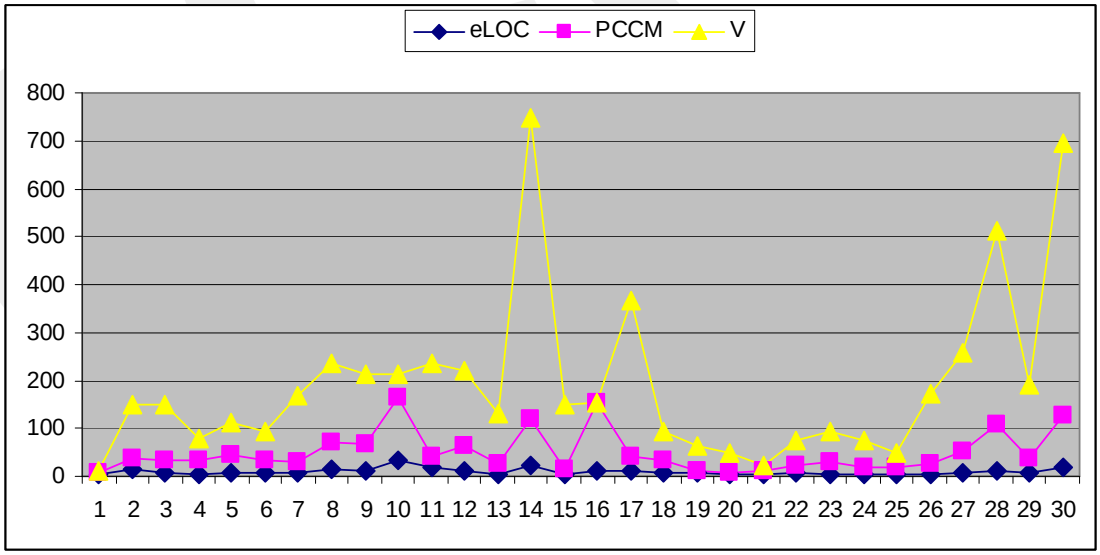


Figure 17: Relative Graph between eLOC, PCCM and Volume

Figure 18 is the demonstration of the comparison of CC, PCCM and Difficulty. Difficulty of a program should have some relation with CC. For most of the examples Difficulty and CC values are more or less similar, but in some cases they may be contradictory as they are in programs 8, 10, 30. As mentioned above PCCM is able to make more sensitive measurements than CC, and this graph shows the similar difference exists between PCCM and Difficulty, too.

Figure 19 shows the comparison between eLOC, CC, PCCM, Effort and Time. In the graph, all the values are remained tiny before Effort. Effort spent to develop a program should have some connections with eLOC, CC, PCCM and Time. Effective LOC varies from 2 to 32, but Effort values change between 13 and 6168. There are some contradictions also. For example, program number 28 has 13 eLOC and program 10 has 32 eLOC. The same programs' values are respectively given by Effort as 6168 and 2544. This does not seem to be an effective measurement to understand how much difficult the program is to be understood by a human. For the given examples the Effort values are too exaggerated and have some contradictions with eLOC, CC and even with Time. It was expected that, at least, it could have a relationship with CC, because if a code is extremely complex, then most probably it consists of a vast number of control paths [41].

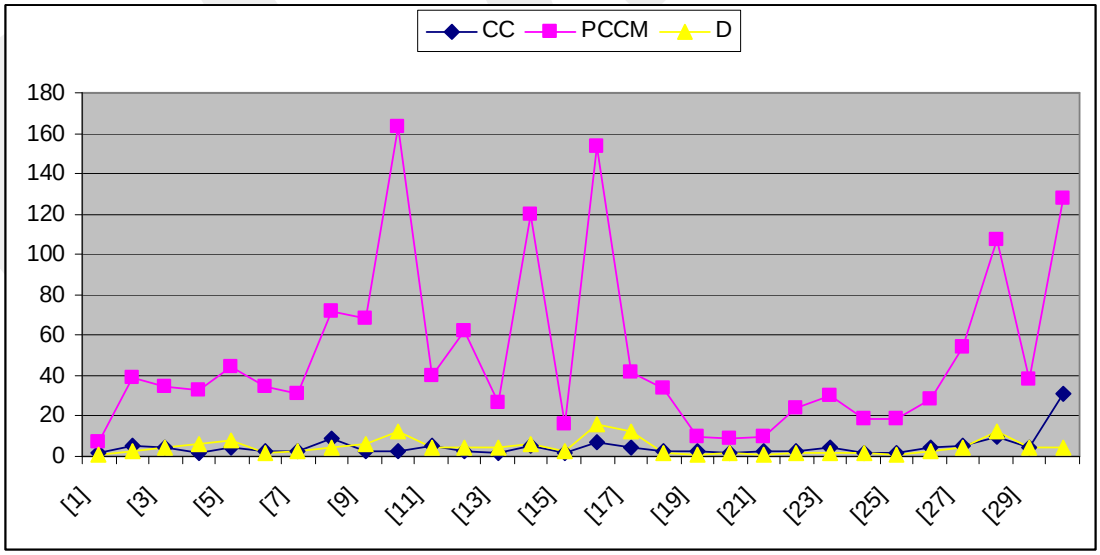


Figure 18: Relative Graph between CC, PCCM and Difficulty

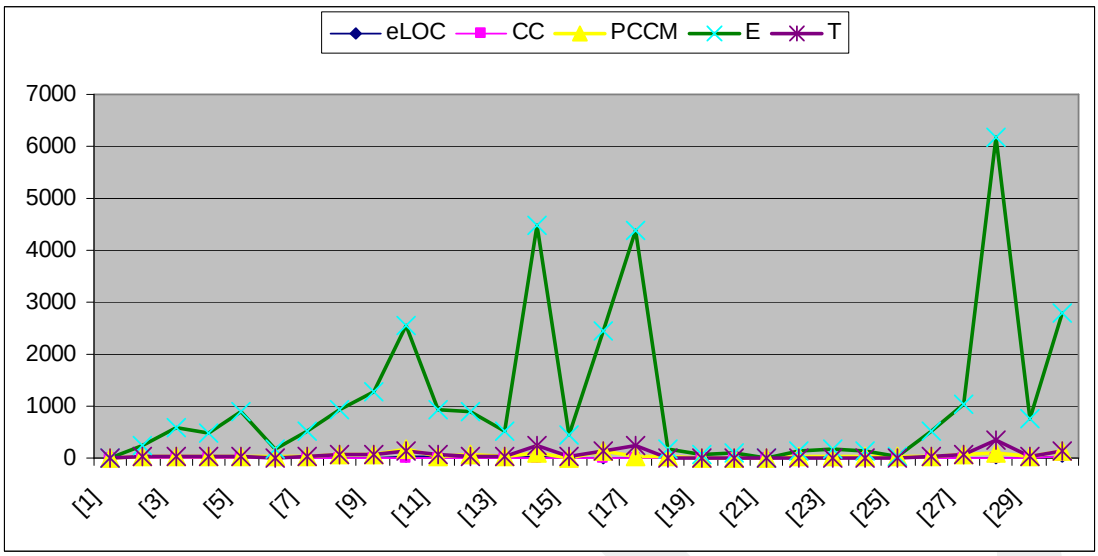


Figure 19: Relative Graph between eLOC, CC, PCCM, Effort and Time

GCRIS

CHAPTER 5
EXTENDING THE METRIC

5.1 Introduction

Previously, a new metric was proposed for the procedural parts of multi-paradigm codes. The research is extended by considering also the OO parts of the codes. This means the extended metric combines procedural and OO factors. This leads to the fact that the proposed metric can also be used both for procedural languages and OO languages. The cognitive aspect is included, too, as in PCCM. However there are some major differences which are:

- The extended metric Multi-Paradigm Complexity Measurement (MCM) is not applicable only for procedural languages, but also OO and multi-paradigm languages.
- Together with MCM, Function Point (FP) is recommended to be used. Thus, differently from PCCM, functional complexity too is measured and hence the code quality can be found.
- MCM proposes two ways of measurement; a formula which makes a detailed but exhaustive measurement, and another formula which eliminates details and yields a simpler measurement tool.

5.2 Multi-Paradigm Complexity Measurement

Firstly the factors which are responsible for the complexity of multi paradigm language are summarised.

Factors of Complexity of Procedural Part:

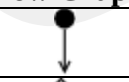
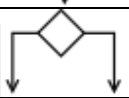
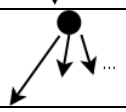
- Variables and constants
- Basic Control Structures

Factors of Complexity of OO Part:

- Attributes and constants,
- Basic Control Structures; and
- Classes

In order to cover a wider aspect of programming features Basic Control Structures are simplified which has already been defined above. The new version of BCS can be seen below.

Table 12: BCS for MCM

Category	CWU	Flow Graph
Sequence	1	
Condition	2	
Nested sub-condition	1	[Figure 2, 3]
Loop	3	
Nested sub-loop	2	[Figure 5, 6]
Module call	2	
Recursion	3	
Exception	1	

It is important to note that while calling a module, '2' is added to the total CWU of the called module [80][50]. Hence, coupling factor is included inside the structure.

In the first part of the research, number of variables, operators, and constants are multiplied with structures. However, in the second stage of the research it is not necessary to multiply variables, operators or constants with structures. Because, otherwise the metric would force the programmers to be confused in too many details. Due to later addition of numerous factors that affect the complexity, it is decided that

following similar approach would totally make up an extremely complex metric. Thus, even though it is still possible to use PCCM, a simpler version of procedural complexity is also proposed.

The metric is developed in a way that it can measure OO and procedural parts separately. However, some programs may not cover OO features in a code. Then, 0 should be assigned to the parts that are not related with OO paradigm.

The Metric

According to the further investigations of the complexity factors the study proposes a metric for multi-paradigm codes as below.

Multiparadigm Complexity Measurement (MCM):

$$MCM = CIclass + CDclass + PCCM \quad \textbf{(1)}$$

Where, **CIclass** = Complexity of Inherited Classes

CDclass = Complexity of Distinct Class

PCCM = Procedural Complexity

Although, PCCM measures the procedural complexity, it is assumed that using PCCM is difficult. MCM includes various complexity factors. PCCM, being a part of MCM, would make the metric too complex and too difficult to apply, it is thought. Therefore, it is recommended that Cprocedural is used in MCM instead of PCCM. Cprocedural is given in (5). However, it is possible to use MCM with PCCM for more detailed measurement.

$$MCM = CIclass + CDclass + Cprocedural \quad \textbf{(1)}$$

Cprocedural = Procedural Complexity

All these factors are defined as follows:

Cclass can be defined as complexity of a class. Cclass takes a major role in the calculation of both CDclass and CIclass. For example, for calculating CIclass, CDclass first, it is needed to calculate Cclass. Cclass is defined as,

$$Cclass = W(attributes) + W(variables) + W(structures) + W(objects) - W(cohesion) \quad \textbf{(2)}$$

Where, **Cclass** = Complexity of Class

The reason of subtraction of cohesion is that it reduces the complexity and thus it is desirable from the point of view of software developers [58].

