A Study on the Cognitive Complexity Metric of a Software

A thesis submitted for the Degree of Doctor of Philosophy

D. R. Wijendra

University of Colombo School of Computing

2023

Declaration

Name of the student: Dinuka Rukshani Wijendra
Registration number: PhD/PT/2021/020
Name of the Degree Programme: Doctor of Philosophy
Project/Thesis title: A Study on the Cognitive Complexity Metric of a Software

1. The project/thesis is my original work and has not been submitted previously for a degree at this or any other University/Institute. To the best of my knowledge, it does not contain any material published or written by another person, except as acknowledged in the text.

2. I understand what plagiarism is, the various types of plagiarism, how to avoid it, what my resources are, who can help me if I am unsure about a research or plagiarism issue, as well as what the consequences are at University of Colombo School of Computing (UCSC) for plagiarism.

3. I understand that ignorance is not an excuse for plagiarism and that I am responsible for clarifying, asking questions and utilizing all available resources in order to educate myself and prevent myself from plagiarizing.

4. I am also aware of the dangers of using online plagiarism checkers and sites that offer essays for sale. I understand that if I use these resources, I am solely responsible for the consequences of my actions.

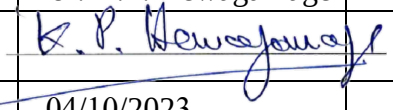
5. I assure that any work I submit with my name on it will reflect my own ideas and effort. I will properly cite all material that is not my own.

6. I understand that there is no acceptable excuse for committing plagiarism and that doing so is a violation of the Student Code of Conduct.

Signature of the Student	**Date (DD/MM/YYYY)**
	04/10/2023

Certified by Supervisor(s)

This is to certify that this project/thesis is based on the work of the above-mentioned student under my/our supervision. The thesis has been prepared according to the format stipulated and is of an acceptable standard.

	Supervisor 1	**Supervisor 2**	**Supervisor 3**
Name	Prof. K.P. Hewagamage		
Signature			
Date	04/10/2023		

****

Plagiarism Policy Compliance Statement

I certify that,

(1) I have read and understood University of Colombo School of Computing Student Plagiarism: Coursework Policy and Procedure

(2) I understand that failure to comply with Student Plagiarism: Coursework Policy and Procedure can lead to academic actions against me

(3) This work is substantially my own, and to the extent that any part of this work is not my own I have indicated that by acknowledging its sources

Name: Dinuka Rukshani Wijendra

Signature: 

Date: 04/10/2023

Abbreviations

IDE – Integrated Development Environment

BCS – Basic Control Structures

LOC – Lines of Codes

BPM – Business Process Modelling

UML- Unified Modeling Language

SDLC – Software Development Life Cycle

POS – Part Of Speech

ER – Entity Relationship

GloVe – Global Vectors for word representation

PNG – Portable Network Graphics

HTTP – Hypertext Transfer Protocol Secure

SPSS – Statistical Package for Social Sciences

Abstract

The cognitive complexity of a software determines the effort required to understand its source code logic. It can be used to indicate understandability and maintainability, which are predominant quality attributes in software development process. Further, personal profile and source code factors can be stated as major factors associated with cognitive complexity. The inclusion of personal profile results cognitive complexity to be a subjective measurement. However, traditional methods of expressing cognitive complexity are limited only to source code factors to express it as an objective measurement. Moreover, a methodology of relating cognitive complexity to indicate understandability and maintainability cannot be observed. As such, this work has studied the mechanisms of applying cognitive complexity in software development and maintenance processes effectively. Accordingly, the procedures of reducing cognitive complexity to improve understandability and maintainability have been introduced. Expression of cognitive complexity by giving more impact in personal profile is a significant achievement of this research work. The usage of software requirements, its logical diagrams, defects tracing, code quality optimization and refactoring have been introduced as cognitive complexity reduction mechanisms. Those mechanisms have been designed using a computational aid. A meaningful cognitive complexity metric has also been introduced to quantitatively indicate cognitive complexity by considering both personal profile and source code factors. The personal factor involvement of the metric has been introduced using a subjective cognitive weight. The components of reducing cognitive complexity have been evaluated with the duration taken to understand a source code. Accordingly, significant duration reduction has been obtained from proposed components comparing to the current practices to process same scenarios. Therefore, the possibility of proposed mechanisms to gain a less comprehension effort and to achieve a less cognitive complexity can be verified. The proposed cognitive complexity metric has been practically and empirically verified through standard software metric frameworks to prove its stability in real applications. Hence, together with the design to attain a lesser cognitive complexity and the metric to quantitatively indicate the subjective user comprehension effort can be used as significant appliances in software engineering.

Keywords: cognitive complexity, cognitive complexity metric, cognitive load, cognitive weight, maintainability, subjectivity, understandability

Table of Contents

List of Tables

List of Figures

List of Algorithms

List of Publications by the Candidate

D. R. Wijendra, K. P. Hewagamage, "Cognitive Complexity Applied to Software Development: An automated Procedure to Reduce the Comprehension Effort", Journal of ICT Research and Applications, vol. 16, No. 3, 2022 DOI: https://doi.org/10.5614/itbj.ict.res.appl.2022.16.3.6

D. R. Wijendra, K. P. Hewagamage, "Cognitive Complexity Beyond Generalization: A Subjective Rating for the Human Comprehension", Fourth International Conference on Advances in Electrical and Computer Technologies 2022 (ICAECT 2022)
DOI: https://doi.org/10.1051/itmconf/20225001003

D. R. Wijendra, K. P. Hewagamage, "*Cognitive Complexity Reduction through Control Flow Graph Generation*", International Conference for Convergence in Technology (I2CT 2022)

DOI: 10.1109/I2CT54291.2022.9824923

D. R. Wijendra, K. P. Hewagamage, "*Application of the Refactoring to the Understandability and the Cognitive Complexity of a Software*", International Conference for Convergence in Technology (I2CT 2022) DOI: 10.1109/I2CT54291.2022.9824082

D. R. Wijendra, K. P. Hewagamage , "*Software Complexity Reduction through the Process Automation in Software Development Life Cycle*", Fourth IEEE International Conference on Electrical, Computer and Communication Technologies (ICECCT 2021)

DOI: 10.1109/ICECCT52121.2021.9616781

D. R. Wijendra, K. P. Hewagamage, "*A Standard Methodology to Assign Cognitive Weights to Compute the Cognitive Complexity of a Software*", International Journal on Recent Trends in Computer Science and Electronics (RTCSE), 2018 ISBN: 978-81-938900-7-3

D. R. Wijendra, K. P. Hewagamage, "*Cognitive Weightage Assignment with Ubiquitous Computing to Compute the Cognitive Complexity*", 11th IEEE International Conference on Ubi-Media Computing, 2018

D. R. Wijendra, K. P. Hewagamage, "*Automated Tool for the Calculation of Cognitive Complexity of Software*", 2nd International Conference on Science in Information Technology (ICSITech), 2016 DOI: 10.1109/ICSITech.2016.7852627

1.0 INTRODUCTION

1.1 Introduction

The cognitive complexity of a software defines the amount of effort required to understand the underlying logic behind its source code [1], [2]. A source code can be considered as a set of instructions which assists to process a certain number of inputs into expected output [3]. Hence, understanding its logic can be defined as the ability to interpret the process followed to convert these inputs to output through set of instructions. Therefore, the concept of cognitive complexity can be applied to the extent that a particular party can understand a source code, so that it can be used as an indicator of source code understandability [4] - [7]. To explain it furthermore, high cognitive complexity implies a high effort utilized on understanding a source code logic, which tends to have less understandability. Similarly, a less cognitive complexity indicates a high understandability due to the less amount of effort applied on understanding the source code logic. Understandability of a source code is an important quality factor inside the software development process because a source code is developed by one person, and it is going to understand and modify by another person in majority of scenarios. Therefore, the ability of accommodating modifications or new changes depends on the ability of understanding a given source code, which can be referred to software maintenance. Software maintenance can be described as the process of applying the modifications to adapt to a another new functionality, correct faults or improve performance or other software attribute [8]. Moreover, a software without maintenance is impractical and uneconomical [9]. Hence, considering the impact of understandability to software maintenance cannot be ignored. Since the emphasis of cognitive complexity is to express the understandability, it can be stated that the cognitive complexity plays a determinant role of software maintenance process as well [10].

The ultimate goal of any software implementation is to have a simple software with a less complexity [11]. A simple software is easy to use and understand, so that the effort utilized to understand its source code should also be less. Hence, a software should be implemented to achieve a lesser cognitive complexity, which in turns to achieve a higher understandability through its source code [12]. The understandability level of each person for a given source code

varies so that their effort applied for its comprehension is different. It is because of the differences associated with personal and source code factors along with logical comprehension process [4]. Hence, the cognitive complexity can be stated as varying from one person to another, which tends to be a subjective measurement [2]. Therefore, quantifying it to represent with a numerical value is a challenging task. Further, it can be believed that the comprehension effort utilized for an individual for a source code becomes lower along with the duration as understandability of the same source code develops gradually. However, there can be certain instances the same source code can be expanded due to the addition of new requirements. Consequently, the understandability of the same source code can become less along with the duration due to the high effort utilized for the comprehension. Therefore, it can be stated that the cognitive complexity of a source code changes along with the duration, which can be defined as a dynamic indicator to impress understandability.

The relationship among cognitive complexity and software complexity should be analyzed as both are used as the measurements to define the complexity of a software in different perspectives. As it has already mentioned, the cognitive complexity refers to the comprehension effort associate with understanding a particular source code [13]. The software complexity refers to the degree of a system or a component has a design or an implementation which is difficult to understand and verify [14]. By analyzing both definitions, it can be stated that they attempt to express the understandability of a software. Currently, several standardized software complexity metrics are used to measure the complexity of software. Each of the complexity metric has been introduced by considering varied software attributes, and it is the rationale behind having a variety of complexity metrics without limiting it to a single complexity metric. Among those, McCabe's cyclomatic complexity is being used widely to derive the software complexity [15]. It considers about the number of linear independent paths of a source code, which can be obtained from the control flow graph generated for its source code. The concept of Basic Control Structures (BCS) controls the number of linear independent paths has been considered through this metric. Hence, it is stated to have high complexity from a high number of independent paths and low complexity from a less number of independent paths in the execution flow of a source code. Then, the aspect of relating the cyclomatic complexity into the comprehension level should be explored. It can be observed that the relationship between the number of liner independent paths and the program comprehension is proportional. Thereby, a source code with high cyclomatic complexity can derive high cognitive complexity, while a source code with less

cyclomatic complexity can result with less cognitive complexity. However, there can be certain practical situations that a source code with high number of execution paths is easy to comprehend, while a source code with a less number of execution paths is difficult to understand. Therefore, deriving a proper relationship of cyclomatic complexity with cognitive complexity is difficult. As another exemplification, Lines Of Code (LOC) metric is also used to evaluate the software complexity [16]. It is believed that a higher LOC tends to reduce the understandability due to the lengthy structure, while a lesser LOC can achieve a higher understandability. Accordingly, a higher LOC source code can result with high cognitive complexity, while a lesser LOC source code outputs a lesser cognitive complexity. However, as it is discussed in cyclomatic complexity, there can be certain situations that a source code with high LOC can have high understandability and a source code with less LOC can have less understandability. Hence, deriving a direct relationship among LOC and cognitive complexity has also become difficult. The reason behind these situations is the objective aspects that the current software complexity metrics are implemented of. Even though software complexity is defined to assess the difficulty level associated with user comprehension, these quantitative and objective metrics have failed to address the actual user comprehension level. Therefore, the term of cognitive complexity should be modelled to evaluate the actual user understandability of a source code, which can be used as a vital indicator inside the software development and maintenance processes as concluded in Figure 1.

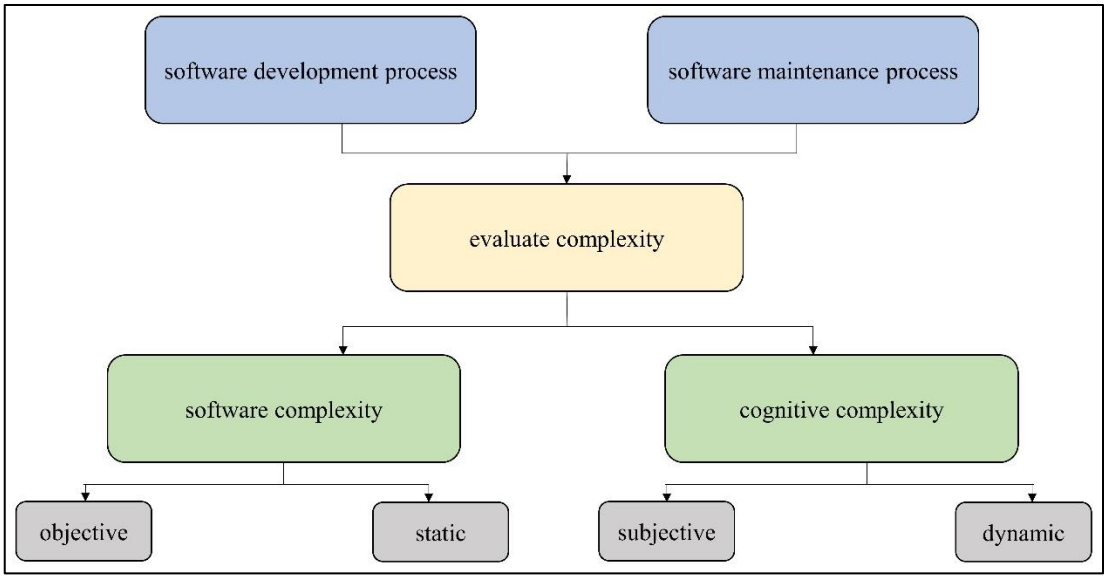


Figure 1. Necessity of Cognitive Complexity inside Software Development and Maintenance Processes

Based on Figure 1, the necessity of applying the concept of cognitive complexity in the field of software engineering can be ensured with respective to its' subjectivity and the possibility of deviating along with the duration.

1.2 Significance of the Study

The cognitive complexity is a direct indicator of expressing the code understandability, while code understandability is a significant and essential factor in code maintenance. Therefore, cognitive complexity can be used to express the maintainability as well. Understandability and maintainability are very important quality attributes which cannot be ignored inside software development process, so that addressing these attributes is a mandatory process to ensure its quality and complexity. Therefore, the significance of applying the cognitive complexity to indicate understandability and maintainability of a source code has been analyzed through this study.

Handling and maintaining a software through its source code is a teamwork inside the software development. Each of its task is performed by a different set of users who have different comprehension capabilities. Moreover, success of the software development process depends on the success of its teamwork, which can be determined through their understandability levels. In other words, the success of teamwork cannot be achieved, if their understandability levels are poor. Consequently, it effects for the progress of software development process. If the comprehension levels of team members have not been considered, unnecessary failures can be happened inside the software development. Therefore, to avoid these failures, cognitive complexity of a software should be addressed properly to increase the understandability. Moreover, the selection process of team members to different phases inside the software development process by aligning with cognitive complexity of software is another alternative to reduce these failures. Since cognitive complexity is varied from each user, it can be used to obtain an opinion about the understandability of each team members in a software project. Hence, the opportunity of categorizing team members according to their comprehension capabilities and assigning them into different stages inside the software development process can also be achieved through cognitive complexity concept to solve unnecessary failures in software development process. As such, cognitive complexity assists to smoothen the development

process, which makes the software team decision process easy. Therefore, the significance of applying cognitive complexity concept inside software project management has also been analyzed with this work.