Where, weight of attributes or variables is defined as:

$$W(variables\ or\ attributes) = 4 * AND + MND \quad (2.1)$$

Where, AND = Number of Arbitrarily Named Distinct Variables/Attributes

MND = Number of Meaningfully Named Distinct Variables/Attributes

Weight of structure W(structures) is defined as:

$$W(structures) = W(BCS) \quad (2.2)$$

Where, BCS are basic control structure.

Weight of objects Weight(objects) is defined as:

$$W(objects) = 2 \quad (2.3)$$

Creating an object is counted as 2, because while creating an object constructor is automatically called. Thus, coupling occurs. Therefore, it is the same as calling a function or creating an object. Here it is meant to be the objects created inside a class. Moreover, a method that calls another method is another cause of coupling, but that fact is added to MCM value inside Weight(structures).

Weight of cohesion is defined as:

$$W(cohesion) = MA / AM \quad [58] \quad (2.4)$$

Where, MA = Number of methods where attributes are used

AM = Number of attributes used inside methods

While counting the number of attributes there is no any importance of AND or MND.

CIclass can be defined as;

There are two cases for calculating the complexity of the Inheritance classes depending on the architecture:

- If the classes are in the same level then their weights are added.
- If they are children of a class then their weights are multiplied due to inheritance property.

If there are m levels of depth in the object oriented code and level j has n classes then the Cognitive Code Complexity (CCC) [51] of the system is given as

$$CIclass = \prod_{j=1}^{m}\left[\sum_{k=1}^{n} CC_{jk}\right] \quad \textbf{(3)}$$

CDclass can be defined as;

$$CDclass = Cclass(x) + Cclass(y) + \ldots \quad \textbf{(4)}$$

Note: All classes, which are neither inherited nor derived from another, are parts of Cdclass even if they have caused coupling together with other classes.

Cprocedural can be defined as;

$$Cprocedural = W(variables) + W(structures) + W(objects) - W(cohesion) \quad \textbf{(5)}$$

weight of variable W(variable) is defined as:

$$W(varialbes) = 4 * AND + MND \quad (5.1)$$

The variables are defined globally.

Weight of structure W(structures) is defined as:

$$W(structures) = W(BCS) + object.method \quad (5.2)$$

Where, BCS are basic control structure, and those structures are used globally. 'object.method' is calling a reachable method of a class using an object. 'object.method' is counted as 2, because it is calling a function written by the programmer. If the program consists of only procedural code, then the weight of the 'object.method' will be 0.

Weight of objects W(objects) is defined as:

$$W(objects) = 2 \quad (5.3)$$

Creating an object is counted as 2, as it is described above (2.3). Here it is meant to be the objects created globally or inside any function which is not a part of any class. If the program consists of only procedural code, then the weight of the 'objects' will be 0.

$$W(cohesion) = NF / NV \quad (5.4)$$

Where, NF is number of functions, and NV means number of variables. Coupling is added inside W(structures) as mentioned in the beginning of the description of the metric.

After completing the calculation of MCM, the program's Function Point value is also needed to be calculated in order to include functional complexity in the proposed metric as it is known. Function Point is a metric that is focused on functionality of software. It is well established and very popular. There are also other metrics to understand the functional aspect of software. For example, estimating with use case points is another metric type to measure the functional complexity [13]. However, FP is chosen for its ease of use and popularity.

Function Point [58]

To measure FP, there are 14 questions to be answered:

1. Does the system require backup and recovery?
2. Are the specialised data communications required to transfer information to or from the application?
3. Are there distributed processing functions?

4. Is performance critical?
5. Will the system run in an existing, heavily utilised operational environment?
6. Does the system require on-line data entry?
7. Does the on-line data entry require the input transaction to be built over multiple screens or operations?
8. Are the ILFs updated on-line?
9. Are the inputs, outputs, files, or inquiries complex?
10. Is the internal processing complex?
11. Is the code designed to be reusable?
12. Are conversion and installation included in the design?
13. Is the system designed for multiple installations in different organisations?
14. Is the application designed to facilitate change and for ease of use by the user?

Above questions should be numbered from 1 (lowest) to 5 (highest).

After answering above questions, the following table should be filled:

Table 13: Function Point

Information Domain Value	**Weighting factor**				
	Count	**Simple**	**Average**	**Complex**	**Total**
EIs	x	3	4	6	
EOs	x	4	5	7	
EQs	x	3	4	6	
ILFs	x	7	10	15	
EIFs	x	5	7	10	
Count Total					

EIs: External Inputs

EOs: External Outputs

EQs: External Inquiries

ILFs: Internal Logical Files

EIFs: External Interface Files

FP = count total x [0.65 + 0.01 x $E(F_i)$]

Then;

CQ = (FP / MCM) * 10,000 **(6)**

Where, **CQ** = Code Quality.

Having higher FP value and a lower MCM value gives higher code quality. Because, by developing cognitively less complex code which yields more functional program shows the efficiency. Thus, in order to obtain code quality, FP is divided by MCM. CQ also provides evidence on the significance of programming languages as demonstrated in the last paragraph of the next section titled "Comparative Study".

The reason for using FP can be explained with an example. For example, there are two programs (*Program1* and *Program2*) and a metric. The metric (*Metric*) measures only the cognitive complexity of a code. *Program1*'s complexity is measured as *X* and *Program2*'s complexity is found to be *Y*. Then, if *X* is greater than *Y*, according to the *Metric*, it is possible to comment that *Program1* is more complex than *Program2*. What the meaning of 'being more complex' is, may not be clear. It is not always obvious whether it is something desirable or undesirable. This fact consequently brings about some possibilities:

1. *Program1* has lots of complex functionalities. *Program2* is a very little program. For this reason *Program1* is more complex than *Program2*. Both of the programs are written efficiently, but *Program1* is bigger than *Program2*.
2. *Program1* has more complex functionalities than *Program2*, but its code is uselessly complex. In other words, even though *Program1* is more functional, the code is developed inefficiently.
3. *Program1* is less functional than *Program2*. *Program2* is capable of accomplishing more complex tasks. However, due to *Program1*'s code's inefficiency, the *Metric* value shows *Program1* as more complex.

If two programs are compared with CQ, and the first program is found to have higher CQ value than the second one, it can be said that the first program is a more efficient

program. Consequently, merely obtaining the code complexity value, it is difficult to comment on the comparison and reach to a conclusion. Because of this, the need for also a functional metric is indispensable.

The division of FP by MCM, usually gives an extremely small number which may be difficult to make decisions on. Therefore, it may be suitable to multiply the division by 10,000, in order to make the calculation easier to understand.

5.3 Demonstration of the Metric

Object-oriented paradigm is a programming paradigm. So, it is expected that the metric would give the same result for all languages which support object-oriented paradigm. In procedural paradigm, each programming language has its own merit. For that reason, logically it is not realistic to make a calculation the same for procedural codes written in different languages.

For empirical validation of the metric, firstly, a code is measured which covers cohesion, coupling, inheritance, polymorphism, attributes, methods, variables, object-oriented features, procedural lines of code, etc. The code, which covers most of possible coding features, is tried in three different languages. For an old multi-paradigm language [8], C++ is used. For a modern OO language, Java [71] is used. For a modern and affluent featured multi-paradigm language, Python [32] is used. Also, its UML figure is given below. The metric is developed for multi-paradigm languages. For multi-paradigm covers both procedural and OO paradigms, the proposed formula can be applied also for procedural and OO languages.

The below program consists of classes of shapes. The root class is Shapes. Other classes are derived from it. Colour class is an outside of the inheritance, but its method is called by other classes.

According to the proposed metric Cclass, CIclass, CDclass, Cprocedural values of the system are calculated. It is worth to mention that during the calculation of complexity of inheritance, CIclass should be carefully computed. The complexity of the classes at the

same level should be added and those values should be multiplied with their parent classes, as shown in the following computation.

Below class diagram's code is given in Appendix B. The code of this diagram was implemented in three different languages; Python, C++, and Java.

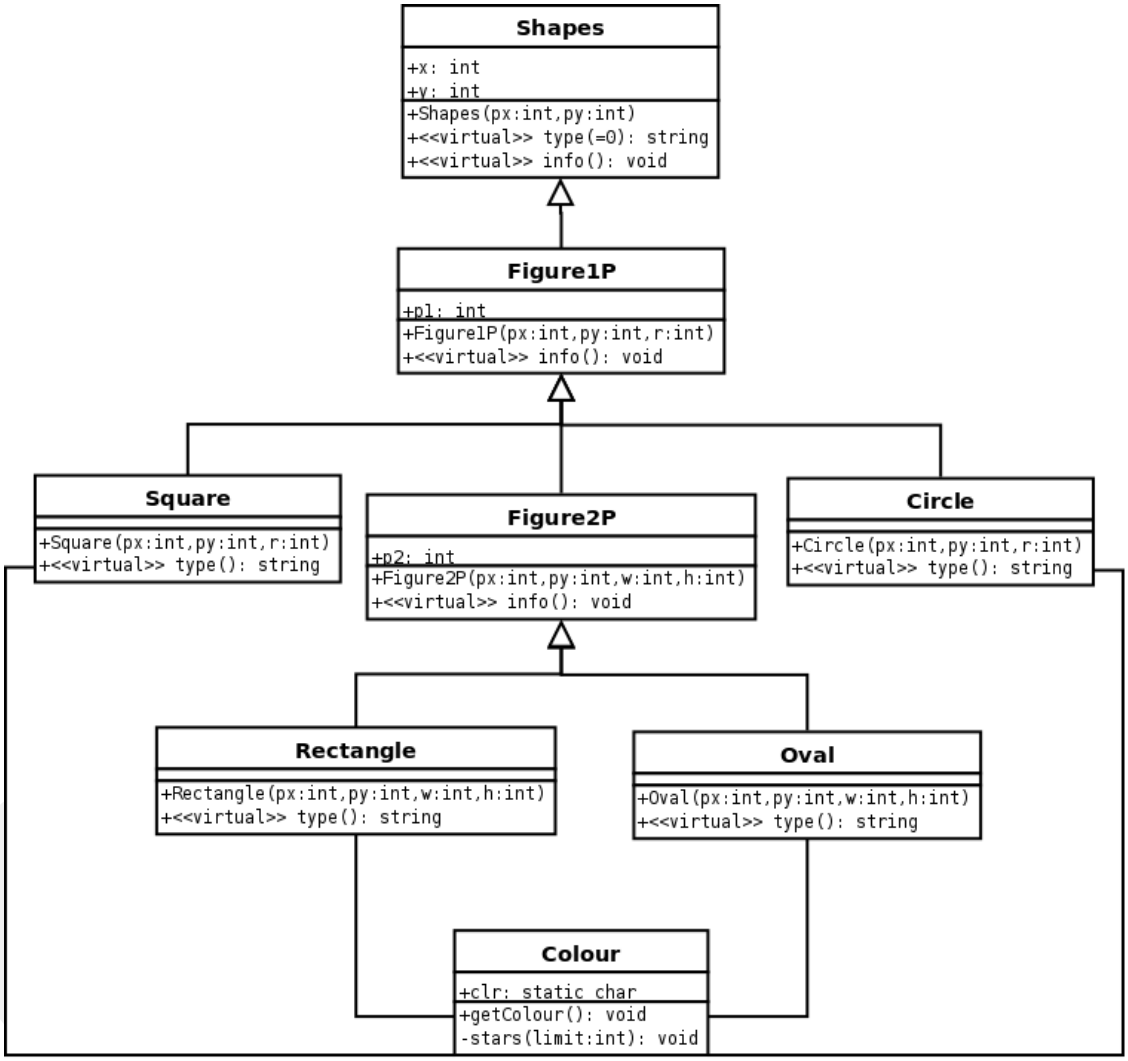


Figure 20: Shapes – Class Diagram

All those three languages had closely similar results. The demonstration is given in the following tables 14-19.

Calculating in Python:

Table 14: Class Complexity of Shapes in Python

class	att	str	var	obj	MA	AM	cohesion	Comp.
Colour	0	33	2	0	0	0	0	35
Shapes	2	6	0	0	2	2	1	7
Figure1P	1	8	0	0	2	1	2	7
Square	0	27	0	2	0	0	0	29
Circle	0	27	0	2	0	0	0	29
Figure2P	1	11	0	0	1	1	1	10
Rectangle	0	27	0	2	0	0	0	29
Oval	0	27	0	2	0	0	0	29

Table 15: Procedural Complexity of Shapes in Python

Non-Class	var+str+obj	Complexity
Cprocedural	24	24

CIclass = Shapes*(Figure1P * (Square + Circle + Figure2P * (Rectangle + Oval)))

= 7 * (7 * (29 + 29 + 10 * (29 + 29)))

= **31262**

CDclass = **35**

Cprocedural = 24

MCM = CIclass + CDclass + Cprocedural

= 31262 + 35 + 24

= 31321

Calculating in C++:

Table 16: Class Complexity of Shapes in C++

class	att	str	var	obj	MA	AM	cohesion	Comp.
Colour	0	33	2	0	0	0	0	35
Shapes	2	6	0	0	2	2	1	7
Figure1P	1	8	0	0	2	1	2	7
Square	0	27	0	2	0	0	0	29
Circle	0	27	0	2	0	0	0	29
Figure2P	1	11	0	0	1	1	1	10
Rectangle	0	27	0	2	0	0	0	29
Oval	0	27	0	2	0	0	0	29

Table 17: Procedural Complexity of Shapes in C++

Non-Class	var+str+obj	Complexity
Cprocedural	32	32

CIclass = Shapes * (Figure1P * (Square + Circle + Figure2P * (Rectangle + Oval)))

= 7 * (7 * (29 + 29 + 10 * (29 + 29)))

= **31262**

CDclass = **35**

Cprocedural = 32

MCM = CIclass + CDclass + Cprocedural

= 31262 + 35 + 32

= 31329

Calculating in Java:

Table 18: Class Complexity of Shapes in Java

class	att	str	var	obj	MA	AM	cohesion	Comp.
Colour	0	33	2	0	0	0	0	35
Shapes	2	6	0	0	2	2	1	7
Figure1P	1	8	0	0	2	1	2	7
Square	0	27	0	2	0	0	0	29
Circle	0	27	0	2	0	0	0	29
Figure2P	1	11	0	0	1	1	1	10
Rectangle	0	27	0	2	0	0	0	29
Oval	0	27	0	2	0	0	0	29

Table 19: Procedural Complexity of Shapes in Java

Non-Class	var+str+obj	Complexity
Cprocedural	71	71

CIclass = Shapes * (Figure1P * (Square + Circle + Figure2P * (Rectangle + Oval)))

= 7 * (7 * (29 + 29 + 10 * (29 + 29)))

= **31262**

CDclass = **35**

Cprocedural = 71

MCM = CIclass + CDclass + Cprocedural

= 31262 + 35 + 71

= 31368

From the above example it is possible to observe that class complexity values are the same in all programming languages. The OO part is the same for three of the languages. Only difference is seen in the procedural part. It is very natural to have slightly different values depending on the programming language, because each language has its own merit, simplicity and functionality. Hence, it can be said that the proposed metric is platform-independent.

Calculation was made as;

1. Complexity of each class was calculated. Attributes, methods, variables, objects, structures, and cohesion were included.
2. Complexity of procedural structure was calculated. Variables, objects, structures, functions, and the main function were included.
3. Classes were separated as a part of inheritance and distinct.
4. Complexity of inheritance was calculated. Super class was multiplied by the summation of the classes which are derived from it.

Complexity of inheritance, complexity of distinct class, and complexity of procedural structure were summed to reach the result of MCM.

The data obtained from the above experimentations provides some valuable information regarding the metrics as well as the features of the languages. It can be observed that the metric values for classes in all three languages are same and are: 35, 7, and 7, 29, 29, 10, for classes Colour, Shapes, Figure1P, Square, Circle, Figure2P, Rectangle, and Oval. These values proved that the proposed metric is language-independent. However the complexity for the whole system in Python, C++ and Java are different and are 31321, 31329 and 31368 respectively. Here it is important to note that at the class level the complexities are the same and the differences occur only at the main program.

Next, Function Point should be calculated.

Function Point:

Table 20: Function Point Calculation of Shapes

Information Domain Value	Weighting factor				
	Count	**Simple**	**Average**	**Complex**	**Total**
EIs	1	3	**4**	6	4
EOs	1	4	**5**	7	5
EQs	0	3	4	6	0
ILFs	0	7	10	15	0
EIFs	0	5	**7**	10	0
Count Total					9

FP questions:

1. Does the system require backup and recovery? 0
2. Are the specialised data communications required to transfer information to or from the application? 0
3. Are there distributed processing functions? 0
4. Is performance critical? 0
5. Will the system run in an existing, heavily utilised operational environment? 0
6. Does the system require on-line data entry? 0
7. Does the on-line data entry require the input transaction to be built over multiple screens or operations? 0
8. Are the ILFs updated on-line? 0
9. Are the inputs, outputs, files, or inquiries complex? 2
10. Is the internal processing complex? 3
11. Is the code designed to be reusable? 4
12. Are conversion and installation included in the design? 0
13. Is the system designed for multiple installations in different organisations? 0
14. Is the application designed to facilitate change and for ease of use by the user? 1

FP = 9 x [0.65 + 0.01 x 10]

FP = 6.75

After finding both MCM and FP values Code Quality (CQ) should be measured as:

CQ = FP / MCM

CQ(C++) = (6.75 / 31329) * 10000 => 2.15455

CQ(Python) = (6.75 / 31321) * 10000 => 2.15510

CQ(Java) = (6.75 / 31368) * 10000 => 2.15187

Through looking at the results, it is understood that the most efficient way of programming the above specified class diagram is writing the code in Python language. The least efficient language is Java for this example. It should be noted that these number do not claim that one language is better than the other one. Each language has its own merit. For one requirement, a specific language may have an advantage of efficiency, for another requirement, other language may be more suitable. The above program is very simple. For this reason, even though being much older C++ may be more advantages than Java. If a big program was under consideration, Java might have much higher CQ value than C++ due to being a more modern language.

5.4 Comparative Study

The well known CK metric suite is also applied for the example under consideration (Figure 20) The metric values for all the CK metrics with MCM values are summarised as follows (Table 21).

Table 21: Comparison between Metrics

Class / *Metric*	Shapes	Figure1P	Square	Circle	Figure2P	Rectangle	Oval	Colour
WMC	3	2	2	2	2	2	2	2
RFC	3	5	7	7	7	9	9	2
DIT	0	1	2	2	2	3	3	0

NOC	1	3	0	0	2	0	0	0
LCOM	2	2	0	0	2	0	0	0
CBO	0	1	1	1	1	1	1	0
MCM	7	7	29	29	10	29	29	35

In Table 21 the comparison of metrics are given. The given metrics except MCM are taken from Chidamber and Kemerer's Metrics Suite [58]. WMC stands for Weighted Methods per Class which is the sum of the complexities of all methods of a class. RFC stands for Response For a Class which is the number of methods that can be triggered by a message sent to an object. DIT stands for Depth of Inheritance Tree which is the maximum length in between the node and the root. NOC stands for Number of Children which is the number of subclasses. LCOM stands for Lack of Cohesion in Methods which is the same with the cohesion part of the proposed MCM. CBO stands for Coupling Between Object Classes which is the number of coupled classes.

Figure 20 shows the graph diagram of the compared metrics. The difference between Figure1P and Square could not be realised by WMC and CBO. Square, Circle, Rectangle and Oval have the same content, however except CBO and MCM all metrics show them different. Colour class has a nested loop and 4 conditions. Moreover, one of its methods calls its other method. Other classes have more primitive codes in comparison with Colour class, despite that class is shown to be the lowest complexity in all metrics proposed in the suite. However, MCM could handle the situation by recognising the difference and assigning the highest value to the class. Generally it is shown that all the given classes have more or less the same complexity. On the other hand, by making more sensitive calculation MCM is able to differentiate their complexities.