1.3 Research Gap

The applications of cognitive complexity can be widely observed in a form of metric, which denotes the human comprehension effort for a given source code as a quantitative measurement [17], [18]. Nevertheless, it contradicts with the definition of cognitive complexity as it should be a subjective measurement, which differs from each user. As such, introducing a single quantifiable and objective value to demonstrate the comprehension level of entire user population cannot be accepted as a valid mechanism of expressing the cognitive complexity. This is because the cognitive complexity is modelled by considering only one aspect, which is the source code factor. The amount of information scattered inside the source code referred as the architectural aspect of the source code [1] and the spatial capacity of the source code [19] are main two factors that source code properties have been quantified with. Each research work is based on the sub properties of these two main source code aspects and has expressed with different computations to quantify the cognitive complexity. Consequently, the problem of deriving a single cognitive complexity metric can be signified, and it results with the problem of obtaining a standardized cognitive complexity metric to be used in real software applications. Nevertheless, source code factor is not the only factor that cognitive complexity should be expressed with. The personal profile should also be considered for cognitive complexity determination, which has not been addressed properly. The involvement of personal profile has been indicated by introducing cognitive weights which is a number to represent the comprehension effort for varied code segments which in turns emphasizes the source code aspect [20]. In other words, the proposed cognitive weights denote the source code attributes, which do not highlight the personal profile factors. Basic Control Structures (BCS), variables, functions, recursive components, object-oriented concepts are some of source code segments which have been considered to assign with cognitive weights [2], [21], [22]. These weightage assignment procedures cannot be granted as they are based on experimental outcomes for a selected user group and using several assumptions to quantify the difficulty level of those users. Therefore, these cognitive weights cannot be

validated, and they are limited only for a particular user group which cannot denote entire user population. Even though the recent set of research have been based on describing the influence of cognitive load to the user understandability, a proper mechanism of expressing it with cognitive complexity has not been initiated. The reason is the gap identified between the relationship of user comprehensibility and cognitive complexity which has not been evaluated properly. Moreover, there are certain number of aspects inside the personal profile to consider for the comprehension effort determination. Therefore, confining personal profile only to cognitive weights cannot be accepted as a valid procedure to express cognitive complexity. Therefore, cognitive complexity should be modelled in a proper way, which can emphasize on both source code factor and personal profile.

As it has been already described, understandability is an important quality factor inside the software development process, and it requires to address the maintainability as well [23]. Moreover, the relationship of cognitive complexity with understandability and maintainability can be clearly observed, so that the concept of cognitive complexity should be linked with software development and maintenance processes. Nevertheless, there is no proper guideline which describes the applicability of utilizing cognitive complexity in both of these processes. Therefore, a proper mechanism should be found to observe the relationship of cognitive complexity with understandability and maintainability.

1.4 Novelty of the Research

According to the definition of cognitive complexity, the comprehension effort of an individual for a given source code should be explored to express the complexity of a software. Thereby, consideration of source code aspects should be there as the source code is a main aspect of determining the logic of a software. Even though the source code is available, the comprehension effort depends on the person who deals with it. Therefore, the personal profile should also be considered as another vital factor for cognitive complexity determination. The consideration of different source code aspects can be detected in current cognitive complexity metrics however, the involvement of personal profile cannot be observed in most of the metrics. Even though the human comprehension level has been introduced as cognitive weights, they are consisted with user limitation and validation issues. Therefore, this research work has given more emphasis on

personal profile involvement, as the comprehension effort associated with each individual is the core concept behind cognitive complexity. Along with that, the aspects of the source code have also been followed to express the cognitive complexity. Consequently, a cognitive complexity metric has been proposed by considering both personal profile and source code factors. The inclusion of human comprehension level through cognitive weights has been presented as a predictive factor inside the proposed metric, which mostly analyzes the individual personal profile and the spatial capacity of a source code. Hence, the subjectivity of personal profile has been maintained which solves the problems of cognitive weights used by current cognitive metrics. The architectural aspect has also been included for the metric computation, so that inclusion of source code aspect combined with architectural and spatial aspects can be verified. Thereby, the proposed metric can be stated as a better and meaningful approach to indicate the cognitive complexity than existing cognitive metrics.

The current works of cognitive complexity are limited only to a form of metric due to ease of use and to compare with other complexity metrics. But in this research work, the applicability of cognitive complexity inside the software development has been analyzed thoroughly. The expectation of any software development is to obtain a less complex software while arranging the development and maintenance phases easier to handle by the development team members. In other words, the cognitive complexity of software supposed to be less to make the software handling process easier. Therefore, apart from the metric computation, this work is aimed to analyze the procedures of reducing the cognitive complexity. The reduction of cognitive complexity has been performed by considering the personal profile. The mechanism of handling the human cognitive load to reduce the cognition effort is the basic scenario that has been followed to reduce the cognitive complexity, and that has been implemented in a computational environment to verify the reduction of cognitive complexity.

1.5 Research Problem

A software development is not a single process as it goes with several phases throughout its life cycle [24], and the success of each phases' completion depends on the level of understanding the logic of that software. Similarly, maintenance of the software cannot be performed without understanding its software logic. Moreover, the logic of a software is usually obtained from its source code, as it is the major component which is used for its implementation process.

Therefore, the ability to understand the logic behind the source code plays a vital role to achieve a smooth functioning of software development and maintenance processes. The understandability level can be indicated with the effort taken for the comprehension, as high understandability occurs with less effort utilized for comprehension and less understandability occurs with high effort utilized for comprehension. Therefore, cognitive complexity can be taken as a direct indicator to represent the understandability of a particular software, and it should be capable of handling the maintainability as well. Therefore, under this research work, the possibility of applying the cognitive complexity concept effectively to enhance the software development and maintenance processes has been studied.

1.6 Research Questions

The factors associated with cognitive complexity should be thoroughly analyzed in order to model its applicability in software development and maintenance processes. Since cognitive complexity indicates the comprehension effort for a given source code, the main factors aligned with the source codes' comprehension effort should be studied. Consequently, it can assist with producing different viewpoints of expressing cognitive complexity. Moreover, sub factors belonging to these main factors should be explored to express the cognitive complexity in more meaningful way. Accordingly, the variation of cognitive complexity along with these main and sub factors can be observed in a clear manner.

The expectation of any software development and maintenance processes is to produce a software with a lower complexity which is easier to use and manage. To achieve it, its source code should be implemented with high understandability, as the source code is the leading factor behind these processes. In other words, the source code should be achieved with low cognitive complexity to attain high understandability. Therefore, the procedures of deriving a less cognitive complexity should be analyzed along with the factors related with cognitive complexity to smoothen the software development and maintenance processes.

The cognitive complexity is quantitatively indicated using a cognitive complexity metric. It further assists to use this concept in the complexity determination as a quantifiable value is easy use and compare with other existing complexity metrics. Therefore, a procedure of introducing a

meaningful cognitive complexity metrics should be analyzed, which in terms of demonstrating the factors associated with it and to facilitate the software development process.

To conclude, three research questions formed to assist the main research problem mentioned in section 1.5 have been listed as follows.

Research Question 1: What are the main factors that cognitive complexity can be modelled?

Research Question 2: What are the procedures to be followed to reduce the cognitive complexity?

Research Question 3: How to introduce a meaningful cognitive complexity metric to facilitate the software developing process?

1.7 Research Objectives

The following research objectives have been formed to achieve the research questions mentioned in section 1.6.

Research Objective 1: Identify the factors of expressing the cognitive complexity

Research objective 1 is aligned with research question 1.

Research Objective 2: Identify the procedures of reducing the cognitive complexity

Research objective 2 is aligned with research question 2.

Research Objective 3: Design a methodology to demonstrate the procedures of reducing the cognitive complexity

Research objective 3 is aligned with research questions 1 and 2.

Research Objective 4: Propose a meaningful cognitive complexity metric

Research objective 4 is aligned with research questions 1 and 3.

Research Objective 5: Evaluate the design and the cognitive complexity metric

Research objective 5 is aligned with research questions 2 and 3.

1.8 Resource Requirements

The computation design has been implemented by using NetBeans Integrated Development Environment (IDE) version 8.2. Along with the IDE, Visual Studio Code, Jupyter Notebook with Python, PlantUML, EasyUML, FindBugs bug tracker, FlaskAPI, Javafx and JavaSwing, ReactJS and NodeJS should be installed to execute main components of the design. Furthermore, Multer, Putout, Axios libraries should be installed in ReactJS and NodeJS. Under machine learning techniques, Numpy and Pandas Python packages, pickle tool and matplotlib and Seaborn libraries have bene used.

1.9 Structure of the Thesis

The first chapter of the thesis indicates the introduction of research work, which includes the significance, research gap, novelty, research problem, research questions and research objectives. The literature review has been described under second chapter. The third chapter outlines the methodology followed to address research objectives which can solve research questions mentioned earlier. The detailed discussion of results gained through the implemented system and proposed metric has been included in fourth chapter. Finally, conclusions have been described under fifth chapter along with the scope and the contribution of the research work.

1.10 Summary

The cognitive complexity of a software determines the amount of effort required to understand the underlying logic behind a source code of a software. The cognitive complexity can be denoted as a direct indicator of user understandability as the understandability of a user depends on the comprehension effort. Further, understandability is an important factor to express the maintainability so that cognitive complexity can be used to express both of these quality attributes. However, the human comprehension effort varies with each user, since the ability of handling a source code varies among them. Hence, the cognitive complexity is a subjective measurement, which is difficult to quantify. The definitions of both software complexity and cognitive complexity assess the difficulty level associated with comprehension, so that there has

to be a direct relationship with each other. However, it is difficult to observe a proper relationship among them as current software complexity metrics assess the complexity only in terms of quantifiable source code aspects, which confine the complexity expression into an objective measurement. Moreover, current findings of cognitive complexity have been only limited with source code consideration, which do not exactly impress the idea behind cognitive complexity. In addition to that, reluctance of a proper mechanism of relating the cognitive complexity concept with understandability and maintainability should be emphasized. Accordingly, the necessity of deriving a proper mechanism to express the cognitive complexity with the aid of these quality attributes has been arrived. Hence, this work attempts to fill the gap of associating cognitive complexity effectively with understandability and maintainability. Accordingly, the problem of applying the cognitive complexity inside the software development and maintenance processes is expected to be addressed inside this research work. It includes analyzing the factors effecting for cognitive complexity, exploring the procedures of handling and reducing cognitive complexity inside software development and maintenance processes through a computational aid, proposing a meaningful cognition metric with personal and source code aspects and verifying both design and the metric for real usage. The ability of deriving an opinion about the understandability and the impact of cognitive complexity with software project management are the significances of conducting this work. The inclusion of overall expression of cognitive complexity as a combination of both personal profile and source code factors and analyzing its applicability throughout the software development process without limiting it for a metric computation are the originalities of this work.

2.0 LITERATURE REVIEW

2.1 Introduction

The cognitive complexity of a given source code plays a determinant role of expressing its understandability. There have been numerous research works performed to find a suitable way of presenting it, so that the understandability of a source code can be analyzed through it [14]. According to literature, the applicability of cognitive complexity has been determined in form of quantitative and qualitative expressions as shown in Figure 2.

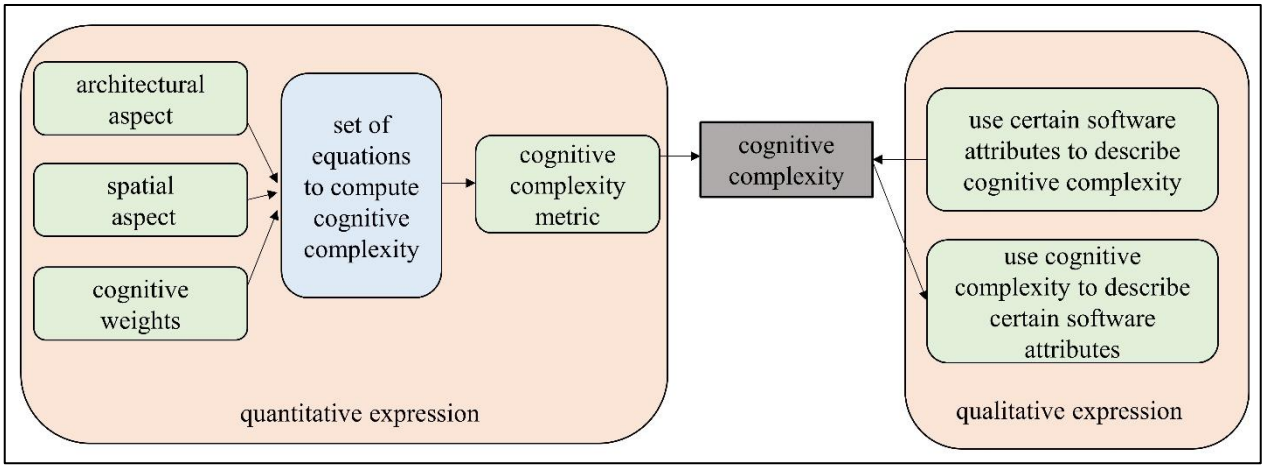


Figure 2. Categorization of Cognitive Complexity in Literature

The quantitative representation of cognitive complexity is regarded as cognitive complexity metric, which is an indicator to express the human comprehension effort through a numerical measurement. The cognitive complexity is a non-quantifiable indicator as it implies a subjective measurement. Therefore, to form a metric, a certain set of factors have been selected and presented it quantifiably. Therefore, cognitive complexity metric artificially quantifies the human comprehension effort which is not exactly equivalent to actual cognitive complexity. The usage of cognitive complexity metric should be accepted due to ease of use and compare with other complexity metrics. Each traditional cognition metric computation is comprised with quantifiable source code aspects and the concept of cognitive weight which addresses the human comprehension level through a numerical value [21]. The quantifiable source code aspect has been derived with respect to the amount of information inside the source code which refers to the architectural aspect of the source code [1] and the spatial capacity of a source code [19]. It is evident that the amount of information inside a source code is proportional to the amount of

effort acquired for its logical comprehension, and the size of a software also a factor of determining the status of the understandability. The architectural aspect has been indicated by considering numerous information categories inside the source code, and the spatial capacity has been attained in terms of LOC. The cognitive weights have been proposed based on a selected user group and by using assumptions to derive their comprehension levels. Based on these categories, a set of equations have been formed to compute the cognitive complexity quantitatively. However, variation of categories of the architectural aspect and the cognitive weights results different computations for the cognitive complexity, which computes different complexity values for the same source code.

Apart from the cognitive metric, the representation of cognitive complexity as qualitative measurements can also be observed. In one direction, cognitive complexity is described using a set of selected software attributes. On the other hand, usage of cognitive complexity as a predictive factor to determine another set of software attributes can be observed. Hence, the traditional attempts to express the concept of cognitive complexity inside the software development can be preliminary categorized into quantitative and qualitative representations.

2.2 Earlier Developments to Measure the Cognitive Complexity

Majority of earlier works regarding the cognitive complexity are based on the quantification of cognitive complexity as a metric. The cognitive complexity metric has been introduced with the concept of *Cognitive Functional Size (CFS)* of a given software by Shao and Wang [25]. It has been stated that the functionality of the source code can determine the level of understandability so that the content of each function has been considered for the human cognition. The functional size of the source code was found to be depended on its input, output and its internal flow and expressed using Equation (1). The amount of input output parameters has indicated the comprehended capacity by an individual as there is a considerable amount of information covered through high number of parameters comparing to a lesser number of parameters. Further, the internal flow has been presented through cognitive weights assigned for BCS. According to Equation (1), *CFS* is computed where N_i is the number of program inputs, N_o is the number of program outputs and W_c is the sum of cognitive weights assigned for each BCS.

$$CFS = (Ni + No) * Wc \quad (1)$$

The cognitive weights assigned for BCS are listed in Table 1.