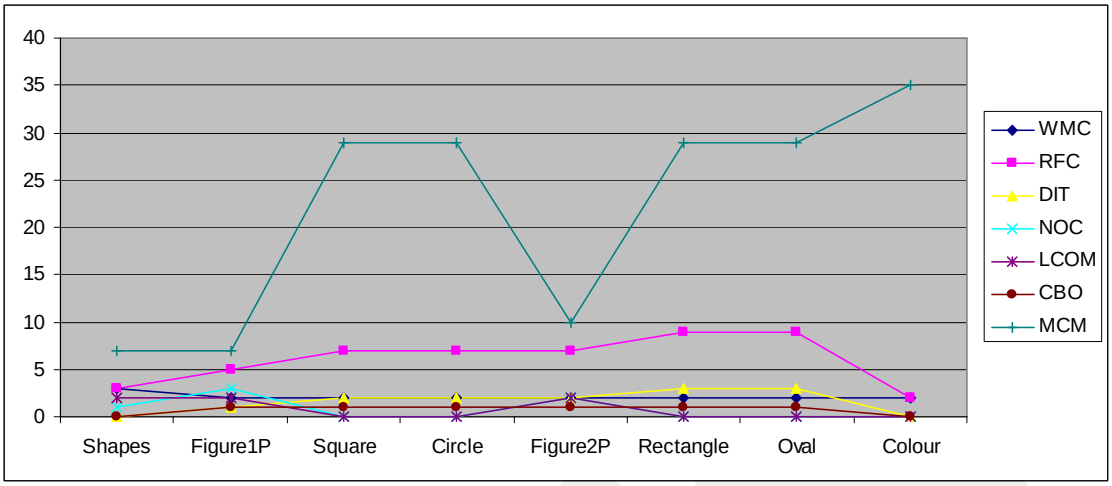


Figure 21: Comparison of Metrics

5.5 Empirical Validation

Pyso

The practical usefulness of a new measure cannot be proved without the proper empirical validation which includes the applicability of the metric on real projects. For this purpose an open source project is selected which is available on the web. It is believed that the open source code is more beneficial for the readers because they can also evaluate the project in the same way as the original author do. The project is a cross-platform set of Python modules designed for writing video games [70]. It includes computer graphics and sound libraries designed to be used with the Python programming language. It is built over the Simple DirectMedia Layer (SDL) library [54], with the intention of allowing real-time computer game development without the low-level mechanics of the C programming language and its derivatives. This is based on the assumption that the most expensive functions inside games (mainly the graphics part) can be completely abstracted from the game logic in itself, making it possible to use a high-level programming language like Python to structure the game.

The complexity of each class is estimated independently. Classes are coupled through two ways: through inheritances and message calls. The inheritance hierarchies of the classes coupled through the inheritance are shown in Figure 21 and 22. Some classes are independent and therefore not affected by inheritance.

In Table 22 the complexity of each class is shown. In the first column the name of the class is given. The metric values for different parameters which affect the complexity of the class i.e. attributes, variables, structures, objects and cohesion are given in column 2 – 7. Cclass is calculated by the equation 2.

A brief description: Pyso is a game library written in Python, by James Tauber. It is based on Pygame which is a platform for video games [54][55][70]. For being a library, it consists of numerous classes.

Table 22: Pyso – Classes

Class	att	str	var	obj	MA	AM	cohesion	Comp.
GameObject	1	1	0	0	1	1	1	3
MapObject	7	27	12	0	1	6	0.1	46.1
Level	2	18	10	0	2	2	1	31
level_zero	1	1	0	0	1	2	0.5	2.5
level_one	3	52	4	2	1	2	0.5	61.5
ImagedObject	2	6	0	0	0	0	0	8
PropTile	0	0	1	0	0	0	0	1
ActorTile	2	3	1	2	0	0	0	8
Sphere	9	9	7	2	2	5	0.4	27.4
Background	1	2	2	0	0	0	0	5
FloorTile	0	0	1	0	0	0	0	1
Cement	3	2	0	0	0	0	0	5
Grass	3	8	1	0	1	1	1	13
Curb	3	2	0	0	1	1	1	6
Void	3	8	3	0	1	1	1	15
Widget	3	24	5	0	4	6	0.6	32.6
Button	10	32	6	0	6	9	0.6	48.6
DirectionButtons	2	15	4	0	2	2	1	22
ViewPort	5	9	0	0	3	5	0.6	14.6
GameObjects	3	7	0	0	3	3	1	11
Clock	11	24	4	0	4	7	0.5	39.5
State	0	0	0	0	0	0	0	1
CyclePath	8	50	6	0	4	8	0.5	64.5

Table 23: Pyso – Procedural

Non-Class	var+str+obj	Complexity
Cprocedural	1304	1304

The complexity of the non classes, defined as Cprocedural is due to global variables structures and objects and is computed in Table 23. It can be easily observed that global complexity also plays an important role in increasing the overall complexity.

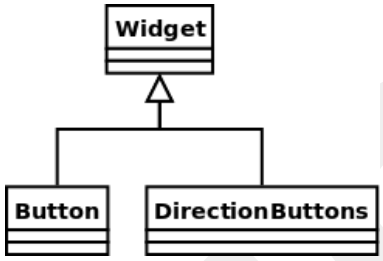


Figure 22: Pyso – Inheritance 1

There are different class hierarchy in this project. The first class hierarchy is shown in Figure 21. In this hierarchy two classes Button and DirectionButtons are on the same level and inherited from class Widget. Due to the effect of this inheritance the complexity of the class Widget is computed as follows;

Widget(Button + DirectionButtons)
=32.6(48.6 + 22)
=2301.5

Another class hierarchy which includes 15 classes are shown in Figure 22. The class Gameobject is at the top of the hierarchy. The complexities of each class under inheritance are given in Table 24. The complexity due to inheritance is computed as 276100.8. The demonstration of the calculation for inheritance is given in the following paragraphs.

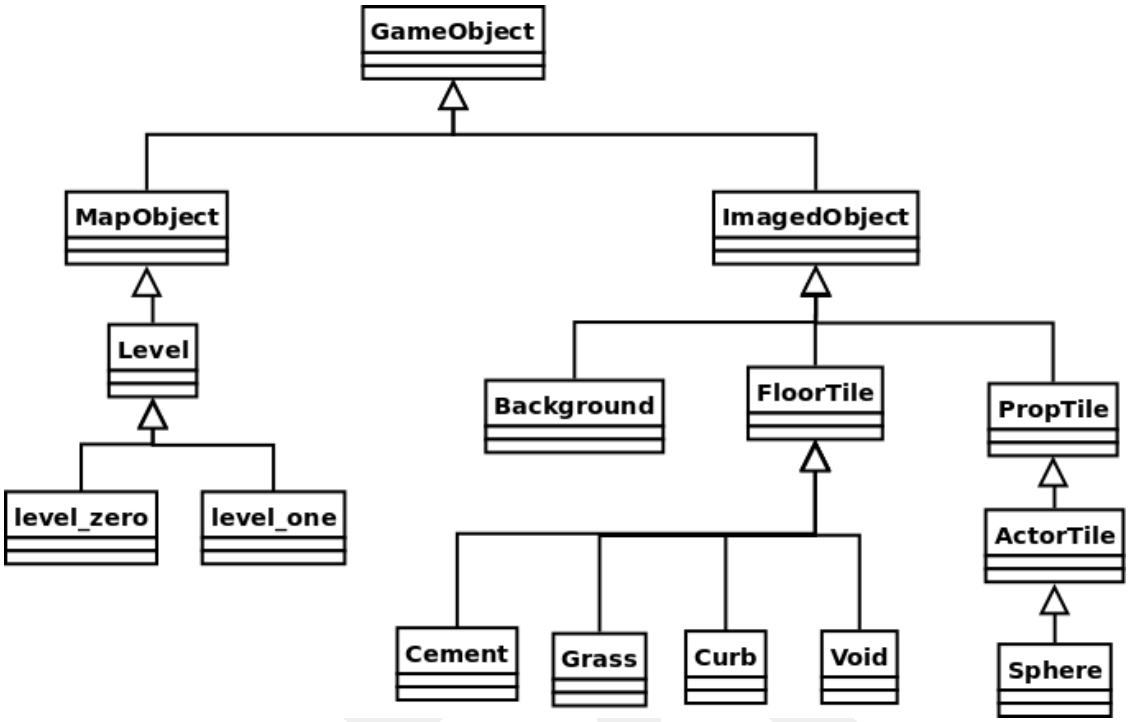


Figure 23: Pyso – Inheritance 2

Table 24: Pyso – Inherited Classes

Class	Complexity
Widget	32.6
Button	48.6
DirectionButtons	22
GameObject	3
MapObject	46.1
ImagedObject	8
Level	31
Background	5
FloorTile	1
PropTile	1
level_zero	2.5
level_one	61.5
Cement	5
Grass	13
Curb	6
Void	15
ActorTile	8
Sphere	27.4

GameObject[MapObject(Level(level_zero + level_one)) + ImagedObject(Background + FloorTile(Cement + Grass + Curb + Void) + Proptile(ActorTile(Sphere)))]

=3[46.1(31(2.5+61.5)) + 8(5 + 1(5 + 13 + 6 + 15) + 1(8(27.4)))]

=276100.8

Ciclass=276100.8+2301.5

=278402.3

Cdclass=130.6

Cprocedural=1304

MCM = Ciclass + Cdclass + Cprocedural

MCM = 279836.9

Table 25: Pyso – FP

Information Domain Value	**Weighting factor**				
	Count	**Simple**	**Average**	**Complex**	**Total**
Eis	0	3	4	6	0
Eos	9	4	**5**	7	45
Eqs	0	3	4	6	0
ILFs	0	7	10	15	0
EIFs	0	5	7	10	0
Count Total					45

FP questions:

1. Does the system require backup and recovery? 1
2. Are the specialised data communications required to transfer information to or from the application? 0
3. Are there distributed processing functions? 0
4. Is performance critical? 1
5. Will the system run in an existing, heavily utilised operational environment? 1
6. Does the system require on-line data entry? 0
7. Does the on-line data entry require the input transaction to be built over multiple screens or operations? 1
8. Are the ILFs updated on-line? 0

9. Are the inputs, outputs, files, or inquiries complex? 3
10. Is the internal processing complex? 3
11. Is the code designed to be reusable? 5
12. Are conversion and installation included in the design? 0
13. Is the system designed for multiple installations in different organisations? 0
14. Is the application designed to facilitate change and for ease of use by the user? 5

FP = 45 x [0.65 + 0.01 x 20]

FP = **38.25**

CQ = (FP / MCM) * 10000

CQ = (38.25 / 279836.9) * 10000

CQ = **1.36686**

The above computation indicates the applicability of the metric on real life applications. This also shows that not only one factor is responsible for the complexity of the whole code but there are several factors which plays the important role in increasing the overall complexity of the code. It is worth mentioning that all these factors are not new but up to now these factors are not unified for complexity calculation purpose. The complexities for all these factors like inheritance, coupling, methods are computed independently in the available complexity metric. It is the first attempt to unify all of them in a single metric.