Table 1. Cognitive Weights Defined for *CFS* Calculation [25]

Category	*BCS*	*Wi (cognitive weight)*
Sequence	Sequence	1
Branch	If – Then – Else	2
	Case	3
Iteration	For – do	3
	Repeat – until	3
	While – do	3
Embedded Component	Function Call	2
	Recursion	3
Concurrency	Parallel	4
	Interrupt	4

According to Table 1, the cognitive weightages assignment has been increased from sequence to concurrent categories inside a given source code. It indicates that a source code with sequential logic is easy to comprehend than a source code with other controlling categories. This can be verified as the least comprehension effort is required for a code segment without any controllers. The cognitive weight of branch has been considered as the next category of comprehensibility. *If-else* conditions shows high understandability over *switch-case* statements. Generally, most users prefer *if-else* statements to handle the conditional checking than *switch-case* statements so that maintaining *if-else* statements inside a source code would lessen the comprehension effort. Accordingly, the weightage assignment of *if-else* statements should be lower than *switch-case* statements. Then, the looping category has been evaluated to observe more comprehension effort than branch category. It can be mentioned that looping criteria contains number of executions occurred within defined number of times, which cannot not be observed in branch category. Thereby, it can be stated that the looping category acquires a considerable amount of effort than branch category. Although *for*, *repeat* and *while* loops are occupied with this looping category, the same weightage (3) has been allocated for all types of loops. It is practically apparent that

users' cognition effort for all the types of loops are not same due to their preference and the analytical skill of each looping criteria. Hence, the same cognitive weight allocation for all three types of loops cannot be granted in this work. Moreover, the procedure of allocating same weight (3) for *switch-case* in branch category and for all types of loops is questionable and unacceptable as a proper reason for that assignment is not mentioned. As the next component, the embedded components have been considered. In there, non-recursive and recursive functions have been assigned with different weightages. Non-recursive functions have been considered to be more comprehensive than recursive functions. The reason for that assignment can be stated as the ability to identify the logic of a non-recursive function than a recursive function, since recursive function executes the same function repeatedly until the base condition occurs. Hence, unless the user has analytical skill of recursion, it is difficult to comprehend the logic behind a recursive component. Therefore, higher weightage allocated for recursive component can be granted. At last, the concurrent components have been selected as the least comprehensive components inside a source code, which requires higher cognition effort. The parallel and concurrent executions have been utilized under this category, and the same weightage (4) has been assigned for those categories. It is unacceptable to generalize the same cognition effort for both of these categories and a valid reason for this weightage allocation has not been highlighted. Therefore, the cognitive weight allocation and the cognitive complexity computation of this research work tends to be ambiguous and cannot be approved for the usage of real applications to denote the user understandability.

Kushwaha and Misra have proposed the dependency of cognitive complexity through the amount of information inside a given software, which information can be represented as a function of identifiers and operators [26]. As a result, *Cognitive Information Complexity Measure (CICM)* was proposed using the product of *Weighted Information Count of Software (WICS)* and the cognitive weight (W_c) of BCS in software as in Equation (2).

$$CICM = WICS * Wc \tag{2}$$

In this metric, *WICS* is the sum of the weighted information count (*WICL*) of every LOC of a given software, which can be denoted in Equation (2.1).

$$WICS = \sum_{k=1}^{LOCS} WICLk \tag{2.1}$$

Furthermore, *WICL* is a function of the identifiers and the operands per line of code, which can be computed as in Equation (2.2). *LOCs* is the number of lines in the source code and ICS_k is the information contained in a software program for the k^{th} line.

$$WICLk = \frac{ICSk}{[LOCs-k]} \tag{2.2}$$

In here, they have found the relativity of human cognition according to the architectural aspect of the source code. It has been calculated by identifying the identifiers and operators in each LOC. That has been calculated as a division of the number of identifiers and operators per LOC with the subtraction received with the remaining LOC as it can be observed in Equation (2.2). In other words, when the LOC count is gradually increasing, the remaining LOC count becomes decreasing, thereby the cognitive effort becomes higher than the earlier LOC. This situation implies the possibility of applying the spatial capacity of a source code, since the user has to remind a considerable amount of information along with incremented LOC. Herein, this can be considered as a vital achievement of demonstrating the cognitive complexity mainly through the architectural aspect and the spatial aspect. Further, the cognitive weights of BCS have been considered for this computation due to the control flow maintenance accomplished by them. However, these weightages were obtained through Table 1, and the drawbacks mentioned earlier are applied in this computation as well. Thereby, this computation cannot be accepted as a valid methodology of expressing the human comprehension.

Another concept of human cognition metric namely, *Modified Cognitive Complexity Measure (MCCM)* is proposed by considering all the operators and the operands in a given software by Misra [27]. The equation proposed to determine the cognitive complexity has used N_{i1} for total number of operators, N_{i2} for aggregate number of operands and W_c is the summation of cognitive weights assigned for BCS. With those parameters, *MCCM* was computed through Equation (3).

$$MCCM = (N\,i1 + N\,i2) * Wc \tag{3}$$

The same concept followed in [26] can be clearly observed in [27], such that both works are based on the architectural aspects inside the source code with respect to the number of operators and operands. But, in [27], the consideration of spatial capacity cannot be identified, which can be stated as a drawback. Moreover, the same weightage assignment for BCS mentioned in Table 1 has been performed continuously, which results the complexity computation into a non-standardized state.

The same work has been extended with the concept by introducing the effectiveness of cognitive complexity by total number of occurrences of input and output variables [28]. Therefore, the metric has been re-introduced as *Cognitive Program Complexity Measure (CPCM)*, which can be observed in Equation (4) as the addition of input output parameters (S_{IO}) of the program and the cognitive weights assigned for BCS (W_c).

$$CPCM = SIO + Wc \tag{4}$$

This highlights the significance of considering the input output parameters for the user comprehension as in [25]. Along with that, the usage of cognitive weights for the BCS based on Table 1 can be seen, which has not been derived through a valid methodology.

The introduction of *New Cognitive Complexity of Program (NCCoP)* by Jakhar and Rajnish has been proposed with respect to the data objects consisting with input and output parameters, internal behavior of the software, operands and the cognitive weight of BCS of each line of code [1]. Hence, the computation is done through the number of variables in a particular line of code (N_v) and the cognitive weights assigned for of BCS (W_c) as in Equation (5).

$$NCCoP = \sum_{k=1}^{LOCS} \sum_{v=1}^{LOCS} Nv * Wc(k) \tag{5}$$

In this approach, the consideration of the architectural aspect has been performed through the number of variables in each LOC. The human comprehension level is denoted with the cognitive weights in Table 1. The invalid procedure of obtaining the cognitive weights through Table 1 prompts this research work to be impractical.

A new approach of evaluating the cognitive complexity was introduced by Chhabra with the concept of spatial complexity of each function in the module and the input output parameters of the module [19]. It is defined as the *Code Cognitive Complexity (CCC)*. The spatial complexity has been calculated as the distance in terms of LOC between the module call to its definition since the greater distance from module definition to its usage requires more comprehension effort to understand its internal logic due to the higher distance of navigating inside the source code. Similarly, a lesser distance between a module call to its actual implementation tends the user to easily comprehend its logic as the navigation becomes lesser. So that, *CCC* has been defined as a combination of the cognitive weight of control statement which the module call has been made (W_c), count of all module calls in the software (m), the spatial distance of the module call from its

definition (*Distance*), number of input output parameters (N_{ip} + N_{op}), cognitive weight of each parameter p_i (W_{pi}) of j^{th} call in the module (MC_j). The proposed equation can be observed in Equation (6).

$$CCC = \sum_{j=1}^{m}(Wc * Distance(MCj)) + \sum_{i=1}^{Nip+Nop} Wpi(MCj) \quad (6)$$

The cognitive weights used for above computation have been demonstrated in Table 2.

Table 2. Cognitive Weights Defined for *CCC* Calculation [19]

Category	*BCS*	*Cognitive Weight*
Sequence	Sequence	1
Branch	If – Then – Else	2
	Case	3
Iteration	For – do, while, while – do	3
	Nested control statements	4
Constant data	Constant values	1
	Enumerations & defined constants	1
Variables	Atomic & elementary	1
	Array (1D) & structure	2
	Multi-dimensional array & pointer-based indirection (single)	3
	Multiple indirections, pointer to structure, etc.	4

In [19], the usage of both architectural and spatial aspects of a source code can be viewed in a wider context. The architectural aspect has not been limited to the number of input output parameters, operators and operands, but with the concept of module calls as well. Modules are implemented to perform a certain number of activities inside the source code, so that the consideration of them to analyze the comprehension effort is essential. In here, the complexity has been derived as the addition of the complexities computed through the spatial and the architectural aspects. In the first component, the spatial complexity has been determined through

the cognitive weight of the control statement which has performed the module call and the spatial distance from its call to its implementation. Secondly, the architectural complexity has been computed through the cognitive weight determination of input output parameters in each module. Moreover, cognitive weights listed in Table 2 have been considered for the computation process.

As in Table 1, the sequence statements have been considered with least cognitive weight (1) which implies the simplicity among other source code categories. As in Table 1, *if-else* has been considered as more complex than sequence statements, while *switch-case* statements are complex than *if-else* statements. It can be accepted as the preference of *if-else* is higher than *switch-case* statements in current practical scenarios, so that *if-else* should be assigned with a lesser cognitive weight. Along with that, same weightage (3) allocation can be observed in for and while loops, which cannot be granted as in Table 1 due to the user preference for one looping criteria. Additionally, the introduction of nested looping can be emphasized with a higher complexity. Generally, the understandability level of nested control structures is much higher than the single controllers so that assigning a higher weight for the nested loops can be accepted. However, all the nested loops cannot be granted as the same weightage as the comprehension level deviated with the preference and the analytical skill of each individual. Hence, a variation of nested loops should be addressed in this work.

Later, the implementation of the coding has been moved with the concept of object-orientation apart from function-oriented coding. Then, the necessity of computing the cognitive complexity for object-oriented source codes has initiated. Consequently, Misra et al. proposed that the cognitive complexity of an object-oriented code can be represented using the corresponding cognitive weights of each method of the class [29]. It was calculated using the cognitive weights assigned for individual BCS in q linear blocks as listed in Table 1, which may consist of m layers of BCS with n BCS in each layer as in Equation (7). Furthermore, each recursive call has been considered as another call within the same method.

$$MC = \sum_{j=1}^{q}[\prod_{k=1}^{m}\sum_{i=1}^{n} Wc(j,k,i)] \quad (7)$$

This achievement cannot be considered as a new approach of computing the human cognition due the BCS consideration, which has been repeated in previous computations as well. Even though the source code has been divided into q number of linear blocks with m number of layers, and with n number of BCS in each layer, it merely considers only the BCS appearance inside the

source code. Moreover, the consideration of cognitive weights in Table 1 highlights the drawbacks appearing repeatedly which have to be handled.

A methodology of computing the *Software Metric for Python* (SMPy) was introduced by Misra and Cafer, which can be applicable for any other object-oriented programming language [30]. According to that study, they have found that the cognitive complexity of an object-oriented software depends on the *Inheritance Complexity* (*CI class*), *Complexity of Distant classes* (*CD class*), *Global Complexity* (*C global*) and the *Complexity due to Coupling* (*C coupling*) and computed the complexity as in Equation (7.1).

$$SMPy = CI\ class + CD\ class + C\ global + C\ coupling \tag{7.1}$$

The concepts taken for this computation can be regarded as a vital achievement as these concepts have not been considered before. Noteworthy, they have estimated the complexity of a simple class before computing the values for *CI class* and *CD class* as it becomes a part of inheritance of distinct class complexities. Therefore, the complexity of a simple class (*C Class*) has been computed as in Equation (7.2).

$$\begin{aligned} Cclass\ =\ & weight\ (attributes) + weight\ (variables) + weight\ (structures) + \\ & weight\ (objects) - weight\ (cohesion) \end{aligned} \tag{7.2}$$

Here, *weight(attributes)* has been determined through *Arbitrary Named Distinct attributes/ variables* (*AND*) and *Meaningfully Named Distinct attributes/ variables* (*MND*) as shown in Equation (7.3).

$$W\ (attributes/varaibles)\ =\ 4 * AND\ +\ MND \tag{7.3}$$

They have found that the arbitrary and meaningfully named attributes and variables are one of the major rationales behind the source code complexity. Further, if a program consists with arbitrary named attributes and variables, its comprehensibility becomes lower, which will increase the cognitive complexity. It has been mentioned that the difficulty level of understanding the arbitrary named attributes and variables is four times more than the meaningful naming [31], which creates the path for the multiplication of AND by 4 to derive the attributes weightage. Similarly, the *W(variables)* has been computed through Equation (7.3). Then, *W(structures)* has

been computed through the weights of BCS inside the source code, and those weights have been listed in Table 3.

Table 3. Cognitive Weights Defined under [30]

Category	*Cognitive Value*
Sequence	1
Condition	2
Loop	3
Nested Loop	3
Function	2
Recursion	3
Exception	2

According to Table 3, a source code without any control structure has been assigned with weight 1, which implies the least human cognition effort. Then, the conditional statements have been assigned with weight 2 as they make the program behavior dynamic and created a combination of sequences built up in different possible situations. Hence, that weight has to be more than the weight allocated for sequence. Loops have been considered as the next level of difficulty associated with cognition due to the same repetition. Although nested loops are more complex than single looping, the same weightage allocation (3) can be observed, which cannot be accepted as a valid scenario. However, the same weightage allocation for all categories of conditional statements, loops and nested loops as 2, 3 and 3 can be observed respectively. Since the preference and the comprehension level of each individual over the multiple controllers within the same controlling category are varying, all categories cannot be assigned with a same value as described earlier as well. Functions increase the efficiency of a source code, although function calls disturb readability of a source code, so that the cognitive complexity of functions should be higher than the sequence. That might be the reason behind allocating the weightage as 2 for functions. The concept of recursion is much complex than non-recursive functionalities, so that weight of 3 has been allocated for recursion. Finally, exceptions have been assigned with weight 2, but the reason behind its allocation has not been mentioned. Surprisingly, the reason behind gaining the same cognitive effort for conditional statements, functions and exceptions has not been mentioned. Therefore, this weightage allocation also cannot be accepted as a validated procedure of expressing the human cognition. Then, the next parameter inside Equation (7.2)

weight(objects) has been always allocated for 2. It has been revealed that the creation of the object is the calling of its constructor, which works like a function. Thereby, the weight of an object has been assigned with the same weight of a function. Nevertheless, functions and constructors have differences, and the comprehension levels of both categories can be different. From that point, that difference should be elaborated in the weightage allocation, which cannot be observed in this achievement. Then, *weight(cohesion)* in Equation (7.2) has been computed as in Equation (7.4) with number of *Methods with Attribute usage* (*MA*) and number of *Attributes inside the Method* (*AM*).

$$W\ (cohesion) = MA/AM \tag{7.4}$$

The last variable of Equation (7.2) indicates the weight of cohesion. Cohesion implies the interaction within a module, and it is expected to have a higher cohesion within a module implementation [32]. If the internal segments are tightly interconnected within a module, the internal logic of that module can be obtained without referring to the other external modules. Hence, a higher cohesion implies a lesser cognitive complexity. This situation might be the rationale behind the weight of cohesion being subtracted to calculate the complexity of a class. After these estimations, they have proposed the calculation process each parameter in Equation (7.1). Firstly, the *Complexity due to Inheritance* (*CI class*) has been obtained by considering the Estimated Complexities (*Cclass*) of the classes in each level. Thereby, *CI class* of an object-oriented source code with *m* number of levels in which its j^{th} level has *n* number of classes is obtained by getting the summation of *Cclass* of all *n* classes, which is then multiplied by the number of levels as shown in Equation (7.5).

$$CI\ class = \prod_{j=1}^{m}[\sum_{k=1}^{n} Cclass\ jk] \tag{7.5}$$

This consideration can be stated as valid as the concept of inheritance is applied to a number of levels, so that the complexities of each class in each layer has to be computed first. As the same behavior applies with the inheritance, it has to be multiplied by the complexities of the total number of layers. Along with that, *Distant class complexity* (*CD class*) can be obtained as the summation of each class estimated complexities. The *Complexity of Global variables* (*C global*) has been determined as the summation of the weight of variables, structures and objects, where the weight of variables has to be computed according to Equation (7.3). At last, *coupling Complexity* (*C coupling*) has been denoted in Equation (7.6), where *c* is the number of connections.

$$C\ coupling = 2\hat{}c \tag{7.6}$$

As coupling creates the method calling of another class, it behaves like a method calling, which its cognitive weight is 2. Further, these connections should be implemented in between 2 entities, so that the base has been taken as 2. When the number of connections is increased, there would be a significant increment of the comprehension effort, so that the number of connections (*c*) has been taken as the concept of power. Accordingly, summation of the complexities derived through inheritance, distinct classes, global variables and coupling has been determined as the cognitive effort of the source code. Although the concepts inside this work are unique and methodological, the problems regarding with cognitive weights in Table 3 have to be verified.