In the section, the metric is applied on a project in Python language. However, a multi paradigm language may consist of other languages. By keeping this issue into consideration, the metric is applied on the projects written in C, C++ and JAVA. All the results on these projects are shown in Table 26. All the computations are given in Appendix C. The metric values for variety of projects in different languages prove its applicability on real projects. Brief descriptions of the projects are given below.

Chatting Application

This is an application developed in Java for chatting. The program is divided into two; client-side and server-side [63]. Inside this program there inheritance between classes are not used. Thought, it has simpler structure than other compared projects, it has higher level of functionality. For this reason it has the highest code quality. Its LOC is 1208.

Microprocessor Simulator

This is a simple 8085 simulator program developed in Java [65]. This project encompasses numerous nested loops. Its LOC is 2332. Due to its extremely complex structure, and simpler functionalities, it has one of the lowest CQ value.

Medical Record Keeping System

It is a small project developed in C. Its aim is to provide a facility to keep records of patients inside a polyclinic [64]. Its LOC is 3224. As expected from a medical record program, this code confers significant functionalities. Its structure is not very complex as seen from Table 26.

NeoMem

NeoMem provides user to store and organise various information in a cross between a word processor and database [52]. It is developed in C++. For being a huge project, 6 classes are chosen in between a vast amount of classes. All the classes are independent from each specified class. It could have different CQ value if the classes as a whole project had been measured.

TreeMaker

TreeMaker is an origami program developed in C++ [74]. All the classes are independent from each specified class. Similar to the previous project, this was too big to analyse the whole project. For that reason, 24 classes are chosen from the project and measured them.

Table 26: Comparison of Projects

	Project 1 Python (Pyso)	Project 2 Java (chatting Application)	Project 3 Java (Microprocessor simulator)	Project 4 C++ (Neomem)	Project 5 C++ (Tree Maker)	Project 6 C (Medical record keeping system)
MCM	279836.9	520.1	697458.9	492	2617	3916
FP	38.25	489.85	153.72	14.55	11.62	1646.45
CQ	1.36686	9418.38108	2.20400	295.73170	44.40198	4204.41777

In Table 26, comparison of projects is given. Project 1, 2, 3, and 6 are completely measured. However, only a small part of Project 4 and 5 is measured due to both being very large programs.

By measuring the given projects, the study measured approximately 30 classes from each mentioned OO language and a full project from a procedural language. The higher the Code Quality the better the code is. Increment of FP increases CQ, increment of MCM decrease CQ. This means when MCM is constant, higher level of functionality increases the code quality. Higher FP indicates more functionality, more merits. Similarly, when FP is constant, higher level of cognitive complexity decreases the code quality. Higher MCM indicates less comprehensibility, more difficult code maintenance. The most desired combination is higher FP and lower MCM.

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

In this study a complexity metric is proposed to include the factors that affect the cognitive complexity of a code. Initially, the proposed metric is formulated to manifest the factors of the procedural complexity. Later, OO complexity characteristics are added to extend the proposed metric to measure the complexity of codes written in a multi-paradigm programming language. The metric can also be applied to codes which are written merely in a procedural language or an OO language since multi-paradigm encompasses both of them.

In the first part, the research focused on most of the factors that affect the comprehensibility of a code. A cognitive complexity measurement metric which is called PCCM was proposed for procedural languages. PCCM has a dynamic structure such that its structure can be changed according to the needs of different procedural programming languages.

The comparative inspection of the implementation of PCCM versus eLOC, CC, and Halstead has shown that:

- PCCM makes more sensitive measurement, so that it enables developers to differentiate even small complexity differences between codes.
- Halstead's assumptions may sometimes mislead developers, whereas PCCM has the least amount of assumptions and those assumptions are based on cognitive aspects.
- CC was not able to make sensitive measurement; most of the similar codes had the same CC values. Similarly, eLOC, for being based on the lines of code, cannot distinguish different structures. Though, empirical validations have shown that PCCM was able to handle those issues.

- Among the specified metrics only PCCM includes cognitive effects. Besides that, only PCCM has a dynamic structure.

For having the mentioned features, it was possible to add OO complexity factors in the research and extend PCCM's structure to broaden its applicability in different paradigms. The reason for extending the metric is to widen the application area of the dynamic structure of cognitive complexity measurement so that it can be used not only for procedural but also for OO and more generally multi-paradigm codes.

The extended metric sustains PCCM's inherited feature of being dynamic, for its structure can be edited due to the needs of any procedural, OO or multi-paradigm programming language. Moreover, the metric should be used together with Function Point so that both cognitive and functional complexity aspects of a code could be calculated. By MCM, the code comprehensibility is measured, while the functionality is measured by FP. After combining MCM and FP in one metric, Code Quality was proposed.

After empirical validation it is obvious that CQ has a platform-independent structure. It means class complexities had the same CQ value in different OO languages. However, because of each language having its own programming merit, the procedural parts of the codes had different CQ values. The comparative case studies for CQ against other popular metrics such as WMC, RFC, DIT, NOC, LCOM, and CBO have shown that CQ has the following properties:

- combines many aspects in one measurement formula
- makes more sensitive calculation
- includes more complexity factors
- includes functional complexity besides cognitive complexity

The higher the CQ the better the code is. Increments of FP increase CQ, increments of MCM decrease CQ. This means when MCM is constant, the higher level of functionality increases the code quality. Higher FP indicates more functionality, more merits. Similarly, when FP is constant, the higher level of cognitive complexity decreases the

code quality. Higher MCM indicates less comprehensibility, more difficult code maintenance. The most desired combination is higher FP and lower MCM.

The CQ value does not contain an upper boundary, for there is no limit in the quality of a software system. For this reason, using CQ or any other metric only for once may not have impact on software quality improvement. However, if the proposed metrics are used continuously in a company, results may be more meaningful for the developers because in this case they can use it in comparison to the previous software products. Briefly, the proposed metrics are useful especially in comparing software products which are developed to accomplish more or less the same tasks. Small size companies usually develop software products of the same category. Thus, most of their products are similar to each other. The proposed metric can be suggested to be used to improve the quality in such companies.

All the existing metrics have their own merits. The goal of the thesis was not to criticise those metrics and claim their inabilities, but rather to understand their benefits and propose combined metrics based on some of them. There are also other metrics used for cost estimating such as Putnam's Model, Constructive Cost Model, and Esterling Time Study Model [1]; but they are out of the research scope of this thesis.

Further research is required to add more complexity factors and simplify the metric so that it becomes more practical. Although the study has tried to include most of those factors, it is possible to add more. After the research and empirical validation of the proposed metrics, some deficiencies about the metrics were also realised and they are as follows:

- Considering the complexity so broadly and detailed makes the measurement difficult to apply. Therefore, simplification may be one of the most essential needs of further research.
- Distinguishing MNV and ANV is sometimes subjective. In some cases it is very difficult to decide whether the variable or attribute should be inside the category of MNV or ANV.

- Usually, the weight of cohesion gives a very low value, as if it is not of much importance. Factors of cohesion may be improved so that those numbers decrease the complexity value in a more significant amount.
- For C which is a procedural language, what should be done for ‘struct’? This is not clear.
- To evaluate the functionality of a code, instead of FP, a more improved metric may be used.
- More empirical validations can be done. The extended metric can be applied on some other multi-paradigm languages such as Ruby and Perl. Also it can be tried on PHP, so that its availability can be understood for web programming languages, too. In addition, testing the metric shall be extended by applying it on big and whole projects, rather than choosing only some classes.
- Comments and indentation may be important factors of cognitive complexity, but those factors were excluded from the scope of this study.
- A better coding standard should be developed among a software team and the style of MNVs and ANVs should be defined by their cognitive choice in order to avoid subjectivity completely.

These deficiencies can be the source of further research. Although the proposed metric has some weaknesses as noted above, because of its major advantages, it can be considered a valuable contribution to the literature in this field, since combining several cognitive aspects with functional aspect is a new attempt in this area.

REFERENCES

[1] Arifoglu, A.: A Methodology For Software Cost Estimation, ACM Sigsoft, vol.18 no.2, 1993.

[2] AT&T Consumer Affiliate Network (last accessed 18.02.2010).
Available at: http://att-promotions.com/glossary.php

[3] Banker, R.D., Datar, S.M., Zweig, D.: Software Complexity and Maintainability CiteSeer Scientific Literature Digital Library and Search Engine.

[4] Barbacci, M.R., Klein, M.H., Longstaff, T.A., Weinstock, C.B.: Quality Attributes of a Software Architecture (last accessed 02.04.2010)
Available at: http://www.vrsiddhartha.ac.in/technology/quality.pdf

[5] Basci, D., Misra, S: 'Data Complexity Metrics for Web-Services' Advances in Electrical and Computer Engineering, Volume 9, Number 2, 2009, pp.9-15.

[6] Basci, D., Misra, S.: Measuring and Evaluating a Design Complexity Metric for XML Schema Documents' Code. Journal of Information Science and Engineering. Sep. 2009, pp.1415-1425.

[7] Basili, V.R.: Qualitative Software Complexity Models: A Summary. In Tutorial on Models and Methods for Software Management and Engineering. IEEE Computer Society Press, Los Alamitos, California, 1980.

[8] Bjarne Stroustrup's FAQ – 'When was C++ invented?' (last accessed 20.03.2010)
Available at: http://www2.research.att.com/~bs/bs_faq.html#invention

[9] Briand, L.C., Bunse, C., and Daly, J.W.: A Controlled Experiments for Evaluating Quality Guidelines on the Maintainability of Object-Oriented Design. IEEE Transactions on Software Engineering, vol. 27, (2001), 513-530.

[10] Bruegge, B., Dutoit, A.H.: Object-Oriented Software Engineering – Using UML, Patterns, and Java, 2nd International Edition, Prentice Hall, 2004.

[11] Budd, T.A.: Multiparadigm Data Structures in Leda, IEEE, 1992.

[12] Chidamber S.R., Kemerer, C.F.: A Metric Suite for object oriented design. IEEE Transactions Software Engineering, SE-6(1994)476-493.

[13] Cohn, M.: Estimating with Use Case Points (last accessed. 23.02.2010)
Available at: http://www.methodsandtools.com/archive/archive.php?id=25

[14] Confoo.Ca Web Techno Conference, (last accessed 14.11.2009)
Available at: http://www.confoo.ca/en

[15] Coplien, J.O.: Multi-paradigm Design for C++, Addison Wesley, 1999.

[16] Costagliola G., Tortora, G.: Class points: An approach for the size Estimation of Object-oriented systems. IEEE Transactions on Software Engineering, vol 31, 1(2005)52-74.

[17] Daeja Image Systems. (last accessed 18.02.2010).
Available at: http://www.daeja.com/support/glossary.asp

[18] Dannelly, R.S.: Winthrop University Lecture Notes in Computer Science (last accessed 16.05.2010)
Available at: http://faculty.winthrop.edu/dannellys/csci626/02_Definition.ppt

[19] Da-wei, E.: The Software Complexity Model and Metrics for Object-Oriented, IEEE, 2007.

[20] DeMarco, T.: Controlling Software Projects, Yourdon Press, New York, 1986

[21] Eclipse Metrics Plugin 1.3.8 (last accessed 23.02.2010)
Available at: http://metrics2.sourceforge.net/

[22] Fenton N. E. , Pfleeger, S. L.: Software Metrics: A Rigorous and Practical Approach, 2nd Edition Revised ed. Boston: PWS Publishing, 1997.

[23] Francalanci, C., Merlo, F.: The Impact of Complexity on Software Design Quality and Costs: An Exploratory Empirical Analysis of Open Source Applications (last accessed 16.03.2010)
Available at: http://is2.lse.ac.uk/asp/aspecis/20080122.pdf

[24] FrontRangePythoneersUc09, (last accessed 05.10.2009)
Available at: http://wiki.python.org/moin/FrontRangePythoneersUc09

[25] Galin, D.: Software Quality Assurance", Pearson Addison Wesley (2004)

[26] Garcia, E.: Software Metrics Through Fault Data From Empirical Evaluation Using Verification & Validation Tools. (Master Thesis) Texas Tech University. May 2008.

[27] Goodman, P.: Practical Implementation of Software Metrics, McGraw Hill, London, 1993

[28] Halstead, M.H.: Elements of Software Science. New York: Elsevier North-Holland, 1997.

[29] Hughes, B., Cotterell, M.: Software Project Management, 4th Edition. McGraw-Hill, 2006

[30] IEEE Computer Society: Standard for Software Quality Metrics Methodology. Revision IEEE Standard (1998)1061-1998.

[31] Ierusalimschy, R.: Programming with Multiple Paradigms in Lua (last accessed 22.03.2010)
Available at: http://www.inf.puc-rio.br/~roberto/docs/ry09-03.pdf

[32] Interview with Guido van Rossum (the developer of Python programming language) (July 1998) (last accessed 20.03.2010)
Available at: http://www.amk.ca/python/writing/gvr-interview

[33] Kan, S.H.: Metrics and Models in Software Quality Engineering, second edition, Addison-Wesley, 2003.

[34] Kaner, C., Bond, W.P.: Software Engineering Metrics: What Do They Measure and How Do We Know? 10th International Software Metrics Symposium, Metrics 2004

[35] Kearney, J.K.: Software Complexity Measurement. Communications of the ACM, vol. 29, p. 1044-1050 (1986)

[36] Kearney, J.K., Sedlmeyer, R.L., Thompson, W.B., Gray, M.A., Adler, M.A.: Software Complexity Measurement. vol. 29, no. 11, ACM, 1986.

[37] Kemerer, C.F.: Reliability of Function Points Measurement: A Field Experiment Communications of the ACM, vol. 36, (1993) 85-87.

[38] Knutson, C.D., Budd, T.A., Vidos, H.: Multiparadigm Design of a Simple Relational Database, V.35(12), ACM, 2000.

[39] Kushwaha, D.S., Misra, A.K.: Improved Cognitive Information Complexity Measure: A Metric that Establishes Program Comprehension Effort, Software Enginering Notes, September 2006, vol 31, no 5.

[40] LangPop 'Programming Language Popularity' (last accessed 20.03.2010)
Available at: http://www.langpop.com/

[41] Marco, L.: Measuring Software Complexity (last accessed. 23.02.2010)
Available at: http://cispom.boisestate.edu/cis320emaxson/metrics.htm

[42] Martin, S.: A Critique of Cyclomatic Complexity as a Software Metric. IEEE Software Engineering Journal, March 1988.

[43] McCabe, T.J.: A Complexity Measure. IEEE Transactions Software Engineering. 2(6): p.308-320, 1976

[44] McCabe, T.J., Watson, A.H.: Software Complexity, McCabe and Associates, Inc. (last accessed 17.03.2010)
Available at: http://www.stsc.hill.af.mil/crosstalk/1994/12/xt94d12b.asp

[45] Metrics 1.3.6 (last accessed 23.02.2010)
Available at: http://metrics.sourceforge.net/

[46] Miller, D.R.: 'James O. Coplien Multi-Paradigm Design for C++' Book review and commentary, January 2001, (last accessed 22.03.2010)
Available at: http://www.inkdrop.net/docs/multiParadigm.pdf

[47] Mills, E.E.: Software Metrics, Software Engineering Institute, Carnegie Mellon University, 1988.

[48] Misra, S., Akman, I.: A Complexity Metric based on Cognitive Informatics, Lecture Notes in Computer Science, Vol. 5009, pp.620-627, 2008.

[49] Misra, S., Akman, I.: A Model for Measuring Cognitive Complexity of Software, Springer-Verlag Berlin Heidelberg 2008, pp. 879-886.

[50] Misra S., Akman, I: Unified Complexity Metric: A measure of Complexity, Proc. Of National Academy of Sciences Section A. (2010)

[51] Misra, S., Akman, I.: Weighted Class Complexity: A Measure of Complexity for Object Oriented Systems, Journal of Information Science and Engineering, Vol.24, pp.1689-1708, November, 2008.

[52] NeoMem (last accessed 21.02.2010)
Available at: http://www.neomem.org/

[53] Pfleeger, S.L., Atlee, J.M.: Software Engineering – Theory and Practice, 3rd International Edition, Prentice Hall, 2006.

[54] Pygame (last accessed 21.02.2010).
Available at: http://www.pygame.org/news.html

[55] Python Game Programming Tutorial (last accessed 21.02.2010)
Available at: http://rene.f0o.com/mywiki/PythonGameProgramming

[56] Python Programming Language – Official Website. (last accessed 18.02.2010).
Available at: http://www.python.org/

[57] Resource Standard Metrics. (last accessed 18.02.2010).
Available at: http://msquaredtechnologies.com/m2rsm/docs/rsm_metrics_narration.htm

[58] Roger S. P.: Software Engineering – A practitioner's approach, 6th Edition. McGraw-Hill (2005)

[59] Rupy 2009, (last accessed 10.11.2009)
Available at: http://rupy.eu/

[60] Rusty Brick Web Definitions and Glossary (last accessed 18.02.2010).
Available at: http://www.rustybrick.com/definitions.php

[61] SciPy.in 2009, (last accessed 16.10.2009)
Available at: http://scipy.in/

[62] Scriptol - Popular Programming Languages (last accessed 23.03.2010)
Available at: http://www.scriptol.com/programming/choose.php

[63] Source Codes World – Chatting Application (last accessed 21.02.2010)
Available at: http://www.sourcecodesworld.com/source/show.asp?ScriptID=524

[64] Source Codes World – Medical Record Keeping System (last accessed 21.02.2010)
Available at: http://www.sourcecodesworld.com/source/show.asp?ScriptID=990

[65] Source Codes World – Microprocessor Simulator (last accessed 21.02.2010)
Available at: http://www.sourcecodesworld.com/source/show.asp?ScriptID=849

[66] Software QA and Testing Resource Centre (last accessed 16.05.2010)
Available at: http://www.softwareqatest.com/qatfaq1.html#FAQ1_13

[67] Software Quality Assurance (last accessed 16.05.2010)
Available at: http://www.sqa.net/iso9126.html

[68] Software Technology Support Centre Software Estimation, Measurement, and Metrics (last accessed 16.03.2010)
Available at: http://www.stsc.hill.af.mil/resources/tech_docs/gsam3/chap13.pdf

[69] Sommerville, I.: Software Engineering, 7th Edition, Addison Wesley, 2004.

[70] Tauber, J.: Pyso (last accessed 21.02.2010)
Available at: http://jtauber.com/pyso/

[71] TechMetrix Research 'Java Application Server Report', 1999.

[72] Testwell CMT++/CMT Java (last accessed. 23.02.2010)
Available at: http://www.verifysoft.de/en_cmtx.html

[73] TIOBE Software – The Coding Standards Company. Programming Community Index for February 2010. (last accessed 18.02.2010).

Available at: http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

[74] TreeMaker (last accessed 24.02.2010)
Available at: http://www.langorigami.com/science/treemaker/treemaker5.php4

[75] Van Roy, P.: Department of Computing Science and Engineering, Catholic University of Louvain (last accessed 17.03.2010)
Available at: http://www.info.ucl.ac.be/~pvr/cvvanroy.html

[76] Verisoft Technology. (last accessed 18.02.2010).
Available at: http://www.verifysoft.com/en_halstead_metrics.html

[77] Vessey, I., Weber, R.: Research on Structured Programming: An Empiricist's Evaluation. IEEE Transactions on Software Engineering, vol. SE-10, (1984) 394-407.

[78] Virtual Machinery. (last accessed 18.02.2010).
Available at: http://www.virtualmachinery.com/sidebar2.htm

[79] Wang, Y., Weber, R.: Toward a Theory of the Deep Structure of Information Systems. International Conference on Information Systems, Copenhagen, Denmark, (1990) 61-71.

[80] Wang, Y., Shao, J.: A New Measure of Software Complexity Based on Cognitive Weights. Can. J. Elec. Computer Engineering, (2003) 69-74.

[81] Westbrook, D.S.: A Multi-paradigm Language Approach to Teaching Principles of Programming Languages, 29th ASE/IEEE Frontiers in Education Conference, 1999, San Juan.

[82] Weyuker, E.: Evaluating Software Complexity Measures. IEEE Transactions on Software Engineering, vol. 14, (1988) 1357-1365.