As another approach, the object-oriented cognitive complexity by Misra et al. was further proposed through a suite of five metrices namely *Attribute Complexity* (*AC*), *Method Complexity* (*MC*), *Class Complexity* (*CLC*), Message Complexity also referred to as *Coupling Weight for a Class* (*CWC*) and *Code Complexity* (*CC*) [20]. The overall complexity is denoted with *CC*, which has been derived through other four metrics. Thereby, *CC* is proposed as shown in Equation (8).

$$CC = \prod_{j=1}^{m}[\sum_{k=1}^{n} WCC\ jk] \tag{8}$$

Firstly, they have computed the *MC* through the BCS in each method according to the values introduced in Table 3. They have further illustrated the whole source code as a combination of *q* linear blocks with *m* layers in each block and with *n* linear BCS in each layer. Therefore, the cognitive weight of all methods has been computed as the summation weight of *n* linear BCS expanded within *m* layers, which is computed through other *q* linear blocks as presented in Equation (8.1).

$$MC = \sum_{j=1}^{q}[\prod_{k=1}^{m}\sum_{i=1}^{n} Wc\ (j,k,i)] \tag{8.1}$$

Through this equation, they have highlighted the behavior of BCS as they control the method execution. Nevertheless, considering only BCS is not sufficient for the architectural aspect of the source code. Consequently, the connectivity of each method inside the source code has been considered. The comprehension effort should be increased with the increment of the message callings. Therefore, it has been considered as another factor of expressing cognitive complexity and proposed as *Message Complexity/Coupling Weight for a Class* (*CWC*) through the summation of weights of calls and weight of called methods as represented in Equation (8.2).

$$CWC = \sum_{i=1}^{n}(2 + MC\ wi) \quad (8.2)$$

According to Equation (8.2), MC_{wi} represents the method complexity of the called method, which can be computed according to Equation (8.1). The weight of 2 is introduced to represent the weight of the message sent for the external method. But, the reason of obtaining that weight as 2 and allocating the same weight for any type of external messages creates a problematic scenario. Further, if there are *n* number of external calls inside the source code, the total message complexity has been determined as the summation of the weights of all message calls. Apart from the BCS and message callings, significance of the attributes inside the methods has been deemed for the human cognition. As repeated in majority of previous research works, the appearance of attributes in terms of inputs, outputs, operators and operands has been observed. Moreover, it is feasible to gain more comprehension effort on more attributes than a smaller number of attributes. Therefore, the *AC* should be computed within the cognitive complexity and its weightage has been assigned to 1. Due to the locality of an attribute to all the objects and the accessibility of it by the other procedures, it has been always assigned to 1. Then, the complexity of a class namely *Weighted Class Complexity* (*WCC*) is computed through the summation of *AC* and *MC* as displayed in Equation (8.3).

$$WCC = AC + \sum_{p=1}^{n} MC\ p \quad (8.3)$$

According to Equation (8.3), the class complexity can be stated as the complexity derived from its attributes and the BCS inside the methods. However, the connectivity of the methods among message callings, which has been calculated in Equation (8.2) is not considered for the class complexity determination. Hence, it can be declared as a drawback of above computation. Finally, the complexity of whole system has been proposed by considering the all the classes and their relationships. As the relationship, they mainly focused about inheritance. The summation of the class complexities in one layer has been computed first and then obtained the complexities according to the level of source code depth through the production. Herein, a source code with *m* levels of depth and j^{th} level has *n* classes, *CC* is given by Equation (8.4).

$$CC = \prod_{j=1}^{m}[\sum_{k=1}^{n} WCC\ jk] \quad (8.4)$$

As *WCC* is comprised with attributes and method complexities, the same problem of message complexity non-consideration has been repeated again. Along with that, the concept of coupling

is ignored in this computation. Moreover, the possibility of addressing the cohesion is not mentioned. Therefore, this computation is not justifiable due to the drawbacks mentioned earlier.

A survey to investigate the applicability of the cognitive complexity for object-oriented programming was carried out by Aloysius and Arockiam [33]. They have classified the cognitive complexity object-oriented metrices as *Class Complexity* (*CC*), *Weighted Class Complexity* (*WCC*) and *Extended Weighted Class Complexity* (*EWCC*). Noteworthy, the computation *WCC* and *CC* are as same as Equation (8.3) and (8.4) respectively. It is significant that they have discussed the advantages and disadvantages of *WCC* and *CC* as well. It is mentioned that the unavailability of applying the object-oriented concepts such as inheritance, encapsulation, polymorphism and overloading in *WCC* computation, and internal structure of methods in *CC* computation. Although *CC* computation is involved with *MC* with the availability of BCS, the impossibility of handling the internal structure inside methods arises with a complication. As a new concept, they have extended the concept of *WCC* as *Extended WCC* (*EWCC*) with the concept of inheritance, since it has been mentioned as a drawback. Therefore, *EWCC* has been introduced as in Equation (8.5).

$$EWCC = AC + \sum_{p=1}^{n} MC\ p + \sum_{j=1}^{m} ICC\ j \tag{8.5}$$

ICC implies the *Complexity due to Inheritance*, which can be computed as in Equation (8.6).

$$ICC = (DIT * CL) * \sum_{k=1}^{s} RMCk + RN \tag{8.6}$$

The concept of method and attribute usage has been introduced with respect to the inheritance under this work. Hence, *RMC* is defined to gain *Complexity of Reused Methods* through *MC* as in Equation (8.1). *RN* indicates the number of reused attributes and *DIT* is the *Depth of Inheritance Tree*. *CL* stands for the cognitive load of L^{th} level, which has been introduced to solve the internal structure of the methods. Although the cognitive level is varying on each individual, they have assumed that *CL* level is always 1, which is a major drawback of creating the objectivity over the subjectivity. Moreover, they claim to have a success on *EWCC* over *WCC* with regard to the inheritance and internal architecture through cognitive load, the assumption has led to its lack of success. Remarkably, they have observed that cognitive complexity is an imperative field in terms of software complexity calculation. However, they have concluded that most of the object-oriented cognitive complexity metrices have not been evaluated or validated under available software complexity metrics validation frameworks and, also do not encounter the features such

as polymorphism, cohesion and coupling in order to complete the calculation process. As an extension and a solution for these, they have expressed that the cognitive complexity of an object-oriented code can be derived through the concept of coupling. The usage of coupling indicates the connectivity between other external modules, and it is recommended to have the coupling in a lower state. This can be verified as a higher number of external connections between modules can increase the difficulty of understanding, so that its cognitive complexity becomes high. Subsequently, they have introduced *Cognitive Weighted Coupling Between Object* (*CWCBO*) [34] as in Equation (9).

$$CWCBO = ((CC * WFCC) + (GDC * WFGDC) + (IDC * WFIDC) + (DC * WFDC) + (LCC * WFLCC) \quad (9)$$

CWCBO includes various types of coupling as listed in Table 4.

Table 4. Types of Coupling Introduced under [34]

Coupling Type	*Description*
Control Coupling (CC)	One module controls the sequence flow of the other module
Global Data coupling (GDC)	Two or more modules share the same global data structures
Internal Data Coupling (IDC)	One module directly modifies the local data of another module
Data Coupling (DC)	Output of one module is the input to another module. Parameter lists are used for the data passing
Lexical Content Coupling (LCC)	A part or all the contents of one module is included in another module

According to Equation (9), each type of coupling in Table 4 has been multiplied with a corresponding rate. That rating has been defined as the corresponding cognitive weight for each type of coupling. The result obtained from the multiplication of one category in Equation (9), prompts the understandability effort of that coupling type. A detailed analysis has been performed to obtain these cognitive weights by conducting a comprehensive test for a selected number of bachelors and master's students. Students who have scored 65% or more for another general examination have been selected as the targeted users, and 30 students were selected through that scenario. All the users were given 2 object-oriented source codes regarding each coupling type and measured the average time taken to understand both of them. The duration

taken to understand a given program indicates its comprehension effort. In other words, higher duration taken to understand a source code implies a higher complexity, while a simple program can be understood within a shorter time period. Accordingly, the mean time taken for each coupling category and the proposed cognitive weights are listed in Table 5.

Table 5. Proposed Cognitive Weights for Coupling Category [34]

Program	*Comprehension Time*	*Coupling Category*	*Mean Comprehension Time*	*Weighting Factor (WF)*
1	40.7	LCC	40.18333	WFLCC = 4
2	39.66667			
3	30.76667	DC	30.88333	WFDC = 3
4	31			
5	21.43333	IDC	22.21667	WFIDC = 2
6	23			
7	10.8	GDC	11.13333	WFGDC = 1
8	11.46667			
9	10.16667	CC	10.11667	WFCC = 1
10	10.06667			

According to Table 5, the weighting factor of each coupling type has been obtained by comparing each of mean comprehension time with minimum mean comprehension time. In other words, the least average duration tends to be about 10, and its weightage has been taken as 1. As mean duration of *IDC* is 2 times larger than the minimum mean, its weight has been considered as 2. Similarly, other weightages have also been obtained under the same criteria. Since the mean duration of *GDC* is closer to the minimum mean, its weight has also been proposed as 1. Although these cognitive weights have been proposed through a valid methodology, these weights are specified only to that user group, which cannot be generalized to entire users. Accordingly, applying this metric as a standardized metric have become problematic.

A methodology of assigning cognitive weights for BCS for cognitive complexity for java-based object-oriented source codes has been proposed by Crasso et al. [22]. Preliminary, variety of BCS have been limited to sequence, branch (*if then else*, *case*), iteration (*for-do*, *while-do*, *repeat-until*), embedded component (function call, recursion) and concurrency (parallel, interrupt), and

they have used the same cognitive weights in Table 1 for this purpose. As their new approach, they have extended the cognitive weightages assignment within embedded component such as call to a local (including super) method, call to a non-local method, call to a method of an external library and call to an abstract method which can be overridden. Sequence category has been extended to *try-finally* without *catch* block and *catch* block has been included to the branch category. The weights allocated to their approach have been shown in Table 6.

Table 6. Cognitive Weights Defined under [22]

Category	*BCS*	*Cognitive Weight*
Embedded Component	Call to a local (including super) method	2
	Call to a non-local method *m*	$2 + W_m$
	Call to a method *m* of an external library	3
	Call to an abstract method *a*, for which $a_0, \ldots\ldots, a_n$ override *a*	$I + f(W_{ax})$
Sequence	Try-finally (no catch block)	2
Branch	catch*[catch]	3

As in Table 1, the cognitive weight for a function call has been determined as 2. It also includes the methods in the calling class or its parent class. The weight of call to a non-local method is computed as the weight of the function calling itself (2), and the weight calculated for the non-local methods according to Table 1. A method calling to an external library has been introduced as 3, due to the functional call weight (2) and 1 weight added to understand the method signature which is being called. The computation of W_m is not applicable in this scenario as the external libraries are closed. The cognitive weight allocated for an abstract method is defined in terms of overridden methods. They have mentioned that it can be obtained either by computing the sum of all overriding methods ($SUM(W_{ax})$), their average ($AVG(W_{ax})$), maximum weight ($MAX(W_{ax})$) or the minimum weight ($MIN(W_{ax})$). Noteworthy, the software engineer should select the option with regard to abstract methods. In that way, the possibility of making a wrong option can be stated, so that it would create a wrong computation. Then, *try-finally* block has been introduced

and assigned the weight as 2 without considering the potential catch blocks. They have emphasized the similarity between the behavior of *if-else* and *try-finally* blocks in sequence category. The *try* block execution of the code moves to the *finally* block in case of exceptions, which is the same behavior of *if-else* statements. Therefore, *try-finally* block has been assigned with weight 2. But, the set of *catch* block is represented with weight 3, which is the same weight of *switch-case* statements. Because, it executes a block of a source code depending on the condition as in *switch-case* statements. Repeatedly, the reason behind the same weightage allocation with all looping criteria has not been mentioned in this work as well. Later, they have defined the *Cognitive complexity for a single Method* (*MC*) as shown in Equation (10.1).

$$MC = log2\,(1 + [1 + MC^1\,(methodBCS)]) \tag{10.1}$$

They have identified that each method represents a sequential BCS to emphasize the entire method body, in which its weight is 1. Moreover, there can be another linear BCS (weight = 1) inside the sequence BCS, which is denoted as *methodBCS*. Then, *MC* is going to be calculated as the summation of both these weights (2) and the weight of arbitrary BCS recursively, as each BCS might contain nested BCS. However, the complexity computation derived through logarithmic base 2 has not been described properly. In addition to that, cognitive load calculated only by using BCS is another drawback in this computation. Accordingly, following equations have been used to compute the *Weighted Class Complexity* (*WCC*) of a class with *m* methods.

$$WCC^1 = AC + \sum_{i=1}^{m} MC_i^1 \tag{10.2}$$

$$WCC = log_2(2 + WCC^1) \tag{10.3}$$

Equation (10.2) and (10.3) assist to compute the complexity of a class as *WCC*, which considers the sum of all method complexities in a class (WCC^1). The summation of weight 2 in Equation (10.3) has been derived with the same explanation of adding weight 2 in Equation (10.1). WCC^1 is computed by considering *Attribute Complexity* (*AC*) in all methods and the complexities due to BCS (MC^1). Attributes are local to every instance of a class and can be accessed by several methods. Therefore, *AC* is assigned to *a*, where *a* is the total number of attributes inside the class. Similarly in Equation (10.1) and (10.3), the rationale of considering the logarithm of base 2 has not been mentioned. Finally, they have expressed the complexity of the entire system as *Code Complexity* (*CC*) as follows.

For the calculation of *CC*, the complexities of all classes and their relationships have been deemed. Inheritance relation has been used for the relationship mapping since any class may be a parent or a child class of other classes. It is noted that the summation of individual *WCC* is performed if the classes with disjoint hierarchies or classes in same hierarchical level of a class hierarchy are observed. The weights of the classes are multiplied if the classes are in same class hierarchy. Accordingly, the *CC* of an object-oriented source code with *d* levels of hierarchical depth and level *j* with *c* classes is denoted as,

$$CC = log_2(1 + \prod_{j=1}^{d}[\sum_{k=1}^{c} WCC_{jk}^{1}]) \quad (10.4)$$

According to this work, the cognitive complexity of an object-oriented source code has been evolved with BCS, attributes, methods and inheritance concept. The limitation of considering only the inheritance under relationship types can be highlighted. Moreover, the process of assigning the cognitive weightages has not been mentioned, which results the above computation as invalid.

Another cognitive complexity metric was proposed by Chhillar and Bhasin [35] indicating that inheritance level of statements in classes, types of control structures, nesting of control structures and the size to determine the cognitive complexity. They have calculated the cognitive complexity of an object-oriented source code using Equation (11), where *Cw(P)* is the weighted complexity measure of program *P*, S_j is the size of j^{th} executable statement in terms of token count (operators + operands + methods/functions + strings) and W_t is the total weight of j^{th} executable statement, which can be obtained as the addition of the weight due to inheritance level of statements, types of control structures and nesting level of control structures as listed in Table 7.

$$Cw(P) = \sum_{j=1}^{n}(Sj) * (Wt)j \quad (11)$$

Table 7. Cognitive Weights Defined under [35]

Category	*Description*	*Cognitive Weight*
Inheritance level	Base class	0
	First derived class	1
	Second derived class	2
Control structures	Sequential statements	0
	Branch statements (if – then – else)	1
	Loops (while, for, do-while)	2
	Switch-case statements with n cases	n
Nesting control structures	Sequential statements	0
	Statements inside outer most level of control structures	1
	Statements inside next inner level of control structures	2

The architectural aspect of the source code has been computed through S_j. The control flow and the relationships of the object-oriented source code has been obtained through W_t. According to Table 7, the cognitive weight is incremented along with the inheritance level incrementation, which verifies the cognitive effort addition in each layer. But, the limitation of inheritance being the relationship type is repeated in this work as well. In control structures, all types of branch and looping criteria have been assigned with same weights without specifying the complexity variation among multiple controllers in same category. The weightage assignment of *switch-case* statements can be stated as valid as it tends to be increased with respect to the number of case statements used. Further, the nested control statements have been handled by assigning another cognitive weight based on the nested level, which tends to be increased along with the nested hierarchy. Since the logic gets more complex with the nested levels, the mechanism of assigning the cognitive weights for nested structures can be mentioned as valid.

Furthermore, Wang and Chiew [36] mentioned the cognitive complexity of a product in terms of functional complexity through seven definitions. Definition one indicates that software functional complexity should be measured with a two-dimensional metric as a product of operational and architectural complexities. Subsequently, following subsections have been assigned with cognitive weightages as listed in Table 8.

Table 8. Cognitive Weights Defined under [36]

BCS	Calibrated Cognitive Weights (W_i)
Sequence	1
Branch	2
switch	3
for-loop	7
repeat-loop	7
while-loop	8
Function call	7
Recursion	11
Parallel	15
Interrupt	22

The weightages in Table 8 have been quantitatively determined based on a series of empirical experiments. It is significant that they have emphasized the variation of the absolute value of BCS cognitive weights for each individual in program design and comprehension. However, they have claimed that relative cognitive weights between the ratio of $w^1(BCS_i)/w(BCS_1)$ where $2<=i<=10$ remains unchanged. Accordingly, they have eliminated the subjectivity among the comprehension level of each individual and determined the objective cognitive weights of ten BCS categories. This procedure can be considered as one way of expressing the subjective measurements in a quantitative manner. However, removing the subjectivity along with the concept of ratio cannot be stated as a better way of handling subjective measurements, and hence, it creates an ambiguity with the definition of cognitive complexity. They have specified two ways of structural BCS patterns inside a software system such as sequential and embedded BCS. In earlier systems, sequential BCS were seen such that the source code layout has become linear. Therefore, the operational complexity has been calculated as the sum of all linear BCS. However,

modern logical implementations have been coded with embedded BCS, so that the operational complexity has to be computed with the sum of sums of cognitive weights of inner BCS. Since there may be several variations of both BCS configurations, a general method of deriving the operational complexity has been defined in definition 2. According to definition 2, the *Operational Complexity of a software system S, $C_{op}(S)$* is computed as the sum of its *n* linear blocks enclosed with individual BCS, which has been denoted by Equation (12.1).

$$C_{op}(S) = \sum_{k=1}^{n_c} C_{op}(C_k) = \sum_{k=1}^{n_c} \sum_{i=1}^{m_k} w(k,i) \quad (12.1)$$

The cognitive weights for each liner BCS can be taken through Table 8. The logic behind Equation (12.1) tends to be valid as the BCS determine the operational behavior of source codes. Hence, computing the operational complexity by obtaining the summation of cognitive weights assigned for each BCS in each liner block can be acceptable. Since they have maintained relative cognitive weight by $w^1(BCS_i)/w(BCS_1)$ as stable to reduce the comprehension effort subjectivity, definition 3 implies its calculation process as follows.

$$w(BCS_i) = \frac{w^1(BCS_i|2<=i<=10)}{w^1(BCS_1|w^1(BCS_1) = w^1(SEQ) = 1)} \quad (12.2)$$

As implies in Equation (12.2), the relative cognitive weight of a BCS is computed as a relative ratio between the tested comprehension effort $w^1(BCS_i)$ and the reference comprehension effort of the sequential BCS_1, which is equivalent to 1. Even though the ratio between the tested and reference comprehension efforts maintains as unchanged, they have to be varied among each individual. Therefore, the computation of the relative BCS weight can be stated as unrealistic. Through definition 4, the explanation of *operational complexity unit* has been introduced. The operational complexity unit has been denoted as a single sequential operation called a *unit function*, which is equivalent to 1. They have further elaborated the operational complexity of a code with only sequential statements equivalent to 1, where the operational complexity is going to be reduced to the symbolic complexity LOC. As the next complexity, they have emphasized the functional complexity of a software. The functional complexity is denoted with the architectural complexity, which is proportional to the number of local and global data objects such as inputs, outputs, data structures and internal variables used in the program. As such, the *architectural complexity of a system S*, ($C_a(S)$) has been defined using the number of data objects and the component levels and mentioned in definition 5 as follows.

$$C_a(S) = OBJ(S) = \sum_{j=1}^{n} OBJ(UDM_j) + \sum_{k=1}^{n_c} OBJ(C_k) \quad (12.3)$$

According to Equation (12.3), *OBJ* is a function that counts the number of data objects in a given data structure. The summation of the count of global variables known as *Unified Data Model* (*UDM*) and the count of all local variables in n_c components have been considered through the second component. Thereby, it can be claimed that this equation quantifies the complexity related to the architectural flow inside a given source code with respect to all possible data categories. Moreover, they have initiated the definition 6 with the concept called *unit of architectural complexity*, which is the complexity of a single data object modelled either globally or locally, and it has been assigned to 1. By elaborating the software architectural and operational complexities, the cognitive complexity of a software system has been introduced by its functional complexity and the size. Accordingly, the *Cognitive Complexity of a software system S*, $C_c(S)$ is computed as a product of $C_{op}(S)$ and the $C_a(S)$.

$$C_c(S) = C_{op}(S) * C_a(S) \quad (12.4)$$

Equation (12.4) highlights the necessity of expressing the cognitive complexity in terms of functional complexity as the functional complexity determines a collection of operations applied to a set of data objects inside the software. Moreover, the functional complexity is directed with the architectural and operational complexities as an operational functionality can be increased with respect to higher number of data objects, which increments the functional complexity as well. A higher functional complexity implies a higher level of processing inside the software, so that a considerable amount of human effort is acquired for its comprehension comparing to a less processing. Hence, Equation (12.4) validates the practicability of considering the functional complexity to denote the cognitive complexity. Further, they have initialized the concept called *unit of cognitive complexity* through definition 7, which a single sequence operation is applied to a single data object. Consequently, they have defined the physical aspect of the functional complexity by the number of functional objects modeled inside the given software system. Additionally, they have emphasized the necessity of computing the functional complexity quantitatively and expressing it as the cognition level of the software. Therefore, this achievement can be introduced as a mechanism of quantifying the cognitive complexity through a reasonable number of factor consideration.

Surprisingly, a methodology of computing the cognitive complexity can be widely seen in SonarSource software projects integrated with SonarCloud and SonarQube[1][12]. This is considered as the recent mechanism of computing the cognitive complexity. It introduces the cognitive complexity as referred to as SonarSource Cognitive Complexity (*SSCC*), which indicates the measure of difficulty level to understand and maintain the control flow of a method [2]. Significantly, it has avoided the usage of objective mathematical models to assess the software complexity, which have used in all the previous works. The concept of control flow suggested in cyclomatic complexity [37] has been considered for the computation of *SSCC* but, in addition, it considers the difficulty level of each BCS irrespective to the computation in cyclomatic complexity. In other words, although the cyclomatic complexity becomes same for two different codes, the user cognition related with these codes can be different. That difference is represented by SSCC metric with respect to the general users. One of the exemplifications to illustrate that work can be observed in Figure 3.

```
//Cognitive complexity 7
 int sumOfPrimes (int m)
 {
        int total = 0;
        // +1
        B: for ( int i = 1; i <= m; ++ i)
        {
                // +2 (1 by nesting)
                for ( int j = 2; j < i; ++ j)
                {
                        // +3 (2 by nesting)
                        if ( i % j ==  0)
                        {
                                continue B;   //+1
                        }
                }
                total += i;
        }
        return total;
}
```

```
//Cognitive complexity 1
 String getWords (int n)
 {
        // +1
        switch (n)
        {
                case 1:
                        return "one";
                case 2:
                        return "two";
                case 3:
                        return "three";
                default:
                        return "< three";
        }

}
```

Figure 3. Example of *SSCC* Calculation [12]

[1] https://www.sonarsource.com/products/sonarqube/

By referring to Figure 3, it can be observed that *sumOfPrimes()* has 2 *for* loops and 1 *if* conditional statement, so that the cyclomatic complexity is 4 due to the number of decision statements are being 3 and adding of 1 according to the complexity computation. Similarly, *getWords()* consists with a *switch-case* statement with 3 cases, so that its cyclomatic complexity is also equivalent to 4. However, the control flow of *sumOfPrimes()* is can be observed as complex than *getWords()* due to the nested BCS and the usage of *continue* statement. On the contrary, the control flow of *getWords()* is more understandable with the regular usage and syntax of *switch-case* statements. Although this complexity variation is not achieved by cyclomatic complexity metric, *SSCC* is capable of illustrating such that *SSCC* of *sumOfPrimes()* is assigned to 7 and *getWords()* is 1. It is noted that the higher value of *SSCC* demonstrates a higher user comprehension effort with high cognitive complexity. Furthermore, the facility of changing the maximum value of *SSCC* has also been introduced, although SonarQube suggests having it to 15 as the maximum cognitive complexity. The calculation of *SSCC* has been performed as follows. The first BCS is assigned with weight 1, and each inner BCS is assigned with the nested level accordingly. In *sumOfPrimes()* outer for loop is assigned to 1 and the inner *for* loop and *if* condition has been assigned with 2 and 3 respectively based on their nested levels. Then, the continue statement inside if statement is assigned with 1. Then, its total cognitive complexity is made into 7. Similarly, *getWords()* complexity is assigned to 1 as it contains only a single *switch-case* statement. Accordingly, a source code with more inner BCS will obtain a higher *SSCC* than a source code without nested BCS, which clearly describes the human capability of understanding a source code with nested and sequential BCS. Even though *SSCC* shows a slight variation among the objective computations of current software complexity metrics, the limitations of considering the human cognition with respect to BCS and the exclusion of real personal profile has not been resolved.

In addition to that, Dissem et al. introduced a software metric called *Asymptotic Path Complexity (APC)* which computes the mental effort required to comprehend a source code based on subjective programmer experience, time to complete the program comprehension tasks, task performance and functional magnetic resonance imaging measurements of human brain [38]. *APC* has been computed with the number of steps (n) executed by the Program P and the number of steps of execution with length l. As such, *APC* of a program P has been denoted as a function $f(n)$ such that the number of different executions of P with $l <= n$ is given as in Equation (13).

$$APC = O(f(n)) \; ; \; l <= n \tag{13}$$

Along with theoretical and empirical validation, they have concluded that *APC* is more refined and meaningful metric than the available control flow metrics like cyclomatic complexity metric. Further, *APC* has correlated with several measurements directed with code comprehension resulting it as another achievement to indicate as cognitive complexity.

2.3 Challenges in Cognitive Complexity Quantifications

Although the cognitive complexity has been quantified as a metric and for ease of use, there are numerous drawbacks with the usage of these cognitive complexity metrics in real applications. Basically, the parameters considered through source code aspect in each computation are different, so that the computations proposed from each work are varied with each other. The categories considered under source code aspect to determine the cognitive complexity can be preliminarily divided into architectural and spatial aspects. The spatial aspect expresses the size of the source code or the distance from a module call from its calling to the implementation in terms of LOC, so that a significant difference of the categories under it cannot be observed. The architectural aspect has been expressed in numerous ways, as it defines the amount of information inside a source code. Accordingly, the possibility of getting varied complexity values for a same source code can be attained, which derives the problem of attaining a single measurement as the cognition effort. Nevertheless, the number of parameters that can be considered through source code aspect to define the human comprehension cannot be limited as the contexts that the source code complexity can be viewed are unlimited. Hence, the possibility of having a computation which demonstrates the entire source code aspect with respect to the human comprehension is unachievable.

Secondly, the inclusion of personal profile can be observed as comparatively lesser than source code aspect involvement in these metrics. The involvement of personal profile has been introduced using cognitive weights which indicates the difficulty level of understanding for selected source code categories. However, there are significant number of drawbacks consisted within these weightages. The assignment procedure of cognitive weights has not been clearly mentioned in majority of the works. Expressing the difficulty levels using quantified weights can be explained as a vital achievement, and it has to be performed through a standard procedure. The experimental background used in [34] to express the weights can be accepted, while the other distributions are merely the assumptions. Nonetheless, the problem of using these weights for

entire user population is still remaining as their applicability is limited only for that user group. The assumption based cognitive weight allocation cannot be validated, as assumptions cannot be applicable for most of the real scenarios. Furthermore, the user preference should be considered for the comprehension effort as the effort utilized for an easily understandable category is less, and the comprehension effort taken to a non-preferable category is high. Therefore, the inconsistency of assigning different source code categories which perform the same functionality with same weightage can be raised.

To conclude, non-limitation of the source code properties taken for the human cognition, inconsistencies observed in cognitive weights are the major drawbacks of the current cognitive complexity metrics, which should be addressed to standardize the cognitive complexity calculation process in the form of metric.

2.4 Cognitive Complexity Expression with Software Attributes

As per the definition of cognitive complexity, the subjectivity associated with the human comprehension level has built the path of not confining it to a metric calculation. Consequently, the usage of cognitive complexity concept has been used to define the other software attributes. On the other hand, the usage of other software attributes has been considered to define the cognitive complexity. As such, the cognitive complexity being a predictive factor can be observed in several previous works.