[83] W3Schools. JavaScript Introduction (last accessed on 21.07.2009).
Available at: http://www.w3schools.com/js/js_intro.asp

[84] W3Schools. JavaScript Loop (last accessed 18.02.2010).
Available at: http://www.w3schools.com/js/js_loop_for.asp

[85] W3Schools. JavaScript Try…Catch Statement (last accessed 18.02.2010).
Available at: http://www.w3schools.com/js/js_try_catch.asp

APPENDICES

Appendix A

Web Sites of JavaScript Codes

[1] http://www.w3schools.com/js/tryit.asp?filename=tryjs_function1

[2] http://www.w3schools.com/js/tryit.asp?filename=tryjs_switch

[3] http://www.w3schools.com/js/tryit.asp?filename=tryjs_elseif

[4] http://www.w3schools.com/js/tryit.asp?filename=tryjs_fornext

[5] http://www.w3schools.com/js/tryit.asp?filename=tryjs_break

[6] http://www.w3schools.com/js/tryit.asp?filename=tryjs_array_for_in

[7] http://www.w3schools.com/js/tryit.asp?filename=tryjs_try_catch

[8] http://www.w3schools.com/js/tryit.asp?filename=tryjs_throw

[9] http://www.webdevelopersjournal.com/articles/jsintro3/js_begin3.html

[10] http://www.scriptci.com/modules.php?name=Downloads&d_op=getit&lid=287

[11] http://www.scriptci.com/modules.php?name=Downloads&d_op=getit&lid=885

[12] http://www.java2s.com/Tutorial/JavaScript/0500__Object-Oriented/Inheritance.htm

[13] http://javascript.internet.com/user-details/ip.html

[14] http://www.web-source.net/javascript_status_clock.htm

[15] http://www.buildwebsite4u.com/advanced/javascript-code.shtml

[16] http://www.sislands.com/coin70/week2/NestedLoops1.htm

[17] http://www.sislands.com/coin70/week3/slideshow.htm

[18] http://www.sislands.com/coin70/week7/counter.htm

[19] http://www.java2s.com/Tutorial/JavaScript/0080__Development/

Trycatchexception.htm

[20] http://www.java2s.com/Tutorial/JavaScript/0300__Event/
SeteventreturnvaluetofalseIE.htm

[21] http://www.java2s.com/Tutorial/JavaScript/0140__Function/Nestedfunctioncall.htm

[22] http://www.java2s.com/Tutorial/JavaScript/0140__Function/
Returnbooleanvaluefromfunction.htm

[23] http://www.java2s.com/Tutorial/JavaScript/0140__Function/
Passintegertofunction.htm

[24] http://www.java2s.com/Tutorial/JavaScript/0180__Math/Mathlog.htm

[25] http://www.java2s.com/Tutorial/JavaScript/0120__String/
ConvertStringtouppercase.htm

[26] http://examples.oreilly.com/jscript2/7.2.txt

[27] http://examples.oreilly.com/jscript2/14.3.txt

[28] http://examples.oreilly.com/jscript2/16.2.html

[29] http://examples.oreilly.com/jscript2/14.1.txt

[30] http://examples.oreilly.com/jscript2/6.2.txt

[31] http://www.web-source.net/javascript_status_clock.htm

[32] http://www.web-source.net/javascript_protect_content.htm

[33] http://www.web-source.net/javascript_popup_window3.htm

[34] http://www.web-source.net/javascript_disable_enter_key.htm

[35] http://www.web-source.net/javascript_typing_status.htm

Appendix B

Example Code in C

```
#include <iostream>
#include <string>
using namespace std;
// Colour
class Colour{
	void stars(int limit);
public:
	static char c;
	void getColour();
};
void Colour::getColour(){
	if (c=='s')
		cout<<"Yellow"<<endl;
	else if (c=='c')
		cout<<"Violet"<<endl;
	else if (c=='r')
		cout<<"Red"<<endl;
	else if (c=='o')
		cout<<"Orange"<<endl;
	else
		cout<<"White"<<endl;
	stars(5);
}
void Colour::stars(int limit){
	int outer_loop, inner_loop;
	for (outer_loop=limit; outer_loop>0; outer_loop--){
		for (inner_loop=1; inner_loop<=outer_loop; inner_loop++)
			printf("*");
		printf("\n");
	}
}
// -----------
char Colour::c;

class Shapes {
public:
	Shapes(int px, int py):x(px),y(py) {}
	int x, y; //position
	virtual string type() = 0;
	virtual void info() {
		cout << endl << "figure: " << type() << endl;
```

```
                cout << "position: x=" << x << ", y=" << y << endl;
        }
};

class Figure1P : public Shapes {
public:
        Figure1P(int px, int py, int r):p1(r),Shapes(px, py) {}
        int p1;
        virtual void info() {
                Shapes::info();
                cout << "property 1: p=" << p1 << endl;
        }
};

class Square : public Figure1P {
public:
        Colour *its_colour;
        Square(int px, int py, int r):Figure1P(px, py, r) {}
        virtual string type() {
                Colour::c='s';
                its_colour->getColour();
                return "square";
        }
};

class Circle : public Figure1P {
public:
        Colour *its_colour;
        Circle(int px, int py, int r):Figure1P(px, py, r) {}
        virtual string type() {
                Colour::c='c';
                its_colour->getColour();
                return "circle";
        }
};

class Figure2P : public Figure1P {
public:
        Figure2P(int px, int py, int w, int h):p2(h),Figure1P(px, py, w) {}
        int p2;
        virtual void info() {
                Figure1P::info();
                cout << "property 2: p=" << p2 << endl;
        }
};

class Rectangle : public Figure2P {
```

GCRIS

```
public:
	Colour *its_colour;
	Rectangle(int px, int py, int w, int h):Figure2P(px, py, w, h) {}
	virtual string type() {
		Colour::c='r';
		its_colour->getColour();
		return "rectangle";
	}
};

class Oval : public Figure2P {
public:
	Colour *its_colour;
	Oval(int px, int py, int w, int h):Figure2P(px, py, w, h) {}
	virtual string type() {

		Colour::c='o';
		its_colour->getColour();
		return "oval";
	}
};

// Freeing memory
void freeRAM(Shapes *objs[], int i){

	delete objs[i];

}
// --------------

int main(void) {
	Shapes **objs = new Shapes*[35];
	// creating objects
	objs[0] = new Circle(7, 6, 55);
	objs[1] = new Rectangle(12, 54, 21, 14);
	objs[2] = new Square(19, 32, 10);
	objs[3] = new Oval(43, 10, 4, 3);
	objs[4] = new Square(3, 41, 3);
	bool flag=false;
	do {
		cout << endl << "We have 5 objects with numbers 0..4" << endl;
		cout << "Enter object number to view information about it " << endl;
		cout << "Enter any other number to quit " << endl;
		char onum; // in fact, this is a character, not a number
		// this allows user to enter letter and quit...
		cin >> onum;
		// flag -- user have entered number 0..4
```

```
            flag = ((onum >= '0')&&(onum <= '4'));
            if (flag)
                objs[onum-'0']->info();
    } while(flag);

    for(int i=0;i<5;i++)
            freeRAM(objs,i);
    delete [] objs;
    return(0);
}
```

GCRIS

Appendix C

Empirical Validation

Chatting Application

Table 27: Chat Application – Classes

class	att	str	var	obj	MA	AM	cohesion	Comp.
CLIENT_INFO	2	2	0	0	1	2	0.5	3.5
MainFrame(S)	0	39	4	20	0	0	0	63
THBind	3	20	0	6	2	3	0.6	28.4
Client_P	2	102	5	14	2	2	1	122
MSG_RDR	0	8	0	6	0	0	0	14
S_Client	0	2	0	2	0	0	0	4
MainFrame(C)	3	30	4	16	1	3	0.3	52.7
Form	3	68	0	20	2	4	0.5	90.5
Sign_UP	0	18	0	16	0	0	0	34
Frame3	0	15	0	10	0	0	0	25
CHAT_WIN	2	16	0	12	2	2	1	29
MSG_READER	0	8	0	2	0	0	0	10
CMD_L	1	34	0	2	2	1	2	35

Table 28: Chat Application – Cprocedural

Non-Class	var+str+obj	Complexity
Cprocedural (S_CHAT)	9	9

All the classes are independent.

CDclass = 511.1

Cprocedural = 9

MCM = CIclass + CDclass + Cprocedural

MCM = 0 + 511.1 + 9

MCM = **520.1**

Table 29: Chat Application – FP

Information Domain Value	Weighting factor				
	Count	Simple	Average	Complex	Total
EIs	17	3	**4**	6	68
EOs	78	**4**	5	7	312
EQs	1	3	4	**6**	6

ILFs	0	7	10	15	0
EIFs	17	5	**7**	10	119
Count Total					505

FP questions:

1. Does the system require backup and recovery? 0
2. Are the specialised data communications required to transfer information to or from the application? 5
3. Are there distributed processing functions? 5
4. Is performance critical? 3
5. Will the system run in an existing, heavily utilised operational environment? 1
6. Does the system require on-line data entry? 3
7. Does the on-line data entry require the input transaction to be built over multiple screens or operations? 3
8. Are the ILFs updated on-line? 0
9. Are the inputs, outputs, files, or inquiries complex? 3
10. Is the internal processing complex? 2
11. Is the code designed to be reusable? 3
12. Are conversion and installation included in the design? 0
13. Is the system designed for multiple installations in different organisations? 0
14. Is the application designed to facilitate change and for ease of use by the user? 4

FP = 505 x [0.65 + 0.01 x 32]

FP = **489.85**

CQ = (FP / MCM) * 10000

CQ = (489.85 / 520.1) * 10000

CQ = **9418.38108**

GCRIS

Microprocessor Simulator

Table 30: Microprocessor Simulator – Classes

class	att	str	var	obj	MA	AM	cohesion	Comp.
UserRam	1	14	0	8	2	1	2	21
RunPro	1	5	0	0	2	1	2	4
Proceed1	22	2512	15	0	7	22	0.3	2548.7
Proceed	17	5454	25	2	13	17	0.7	5497.3
SetFlag	5	44	0	0	1	5	0.2	48.8
RunErrors	0	12	0	8	0	0	0	20
MemArea	4	18	4	6	2	4	0.5	31.5
InstArea	2	18	0	24	1	1	1	43
SetC	2	15	3	0	1	2	0.5	19.5
Check	1	22	8	0	1	1	1	30
Check1	2	50	0	0	1	2	0.5	51.5
About	0	10	0	12	0	0	0	22
Check2	4	16	2	0	1	4	0.2	21.8
Check3	2	18	2	0	1	2	0.5	21.5
Check4	3	27	2	0	1	3	0.3	31.7
FlagsWindow	0	10	2	8	0	0	0	20