Hansen et al. have found that there is a strong relationship between the cognitive complexity and the source code, expertise and correctness using Python source codes [39]. The source code of software is a possible milestone to demonstrate its underlying logic. Hence, the comprehension of a software can be determined through the understandability of its source code. The correctness of a source code is another essential parameter for the understandability. A source code with a smaller number of errors can lead to its logic directly so that the comprehension effort can be reduced to lessen its cognitive complexity. If a source code consumes a high number of errors, a considerable amount of effort has to be utilized to avoid the errors, and then the effort has to be given for the comprehension. Therefore, an erroneous source code would lead to a high cognitive complexity. Accordingly, they have identified that the errors of a source code as a direct parameter to determine the cognitive complexity. On the other hand, they have introduced the

expertise of analyzing a source code can express its cognitive complexity. If an individual's skill level of comprehending the logic of software is high, the effort that has to be utilized for the comprehension process is low, which results a lesser cognitive complexity. The cognitive complexity becomes higher for a user, whose skill level is lower for the comprehensibility so that understanding requires a considerable amount of effort. Therefore, the analysis of the cognitive complexity in terms of user expertise can be mentioned as valid. Noteworthy, they have extended the same analysis for other programming languages using more realistic programs to observe the parameters effecting for the cognitive complexity. Consequently, they have observed that the same parameters can be applied for the user comprehension regardless of the programming languages used. That can be evaluated through the variation among the comprehension efforts associated with different programming languages. In other words, although the expertise level and the number of errors can be varied with respect to the programming language used, these parameters can still be considered to determine the cognitive complexity of a given source code.

Several research works have introduced that the cognitive complexity can be used to identify the locations of the software with high frequency of errors and to evaluate the quality of the software [40], [41]. As it has been mentioned earlier, errors of a source code can control the cognitive complexity. Subsequently, a code segment with high frequency of errors can result a high cognitive complexity, since the understandability level of the erroneous code is evidently high. Similarly, a code with less errors can imply a less cognitive complexity. The cognitive complexity of a simple software is relatively low as it can be easily comprehended. Furthermore, the applicability of cognitive complexity to assist programmers to detect code defects effectively using a novel approach called Human-Machine Pair Inspection has been carried in [42]. Moreover, the ability of a software to be functioned as per the user requirements is known as software quality [43]. As errors violate the possibility of proper functioning, the necessity of having an erroneous source code should be maintained in order to derive a quality software and to achieve a lesser cognitive complexity.

The significance of evaluating the relationship among understandability and cognitive complexity has been performed in [4], [5], [10], [13], [44]. They have clearly outlined that a software with lesser understandability denotes high cognitive complexity. Similarly, a software which can be easily understood derives a less cognitive complexity. Recent research conducted by Lenarduzzi et al. highlights the importance of having cognitive complexity as an indicator of expressing user understandability [45]. Their case study involved with the manual inspection of a set of Java

classes that exhibit different code complexity levels such as cognitive complexity and cyclomatic complexity measured by SonarQube. Based on the users' rating levels, they have interpreted that cognitive complexity being a vital determinant to measure user comprehension comparing to the other metrics like cyclomatic complexity. Moreover, Wyrich et al. have found the subjective comprehension level of users are cognitive biased and depend on the factors like anchoring effect [7].

A determination of predicting the success of software projects using cognitive complexity has been proposed by [2]. A successful project is referred to its functionalities are being processed according to the customer requirements, which its quality becomes high. According to [40] and [41], a quality project is found to have less comprehension effort so that its cognitive complexity should be low. Therefore, the concept of cognitive complexity is taken as a selective factor to determine the success rate of a project can be stated as valid.

Akman et al. have initiated that the simplicity is the best way to deal with the software complexity, and it is crucial to apply the concept of simplicity inside the software development to measure error prone, time consuming and complex activities [46]. This can be verified through the cognition efforts that error prone, time consuming and complex activities consume. As it is already verified in [39] - [41], there is a proportional relationship of the errors of a source code and its cognitive complexity. An activity is known to be complex, if it is difficult to understand. Hence, a complex activity consumes a considerable amount of time to comprehend, so that it becomes a time-consuming task. Accordingly, a complex and time-consuming task can reach to a high cognitive complexity due to the high effort allocated for its logical comprehension. Furthermore, they have identified that cognitive complexity plays a leading role of developing software projects. The reason behind that is the consideration of the varied cognition abilities among the development team through different task allocations. Each member of the development team has different capabilities of handling software. Therefore, it is essential to categorize the team members according to their cognitive capabilities and assign the processes accordingly. This describes the necessity of assigning the complex tasks to the team members with high expertise and high cognitive load balancing, where the simple tasks have to be allocated to the other members in the development team.

Another different approach of exploring the relationship between current source code metrics with cognitive load has been described in [47], which can be treated as a major milestone in

cognitive complexity concept as well. It has been found that the difficulty of the software development can be indicated as the difficulty level associated with its source code. Therefore, reason behind its complexity computation by considering certain aspects of the source code has been verified as true. However, it has been introduced that this difficulty is associated with the increment of users' cognitive load which can be estimated using psycho-physiological measures, so that the complexity can be denoted with respect to the cognitive load. Accordingly, the existence of the impact of cognitive load with cognitive complexity can be analyzed as follows. The difficulty level associated with a source code or the user understandability of a source code can be expressed with its cognitive complexity. Therefore, the possibility of indicating cognitive complexity through users' cognitive load can be emphasized as a significant approach.

Numerous research works on the applicability of cognitive load with user comprehension has been assessed in [48]. In there, the accuracy of current software metrics to determine the user understandability has been questioned. Consequently, it has proposed to use wearable body sensors to provide cognitive perspective on code comprehension and software metrics. Furthermore, Hao et al. have conducted an experiment to observe the accuracy of code complexity metrics which include cyclomatic complexity, Halstead complexity and cognitive complexity provided by SonarQube [49]. Accordingly, they have experienced the failure of above-mentioned metrics to capture the complexity perceived by the programmers in terms of their cognitive loads. These situations imply the necessity of introducing a proper metric which indicates the cognitive load effecting with user comprehension. As the user comprehension is related with cognitive complexity concept, expressing cognitive complexity in terms of cognitive load can be verified over again.

2.5 Applicability of Cognitive Complexity in Other Domains

The comprehension effort of a person should not be limited only for the software development process, as its applicability can be expanded into wider domains. Therefore, the applications of other domains have been considered to observe how the concept of cognitive complexity has been used within them. Gruhn and Laue have introduced a mechanism to implement the modelling elements to reduce the cognitive complexity in Business Process Models (BPM) [50]. The underlying logic behind that concept is to reduce the unnecessary misunderstandings that

user might have with the manual comprehension process. Therefore, by implementing the modelling elements they have expected to reduce the cognitive load effected to the unrequired misunderstandings and to gain a less cognitive complexity. Moreover, they have proposed an automatic search and pattern finding algorithm for the patterns to improve the comprehensibility, which in terms found to reduce the cognitive load. Furthermore, they have searched to implement more modelling elements to reduce the cognitive complexity in terms of cancellation of processes and exceptions in the control flow. Hence, reducing the opportunities that might occur with misunderstandings can be treated as one procedure to achieve a lesser cognitive complexity. Moreover, they have analyzed the applicability of visualizing BPM using Unified Modelling Language (UML) diagrams in order to communicate with stakeholders inside the software development process, and to use them to make the understanding and maintaining processes easier [51]. They have found that the visualization process can indicate the logic clearly and in a consistent manner, so that users can recognize it easily. The reduction of the cognitive effort with their visualization has been verified using the cognitive complexity metric proposed by Shao and Wang [25]. Thereby, the logic representation using visualization techniques can be granted as a way of simplifying the human cognition effort.

An attempt to verify the human cognition the air traffic have been discussed in [52]. In there, a model to simulate the air traffic has been developed and applied for a comparison using different complexity factors. Consequently, it has been found that the effectiveness of cognitive complexity in terms of attention, decision making, memory and perception can control the level of air traffic. It has been described as the cognitive load related to the attention, decision making, related memory and the perception can make the air traffic controlling process easy, so that they can assist to reduce the cognitive complexity. Moreover, another approach was achieved by initiating the Brahms Generalized Uberlingen Model (Brahms-GUM), which simulates the cognitive behavior of aviation work practices [53]. It was developed by analyzing and generalizing the roles, systems and events, which leads to an aircraft collision. It also simulates the asynchronous behavior of distributed processes in which the sequence of temporal interactions can be mutually unpredictable and constrained with each other. The logic of these simulations has been stated as the processes of assisting the cognitive load, which can reduce the cognitive complexity of aviation practices by reducing the comprehension effort related with each user.

The necessity of handling the gaming levels through cognitive complexity concept has been explored in [54]. It is found that the competitions designed by analyzing the participants' comprehension ability have improved the performance and anxiety of low-achieving students, while achieving smoother tasks. It is therefore ensured the possibility of designing the games by analyzing the cognitive levels of each participant without maintaining a static gaming environment for all users. Hence, the possibility of increase the interaction and the performance of gaming participants has been verified through this work.

As another approach, Presbitero has explored the influence of cognitive complexity in career paths. Accordingly, he has indicated the influence of proactive personality and cognitive complexity between proactive career planning and proactive career enacting [55]. Consequently, he has verified that the employee's proactive personality and the cognitive complexity assist to strengthen the relationship between proactive career planning and proactive career enacting, which can be used for the development of organizations as well. Therefore, the comprehension effort utilized by an individual with regard to the career along with the personality can be believed as predominant factors beneath the success of the persons' career path and the organization.

The impact of socio-cognitive complexity in the domain of theater applications is found by Silva et al. [56]. Their study has involved a construction of an instrument which can explore the complexity of thoughts among actors, directors and other practitioners in theaters. Consequently, they have verified lower cognitive complexities regarding the theater applications among individuals with high experience in acting and directing. Thereby, they have verified the possibility of analyzing the psychological impact of theaters based on socio-cognitive complexity. Further, the assistance of the experience of a person regarding a particular domain to reduce the cognition effort can be stated through this study. Therefore, the concept of cognitive complexity has been verified as a practicable factor inside wider domains, which ensures its applicability expanded beyond the software and its complexity computation.

2.6 Standardization of Cognitive Complexity Metrics

Each complexity metric should be empirically and theoretically verified with its accuracy, stability and the practicability in real processes for its wider usage, to avoid the variety of metric introductions to compute the same complexity. For that purpose, standardized software properties and frameworks have been introduced, and proposed complexity metrics must be verified through them. The theoretical validation of the proposed metrics has been conducted through Weyuker properties [57] and Briands' framework [58]. Weyuker has defined nine properties to be satisfied by a newly proposed metric to ensure its practical usage, while Briands' framework consists with five modular complexity properties to be satisfied. The satisfaction of majority of the properties inside these frameworks has proven the applicability of the proposed metric in real applications. Noteworthy, several cognitive complexity metrics have been validated through Weyuker properties and Briands' framework to verify their usages.

The cognitive metric proposed by Misra [59] has been validated through Weyuker properties. The metric has successfully proven upon seven properties through its evaluation process. Furthermore, the extended *Cognitive Information Complexity* measure (*CIC*) introduced by Kushwaha and Misra [60], [61] have evaluated their metric under Weyuker properties to validate its stability of usage. Another major approach of determining the cognitive complexity through spatial complexity of an object-oriented source code by Gupta and Chhabra [62] have also used Weyuker properties as their validation framework. An inheritance cognitive complexity metric proposed by Misra et al. [29] was validated through the properties defined under Brainds' framework for its stability. The object-oriented cognitive complexity metric proposed by Misra [59] has been successfully evaluated through Briands' framework by satisfying all the properties listed. Furthermore, the cognitive metric proposed for object-oriented codes with respect to the spatial capacity by Chhabra and Gupta [63] has been verified through Briands' framework. Another metric introduced for object-oriented codes by Misra and Akman [64] has also been validated through both frameworks in order to prove its usage in real applications.

Although the stability of the proposed cognitive complexity metrics has been theoretically proved in terms of Weyuker properties and Briands' framework, the empirical validations to validate their practicality has not been performed. Moreover, the validations of the computational processes introduced by these metrics has also not been indicated. The complexity values computed through proposed cognitive complexity metrics are validated for a selected set of users,

since the cognitive weights used are based on them. Nevertheless, those cognitive weights are assumptions-based values which are based on selected users' experience. Although these weightages are assumptions, their validation has to be processed to be applicable within that user group, which has not been performed. Subsequently, the computational process of these metrics over the selected user group cannot be observed. Apart from that, the applicability of these computations over the entire user population must be proved statistically to ensure the standardization. However, none of these metrics are able to standardize their metrics over statistical and hypothetical procedures, which raise the problem of generalizing their computational processes. This situation leads to observe a significant number of drawbacks with their computational processes which cannot be granted to be denoted as the cognitive complexity. Hence, a cognitive complexity metric should be theoretically and empirically validated, and statistically capable of proving its usage in real applications to demonstrate the user comprehension effort of a given source code.

2.7 Summary

There has been a considerable number of efforts proceeded to express the cognitive complexity of a given software by analyzing its source code and presented it quantitatively by defining cognitive complexity metrics. Most of the computations are based on the source code properties which have been expressed through the architectural and spatial aspects. The architectural aspect is considered to be the amount of information inside a source code, which can be expressed in numerous ways. The spatial aspect is denoted through LOC. Hence, introduction of different cognitive complexity metrics can be stated due to the varied ways of expressing the architectural aspect. The consideration of variables, attributes, input output parameters, operators, operands, BCS, functional complexity, operational complexity and object-oriented concepts like inheritance, coupling and cohesion are some of the categories that the architectural aspect has been expressed in different research works. The effectiveness of the human comprehension is included as a form of a weightage in these metrics. It is a numerical value denoted as cognitive weight, which has been assigned based on assumptions taken regarding the experience levels of a target user group. Assumptions cannot be considered as valid procedure of introducing cognitive weights, thereby the problem of validating the cognitive complexity metrics has arisen. Even

though these weights have been assigned through an experimental background, the limitation of those weights only to the experimental user group exists. Non-consideration of the personal preference of using the source code controllers can also be observed, as multiple controllers with same functionality have been assigned with the same weight. Moreover, the inclusion of personal profile cannot be accepted in these metrics as they emphasize more on source code aspects.

Rather than limiting the cognitive complexity into a quantifiable value, the applicability of it as an analytical and predictive factor based on other software attributes has been explored by several works. Consequently, the user understandability, user expertise, error prone source codes, time consuming and complex tasks and success rates of software projects and products have been presented as the parameters of controlling the cognitive complexity. This accomplishment can be considered as a vital achievement to assess the qualitative aspects effected for the human understandability effort apart from the mathematical quantifications.

The applications of the concept of cognitive complexity in other domains can be treated as predominant instances to describe cognitive complexity. As such, the possibilities of using the cognitive complexity in BPM, aviation work and air traffic practices, career organizations and theater applications have been observed. Accordingly, the concept of process automations, simulations and the experience effected in a particular domain have been highlighted to reduce the cognitive load, and thereby to achieve a reduced cognitive complexity.

To conclude, the expression of cognitive complexity can be observed as a form of metric and as an analytical and predictive factor within the context of software. Moreover, its applicability can be observed widely in other contexts as well. However, the problem of expressing a standardized mechanism to apply and use the cognitive complexity in software development process is still existed.

3.0 METHODOLOGY

3.1 Introduction to Methodology

The main purpose of this research project is to analyze the procedures of applying the cognitive complexity through software development and maintenance processes. The main rationale for this scenario is the non-existence of a standard mechanism of using the cognitive complexity concept inside the software development process, although it has been used as a form of numerous metrics and as a factor of qualitatively determining the other software attributes. Therefore, using this concept inside the software development results inconsistence situations. Hence, this research works attempts to solve this scenario as follows. The main factors that can be used to express the cognitive complexity and the sub factors associated with each main factor have to be explored to create the path of describing it in detail. Subsequently, procedures of handling the human cognition have to be explored by considering the main and sub factors related with human comprehension. A software with lesser cognitive complexity is one of the main expectations inside its development process [12], so that mechanisms of reducing the cognitive complexity should be attained by analyzing the procedures which can control the human cognition. Accordingly, a computational design of the procedures that can be used to reduce the cognitive complexity is implemented. Additionally, an introduction of a cognitive complexity metric has been performed, which can solve the drawbacks mentioned in literature. At last, the evaluation of the computational design and the metric has been accomplished to verify their usage in real applications.