Table 31: Microprocessor Simulator – Cprocedural

Non-Class	var+str+obj	Complexity
Cprocedural	0	0

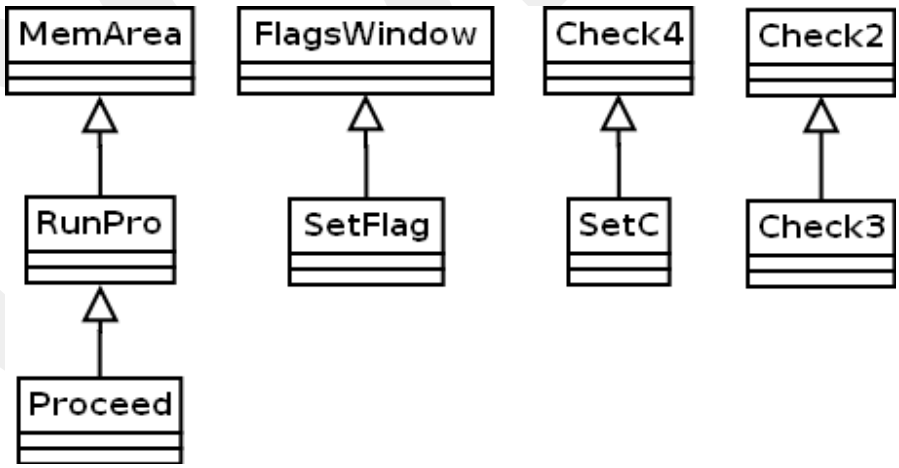


Figure 24: Microprocessor – Inheritance

MCM = CIclass + CDclass + Cprocedural

CIclass = 31.5(4(5497.3)) + 20(48.8) + 31.7(19.5) + 21.8(21.5)

CIclass = 692659.8 + 976 + 618.15 + 468.7

CIclass = 694722.7 + 2736.2

CDclass = 2736.2

MCM = 694722.7 + 2736.2

MCM = **697458.9**

Table 32: Microprocessor Simulator – FP

Information Domain Value	**Weighting factor**				
	Count	**Simple**	**Average**	**Complex**	**Total**
EIs	0	3	4	6	0
EOs	17	4	**5**	7	85
EQs	0	3	4	6	0
ILFs	0	7	10	15	0
EIFs	14	5	**7**	10	98
Count Total					183

FP questions:

1. Does the system require backup and recovery? 0
2. Are the specialised data communications required to transfer information to or from the application? 3
3. Are there distributed processing functions? 0
4. Is performance critical? 2
5. Will the system run in an existing, heavily utilised operational environment? 0
6. Does the system require on-line data entry? 0
7. Does the on-line data entry require the input transaction to be built over multiple screens or operations? 0
8. Are the ILFs updated on-line? 0
9. Are the inputs, outputs, files, or inquiries complex? 3
10. Is the internal processing complex? 2
11. Is the code designed to be reusable? 5
12. Are conversion and installation included in the design? 0
13. Is the system designed for multiple installations in different organisations? 0
14. Is the application designed to facilitate change and for ease of use by the user? 4

FP = 183 x [0.65 + 0.01 x 19]

FP = **153.72**

CQ = (FP / MCM) * 10000

CQ = (153.72 / 697458.9) * 10000

CQ **= 2.20400**

Medical Record Keeping System

Weight(variables) = 40

Statements: 3000

Loop: 592

Condition: 285

Cohesion: 56/40=1

Weight(structure) = 3876

Cprocedural = 3916

LOC= 3224

Table 33: Medical System – FP

Information Domain Value	**Weighting factor**				
	Count	**Simple**	**Average**	**Complex**	**Total**
EIs	5	**3**	4	6	15
EOs	428	**4**	5	7	1712
EQs	0	3	4	6	0
ILFs	0	7	10	15	0
EIFs	30	5	**7**	10	210
Count Total					1937

FP questions:

1. Does the system require backup and recovery? 4
2. Are the specialised data communications required to transfer information to or from the application? 2
3. Are there distributed processing functions? 3

4. Is performance critical? 2
5. Will the system run in an existing, heavily utilised operational environment? 3
6. Does the system require on-line data entry? 0
7. Does the on-line data entry require the input transaction to be built over multiple screens or operations? 0
8. Are the ILFs updated on-line? 0
9. Are the inputs, outputs, files, or inquiries complex? 1
10. Is the internal processing complex? 1
11. Is the code designed to be reusable? 1
12. Are conversion and installation included in the design? 0
13. Is the system designed for multiple installations in different organisations? 0
14. Is the application designed to facilitate change and for ease of use by the user? 3

FP = 1937 x [0.65 + 0.01 x 20]

FP = 1646.45

CQ = (FP / MCM) * 10000

CQ = (1646.45 / 3916) * 10000

CQ = **4204.41777**

NeoMem

Table 34: NeoMem – Classes

class	att	str	var	obj	MA	AM	cohesion	Comp.
Color	2	5	1	0	0	0	0	8
Clock	0	133	6	64	0	0	0	203
Crypto	0	111	5	14	0	0	0	130
RichEdiDocEx	0	2	0	2	0	0	0	4
StatusBarEx	0	73	2	16	0	0	0	89
Undo	0	48	0	10	0	0	0	58

Table 35: NeoMem – FP

Information Domain Value	Weighting factor				
	Count	Simple	Average	Complex	Total
EIs	0	3	4	6	0

EOs	0	4	5	7	0
EQs	0	3	4	6	0
ILFs	1	7	10	**15**	15
EIFs	0	5	7	10	0
Count Total					15

FP questions:

1. Does the system require backup and recovery? 4
2. Are the specialised data communications required to transfer information to or from the application? 4
3. Are there distributed processing functions? 0
4. Is performance critical? 2
5. Will the system run in an existing, heavily utilised operational environment? 1
6. Does the system require on-line data entry? 1
7. Does the on-line data entry require the input transaction to be built over multiple screens or operations? 1
8. Are the ILFs updated on-line? 0
9. Are the inputs, outputs, files, or inquiries complex? 0
10. Is the internal processing complex? 2
11. Is the code designed to be reusable? 5
12. Are conversion and installation included in the design? 5
13. Is the system designed for multiple installations in different organisations? 3
14. Is the application designed to facilitate change and for ease of use by the user? 4

FP = 15 x [0.65 + 0.01 x 32]

FP = 14.55

MCM = 492

CQ = (FP / MCM) * 10000

CQ = (14.55 / 492) * 10000

CQ = **295.73170**

GCRIS

TreeMaker

Table 36: TreeMaker – Classes

class	att	str	var	obj	MA	AM	cohesion	Comp.
SList	1	2	0	2	0	0	0	5
SDict	1	55	1	6	4	1	4	59
Iterator	0	11	0	4	0	0	0	15
IteratorDict	0	13	0	4	0	0	0	17
SIntList	0	2	0	4	0	0	0	6
SIntDict	1	46	0	4	4	1	4	47
ClassList	0	10	0	4	0	0	0	14
ClassSDict	2	528	2	0	1	2	0.5	531.5
SortedRefItems	0	2	0	4	0	0	0	6
ConfigEnum	0	12	0	6	0	0	0	18
HelpController	0	32	1	4	0	0	0	36
NewtonRaphson	5	184	5	0	3	5	0.6	193.4
ScaleOptimizer	2	71	4	2	2	2	1	78
Matrix	0	10	3	0	0	0	0	13
Array	7	78	8	0	12	7	1.7	91.3
Dpptr	2	15	0	0	5	2	2.5	14.5
Optimizer	3	10	0	2	3	3	1	14
CreaseOwner	0	11	0	2	0	0	0	13
NodeOwner	0	2	0	2	0	0	0	4
EdgeOwner	0	2	0	2	0	0	0	4
Poly	10	1155	2	0	22	10	2.2	1164.8
Vertex	12	178	12	0	18	12	1.5	200.5
VertexOwner	0	2	0	0	0	0	0	2
ConditionNodeFixed	5	68	0	0	15	5	3	70

MCM = 2617

Table 37: TreeMaker – FP

Information Domain Value	Weighting factor				
	Count	Simple	Average	Complex	Total
EIs	0	3	4	6	0
EOs	0	4	5	7	0
EQs	0	3	4	6	0
ILFs	2	**7**	10	15	14
EIFs	0	5	7	10	0
Count Total					14

FP questions:

1. Does the system require backup and recovery? 0

2. Are the specialised data communications required to transfer information to or from the application? 0
3. Are there distributed processing functions? 0
4. Is performance critical? 0
5. Will the system run in an existing, heavily utilised operational environment? 2
6. Does the system require on-line data entry? 0
7. Does the on-line data entry require the input transaction to be built over multiple screens or operations? 0
8. Are the ILFs updated on-line? 0
9. Are the inputs, outputs, files, or inquiries complex? 1
10. Is the internal processing complex? 1
11. Is the code designed to be reusable? 5
12. Are conversion and installation included in the design? 5
13. Is the system designed for multiple installations in different organisations? 0
14. Is the application designed to facilitate change and for ease of use by the user? 4

FP = 14 x [0.65 + 0.01 x 18]

FP = 11.62

CQ = (FP / MCM) * 10000

CQ = (11.62 / 2617) * 10000

CQ = **44.40198**

GCRIS

Appendix D

Other JavaScript Codes

Table 38: Other Scripts

Program	eLOC	CC	PCCM	Halstead			
				V	D	E	T
[/31]	21	4	65	370	8	2960	164
[/32]	12	5	36	148	1	148	8
[/33]	19	3	76	262	2	524	29
[/34]	6	4	32	271	2	542	30
[/35]	25	4	104	470	6	2820	156

GCRIS

Appendix E

Terms

Metric: It is way of measuring software code in a way or more.

Edge: It is a statement in a code.

Vertices: It specifies the change of statements.

Variable: It is a symbol that holds a value.

Object: It is an entity that carries the methods and attributes of its class.

Class: It is a template for creating an object.

Structure: It is notion of statements.

Coupling: It is the bound between modules

Cohesion: It indicates the togetherness of functionalities.

Inheritance: It is a way to form new classes depending on already defined classes.

Software Complexity: It is complexity that affects cost, effort, comprehensibility and time during development or maintenance.

Programming Paradigm: It is a style of computer programming used to solve some of the software engineering problems.

Flow Graph: It is a kind of diagram that shows the flow of a program.

Empirical Validation: It is used for experimental approval of a proposed solution.