3.2 Analysis of the Factors and Sub Factors Effecting for Cognitive Complexity

The comprehension effort related with a software is referred to its cognitive complexity, and it is expressed by analyzing the source code implemented for it. Even though the comprehension effort is analyzed for a software, it is derived through its source code. The same practice has been followed by existing software complexity metrics, as the source code is the major evidence of detecting the logic of a software [65]. Therefore, the usage of source code to determine the comprehension effort can be accepted to define the cognitive complexity of its software. Since

the cognitive complexity is directed to the understandability, the responsible party of understanding the software logic should be analyzed. Understanding the logic of a source code is performed by two parties. The logical identification of a software should be performed by the software team members engaged within the software development process, as they cannot implement the software without comprehending its logic. Furthermore, the source code logic is essential in maintenance process, since they should acquire new changes and modify the current functionalities by identifying the logic of the source code. On the other hand, the computer is the other party that requires source logic of the source code. Once the source code is implemented, it should debug, execute and run to produce the outputs. These operations are performed once the source code is transformed into a computer readable format. The expected outputs are produced after the comprehension process of the source code is completed by the computer. Therefore, the comprehension of a source code logic is performed by software development team members and the computer. It should be noted that this research work analyzes the comprehension effort utilized by development team members as the cognitive complexity, and it has been denoted as the human comprehension effort. Therefore, as the preliminary task, the factors related with the human cognition effort has been explored.

Based on the concept of analyzing the human comprehension effort of a source code, the cognitive complexity can be viewed by two main aspects. The effort utilized for the comprehension is gained from humans, which is referred to the team members in the software development process. Hence, one major factor of expressing the cognitive complexity can be stated as personal profile. The comprehension effort is measured from the source code representing a particular software. Therefore, consideration of the source code for the cognitive complexity cannot be ignored. Herein, the cognitive complexity is presented using two major factors namely personal profile and source code. Then, the sub factors of these two main aspects have to be explored to imply the relationship to the cognitive complexity. The possible sub factors identified under personal profile and source code factors can be observed in Figure 4.

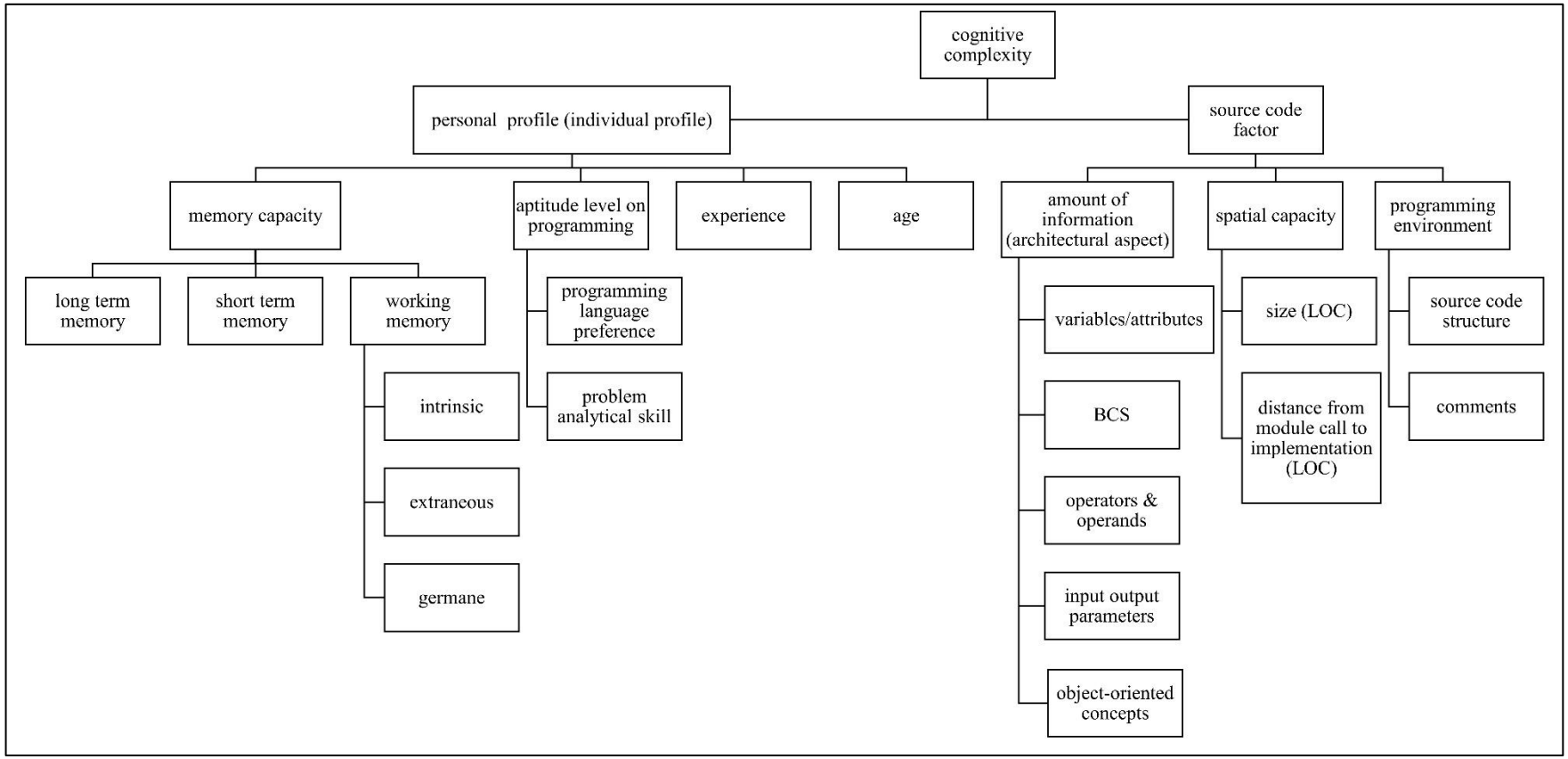


Figure 4. Factors Effecting with Cognitive Complexity

According to Figure 4, personal profile related to cognitive complexity has been categorized as follows. Capacity of memory of an individual is a vital factor of determining the cognition effort [47]. A person with high capacity of program related memory requires a less effort for the logical comprehension, and a person with less capacity of program related memory requires a high effort for the logical comprehension. Accordingly, a high capacity of program related memory leads to a less cognitive complexity, while a less capacity of program related memory can lead to a high cognitive complexity [48]. Further, memory of a human is categorized into long-term, short-term and working memories. Long-term memory refers to vast storage of knowledge and information about previous events, which remains for a longer duration and difficult to deny by an individual. The short-term memory holds for a limited amount of temporary information, which can be accessed very easily, while working memory refers to the memory used to plan and carry out a given task and it is not completely distinct from short-term memory [66]. Then, the applicability of the types of memory for a source code comprehension should be analyzed. The long-term memory applies for the programming language, tools and technologies that the user is aware for a longer duration. The usage of the information such as variables, BCS, input output parameters, and functions inside a given source code can be considered under the short-term memory. The working memory is applicable at the time of dealing with a particular source code, which attempts to find a pattern of the logical flow of the source code by connecting with long-term and short-term memories. Therefore, the capacities of these three types of memory should be acquired for the logical comprehension of a source code. The reluctance of these memories can result to require more effort on understanding the source code, which outcomes a high cognitive complexity. However, working memory is the determinant factor of understanding the logic behind a software at a given time, as it is the type of memory applied with that duration along with the assistance of long-term and short-term memories. Hence, more emphasis has to be given for the working memory when analyzing the comprehension effort taken by a user within a given time period. Accordingly, the capacity of working memory has been taken to consider the amount of comprehension effort for a given source code. The amount of information that the working memory can hold for a given time period is called as the cognitive load [67]. An individual who is capable of maintaining high amount of program related information inside the working memory can easily understand that code without more effort, while an individual with less cognitive load should have more effort for understanding. As such, a high program related cognitive load directs to a less cognitive complexity, and a less program related cognitive load

directs to a high cognitive complexity. Furthermore, the cognitive load can be categorized into three types namely intrinsic, extraneous and germane [68]. Intrinsic cognitive load determines the amount of information that should be processed to comprehend a given logic. It depends on the complexity of software and the number of elements that have to be kept in the mind at the same time. Therefore, intrinsic cognitive load can be referred to as the memory required for necessary processing, and it should be maintained in a higher state to increase the understandability and achieve a less cognitive complexity. Similarly, a lack of intrinsic cognitive load reduces the required memory for the comprehension, which reduces the understandability with high cognitive complexity. Extraneous cognitive load refers to the cognitive engagement with excessive distractions, which does not support for actual understanding. As an exemplification, this type of cognitive load is utilized for navigating the code segments in an unstructured source code, so that it can be considered as an unnecessary effort utilized for the comprehension. Therefore, the extraneous cognitive load should be maintained in a lower state to lower the unnecessary cognitive load spent on the comprehension process, thereby ensuring a lesser cognitive complexity. A high amount of unnecessary cognitive load reduces the possibility of comprehending the actual logic, so that it results with a high cognitive complexity. Germane processing includes a deep cognitive load which is directed with the prior knowledge of the user. This includes organizing the contents mentally and combine them with the prior knowledge, which assists for the long-term memory of an individual. A higher capability of germane processing creates the logical understanding process easier, which can reduce the cognitive effort. On the other hand, a lack of germane cognitive load reduces the connectivity associated with the prior knowledge and long-term memory, thereby resulting with high cognitive complexity. Therefore, the ability of managing the types of cognitive load along with the memory capacity of each individual through the comprehension processing is essential to determine the amount of cognition effort to imply the cognitive complexity.

Another factor of determining the comprehension effort is the skill level regarding the programming concepts. The skill level of a user and the extent that memory capacity and cognitive load can be handled are directly connected. In other words, the lack of a proper knowledge to comprehend a source code directs the inability of managing the memory and the cognitive load required for its comprehension. The aptitude level on programming concepts can be observed using two ways. Firstly, the user should aware about the programming language that the source code has been written. The programming language consists with a set of symbolic

representations and instructions, which is different from the users' native language [69]. Therefore, the user should have an aptitude of the programming language to comprehend the content inside the source code. It is evident that the skill level of each programming language is different for each user based on their preferences, and the expertise of a particular programming language has been found to be a factor of determining the cognitive complexity [39]. Moreover, along with the awareness of the programming language, analytical skill of a problem should be required to understand the logic of a source code. In other words, although a user knows a particular programming language, if it cannot be applied to analyze the problem addressed by the source code, the comprehension of the source code cannot be achieved as expected. Therefore, awareness of the programming language together with the problem analytical skill are mandatory to handle the memory and the cognitive load in terms of cognitive complexity.

It has been found that the experience on a particular background plays a vital role of determining the comprehension effort of the same background [56]. The same concept can be applied for the cognitive complexity of a software, so that experience regarding the programming background can be considered as a determinant factor of the cognitive effort. As such, a user with high experience of the programming concepts can easily understand a source code with less effort, which results a less cognitive complexity. On the other hand, a user with less programming experience requires a considerable effort for the comprehension and results a high cognitive complexity. Furthermore, the experience can be related to the concepts of the expertise level, memory capacity and the cognitive load. The experience gradually grows in a particular background along with the duration. Similarly, if a user deals with programming concepts for a considerable duration, the experience of the user can be mentioned as high. It happens due to the development of aptitude level of programming languages and problem analytical skills along with the time. Consequently, a source code which is difficult to understand at the beginning can be easily understood with the duration pass. Hence, it can be concluded that the cognitive complexity is a dynamic factor as well. Through those, the experience can be considered as a controlling mechanism of memory capacity along with the cognitive load. Therefore, the experience of a user should be considered to determine the comprehension effort related to a given source code, thereby to state the level of cognitive complexity.

As the next factor of personal profile to determine the cognitive complexity, age of the user has been considered. It has been found that the age of a user is an indirect parameter for the comprehension effort [70]. In other words, age cannot be considered as a direct parameter to

denote the cognitive effort of a user. On one direction, age of a person is gradually increased with the experience, so it can be stated to achieve a less cognitive complexity based on the experience that the user has gained. But, there can be several situations, where a aged and non-experienced user can understand a source code logic easily. On the other hand, there is a possibility that a novel user with less aged can also comprehend a source code logic easily. Therefore, age cannot be considered as a direct factor of detecting the cognitive complexity. However, age along with the programming-based experience can sometimes use as a merged factor of determining the cognitive complexity.

The other main factor to express the cognitive complexity is the source code. Under the source code factor, the architectural aspect is found to be a direct parameter to determine the cognitive complexity [17], [21], [31]. The architectural aspect refers to the amount of information inside the source code, and it is evident that it effects to the users' cognitive complexity, since a higher amount of information directs to have a high cognitive load to be handled. Therefore, the effort that a user should take for the understanding process goes high, which results in high cognitive complexity. Similarly, a lesser amount of information leads to a less cognitive load to be handled, thereby ensuring a less comprehension effort with a less cognitive complexity. The ways that the architectural aspect can be presented are innumerable, and the best exemplification can be given with the considerable number of research works mentioned in literature. Some of the aspects that the architectural aspect can be expressed are the form of input output parameters [1], [25], BCS [1], [25], [28], variables/attributes [30], operators and operands [27]. Moreover, several research works have emphasized the object-oriented concepts such as inheritance, coupling and cohesion [20], [34], [35] under the architectural aspect. Therefore, the effect of considering the architectural aspect of a source code for the cognitive complexity cannot be ignored.

The spatial capacity of the source code is another parameter of a source code to represent the user cognition effort. It can be defined in two ways as the size and the distance from module call to its implementation [19], [26]. Both are expressed with LOC. A source code with higher LOC tends to complex than a source code with less LOC. Hence, high LOC source code can be difficult to understand which takes high comprehension effort with high cognitive complexity. Also, a less LOC source code tends to be easier to understand, so that it gets a lesser amount of comprehension effort with less cognitive complexity. Nonetheless, there can be certain situations where a high LOC source code directs to an easier comprehension to achieve a less cognitive complexity, while a less LOC source code is difficult to understand with high cognitive

complexity. As such, the size in terms of LOC cannot be taken as a direct parameter of expressing the cognitive complexity. The distance between a module call to its implementation is the other parameter inside the spatial capacity of a source code. This concept is based on the memory capacity and the cognitive load with respect to the distance that a user should navigate when there is a module call. If there is a module call, the source code should refer to its implementation to execute it. The greater the distance between a module call to its implementation implies that the user should acquire a much effort to memorize the logic, whereas the less distance of the same scenario requires a lesser comprehension effort. Along with those situations, there is a possibility of getting a high cognitive complexity for a higher distance, while a less cognitive complexity occurs for a less distance navigation. Therefore, the size and the distance can be considered as the factors effecting for the cognitive complexity of a user under the spatial aspect of a source code.

Furthermore, the programming environment that the source code is generated supports for the amount of cognition effort of an individual. Generally, the source codes for software are implemented with the usage of an IDE, and they are capable of handling the source code more conveniently than a source code without an IDE. IDE is capable of providing the facilities such as proper indentation mechanism, usage of different colors to identify data types over variables/attributes to make the comprehension process much easier. For an instance, a source code written in a notepad is difficult to understand due to the lack of facilities provided by it. Therefore, the selected IDE guides the user to increase the understandability of a source code, which can control the cognitive complexity. Furthermore, the comments are included within the source code to increase the user understandability. The comments are independent from the programming language and the IDE used, which describes the logic of a select code segment in a form that any user can comprehend [71]. The comments further ensure the usability and the maintainability of a source code without restricting its usage only for its developers. On the other hand, a commented source code is considered to achieve a quality source code [49]. Hence, inclusion of the comments inside the source code is essential to assess and control the cognitive complexity. Herein, the supportive programming environment that a source code has been implemented provides an aspect of the source code to express the cognitive complexity.

Although Figure 3 demonstrates the major factors and their sub factors to determine the user comprehension effort of a source code, sub factors listed under the personal and source code aspects cannot be limited. Therefore, it should be noted that the considerable number of sub

factors have been considered for the cognitive complexity determination to explore it using those terms. However, personal profile and source code factors must be considered as the major factors behind the cognitive complexity, since its definition, the comprehension effort of a person acquired for a given source code implies these two factors.

3.3 Procedures of Reducing the Cognitive Complexity of Software

Firstly, the purpose of reducing the cognitive complexity should be discussed. Generally, a source code is said to be complex, if it cannot be understood easily, so that a high amount of effort is required to comprehend its logic, which increases its cognitive complexity. Hence, reducing the cognitive complexity should be performed for a source code to make it as non-complex. Furthermore, reducing the cognitive complexity assists to ensure a high understandability [13]. A high cognitive complexity is occurred due to the inconsistencies associated with personal and source code factors, as they are the major two factors behind the cognitive complexity which have been descried under section 3.2. Further, the inconsistencies occurred due to the misunderstandings of an inaccurately implemented logic also makes the understanding process difficult and effect to the comprehension effort to increase the cognitive complexity [72]. Therefore, high effort utilized with comprehending process of a source code can be defined as the comprehension effort of a source code with correct logic implementation and the comprehension effort of a source code with incorrect logic implementation. It should be noted that this research work is focused to analyze comprehension effort reduction mechanisms for a source code with correct logic implementation, by assuming the source code with correct logic is already available. Accordingly, the inconsistencies of personal and source code factors of a source code with correct logic has to be solved to gain a less comprehension effort, thereby to ensure a less cognitive complexity. It is noteworthy that this work is mainly highlighted personal profile over the source code factor as the literature of the cognitive complexity lacks personal profile consideration. The lack of personal profile of the comprehension process can be indicated as the inconsistencies involving with the persons' working memory and the cognitive load [67]. If the necessary information consists with a persons' cognitive load, comprehension process becomes easy so that the effort utilized for understanding becomes less and achieve a less cognitive complexity. Further, the lack of cognitive load of a user tends the comprehension process difficult, so that a considerable amount of effort has to be taken to understand the logic

behind a source code, which outcomes a high cognitive complexity. As it is already described, the other personal factors such as aptitude level on programming and experience are directed with this cognitive load. Along with the experience growth related to the programming background, the aptitude level on programming is increased. It is because of the cognitive load that a user can handle increases along with the duration. Therefore, cognitive load is the core component that has to be considered with effect to the human comprehension effort under personal profile, as the other personal profiles also direct with the cognitive load. Thereby, reducing the cognitive complexity should be performed in a way that it can handle the cognitive load of a user to gain a less comprehension effort. The procedures that can handle the cognitive load of a user can be viewed in Figure 5.

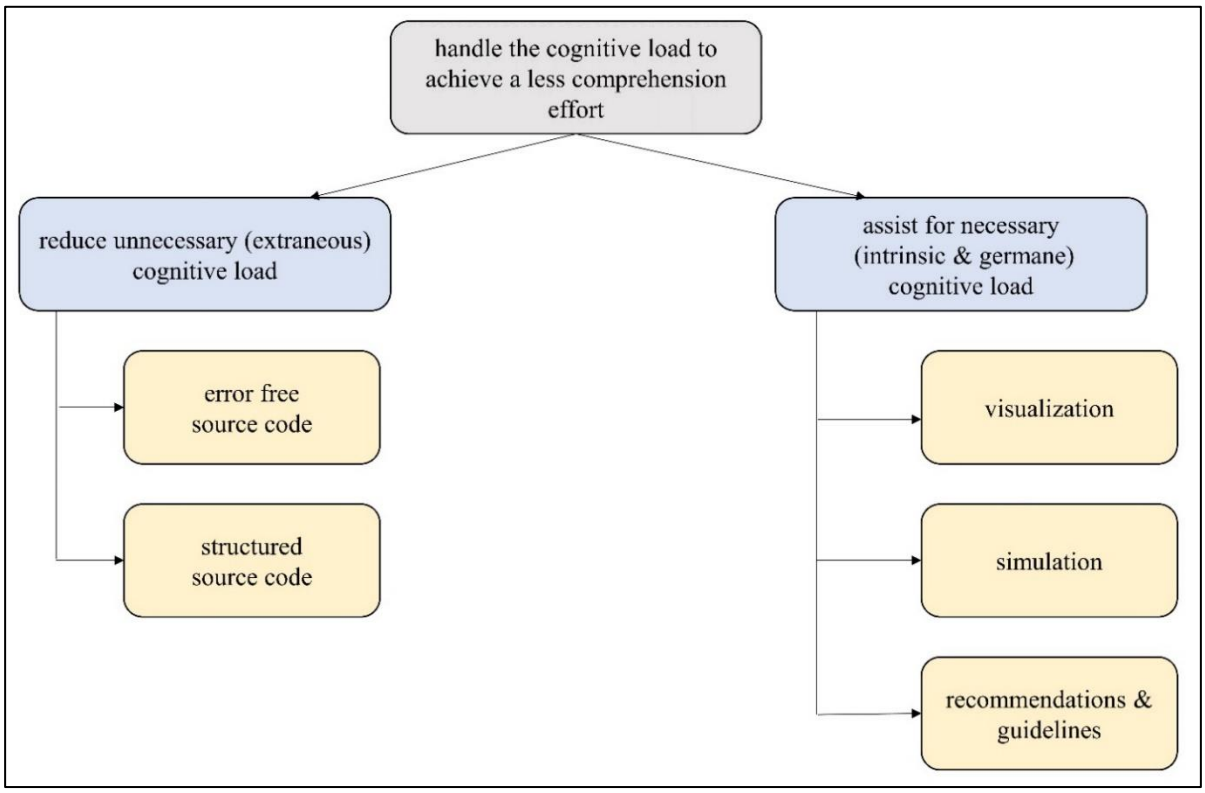


Figure 5. Procedures to Achieve a Less Cognitive Complexity by Handling the Cognitive Load of a User

Based on Figure 5, handling the cognitive load for a less comprehension effort with respect to a source code can be achieved in two ways. The first procedure can be described as reducing unnecessary cognitive load. The memory load dedicated to process unnecessary information is called as extraneous cognitive load, and it has been found that it creates unnecessary complexity over the required processing [68]. Since extraneous memory is not directed with the source code

logic, increment of that type of memory leads to reduce the possibilities of processing the required information which are known as intrinsic and germane cognitive loads. Hence, the possibilities of reducing extraneous cognitive load should be explored. The defects inside the source code negatively influences to the smooth functioning of a source code, which does not relate to the source code logic [73], [74]. Further, a source code with defects has found to be reached to a high cognitive complexity [39] - [41], so that reducing the coding defects should be considered as one procedure to mitigate the extraneous load. Moreover, an unstructured source code consumes a considerable amount of comprehension effort which decreases the code quality and consumes unnecessary cognitive effort [40], [41]. Hence, developing a structured source code can reduce the extraneous cognitive load effecting to the comprehension. These two situations can be further explained as a structured and error free source code is easy to comprehend, and no extra effort is manipulated to solve the issues regarding the errors or the code quality. Furthermore, handling coding defects and code quality issues are some of the processes arrives with the software maintenance [75], [76]. As such, the reduction of cognitive complexity of a source code can be further described as the reduction of extraneous cognitive load associated with handling defects and code quality issues inside the maintenance process in one dimension, apart from the effort utilized for code modifications. Therefore, the reduction of unnecessary cognitive effort by reducing the coding defects and quality issues can create the opportunity to increase the necessary cognitive load, thereby ensuring a high understandability by achieving a less cognitive complexity.

The other way of reducing the comprehension effort is to create certain mechanisms of assisting with necessary cognitive load. Even though the reduction of extraneous cognitive load can be achieved by reducing coding defects and quality issues, the possibilities that a user cannot understand a source code logic still exists due to the reluctance of required information contains with the cognitive load. Therefore, a supportive mechanism has to be supplied to the intrinsic and germane cognitive loads. Accordingly, inclusion of programming independent factors has been studied as the guidance mechanism as the user comprehension effort varies with programming dependent factors, so that defining supportive mechanisms for each type of source code becomes problematic. Therefore, as the first type of assistive mechanism, the visualization of source code logic has been introduced. The visualization of a logic is recognized to increase the understandability as the comprehension level of any user is found to be higher in the diagrammatical representation than it originally appears [51]. The reason behind that situation is

the visualization is being considered as a common way of illustrating the logic of a complex process by numerous applications and it is being independent from any technique [77]. Therefore, the same concept has been taken as an assistive procedure to expose the logic of a source code, so that the user gets an opportunity to refer to the visual representation along with it. Furthermore, this can be taken as the guidance to provide the necessary details of the source code logic, if its logic is found to be complex for a user. Therefore, the visualization of a source code logic can be considered as a guidance mechanism for the cognitive load of users to make the understanding process easier by reducing the cognitive effort.

As the other procedure of assisting with the necessary cognitive load, the simulation of processes can be considered. It has been found that unnecessary misunderstandings cause to increase the cognitive complexity, thereby introducing the simulation techniques can assist the user with the actual processing of a given application [50], [52], [53]. Therefore, simulation techniques can also be applied to a source code to emphasize its actual logic as a method of guidance. As an exemplification, simulation functionalities to expose the source code behavior and the expected outputs can be considered to guide the users' cognitive load with a correct identification, thereby to make the understanding process easier with a less cognitive complexity.

Another type of assisting the cognitive load is to provide the guidelines and recommendations regarding a processing. This can be related with the simulation technique of a process but, being process independent is the determinant aspect behind the guidelines and the recommendations. Providing the guidelines can help the user to smoothly understand the flow of logic in a complex code segment, while the recommendations assist the user to manage a certain process with unnecessary misunderstandings. Therefore, providing both of guidelines and recommendations can be taken as an assistive mechanism of guiding the user in a correct and a smoother way of logical comprehension to achieve a less cognitive effort and a less cognitive complexity.

To conclude, the cognitive complexity should be handled in a way that it effects with the user understandability. A lower understandability directs with a high cognitive complexity, which goes beyond the expectation of any software development process. The lack of the cognitive load is the preliminary reason behind this scenario. Hence, the cognitive load has to be handled to lower the cognition effort and to achieve a less cognitive complexity. It can be performed by reducing unnecessary/extraneous cognitive load and by assisting with intrinsic and germane cognitive loads. Maintaining a defects free and structured source code have been identified as the

procedures that can be followed under the unnecessary cognitive load reduction, while the visualization, simulation and providing recommendations and guidelines have been introduced as the processes of assisting intrinsic and germane cognitive load.

3.4 Methodology of the Design for Reducing the Cognitive Complexity of a Software

The applicability of the cognitive complexity inside the software development and maintenance processes has been highlighted by exploring the procedures of reducing the cognitive complexity as discussed in section 3.3. Then, the methodology of designing these procedures using a computational aid has been analyzed for ease of use. The overview of the proposed system can be viewed in Figure 6.

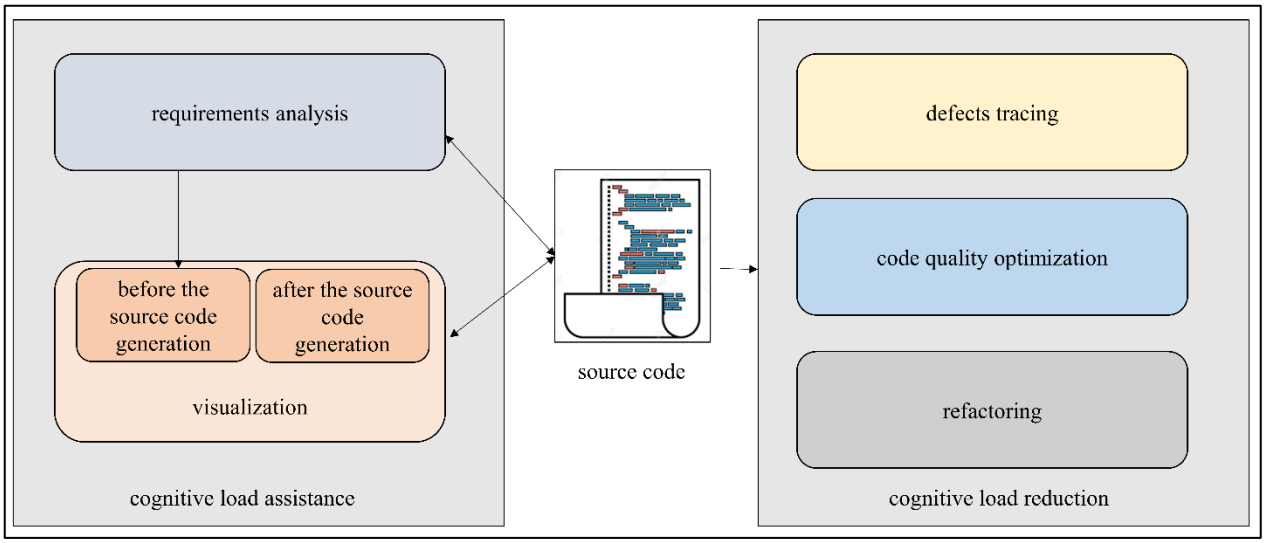


Figure 6. Overview of Proposed System to Reduce Cognitive Complexity

As per Figure 6, these components are responsible to represent the procedures under the cognitive complexity reduction. Cognitive complexity reduction components are comprised with the procedures of assisting necessary cognitive load and reducing unnecessary cognitive load. These procedures have been included as the subcomponents of above two main components and

described as follows. It should be noted that the proposed system is implemented using NetBeans IDE[2].

3.4.1 Assisting for Necessary Cognitive Load: Requirements Analysis Component

A source code of a software is implemented by referring to its requirements. Understanding user requirements for a software is a significant factor to determine its success and its complexity [38]. Therefore, if a source code of a software tends to be complex, the problems related with its logical comprehension can be solved by referring to its requirements, which can reduce the comprehension effort. This can be further explained by the dependency of the comprehension effort with the first piece of information offered to an individual [7]. Therefore, it can be considered as an assistive mechanism for the necessary cognitive load to increase the understandability, and to achieve a less cognitive complexity. Since this research work is emphasized a source code with correct logic implementation for the cognitive complexity determination, the requirements have to be stated as correct, because there is no opportunity that incorrect requirements can lead to a correct logical implementation inside a source code. However, it can be generally experienced that handling the requirements of a software is not easy as expected [78]. Therefore, a new methodology of representing the requirements of a software as an assistive mechanism to comprehend the source code logic has been proposed.

The expectation of this component is to present the requirements in more convenient way that can be used to comprehend the source code logic and for the usage of other phases inside the software development process. The overview of the requirements analysis component is demonstrated through Figure 7.

[2] https://netbeans.apache.org/

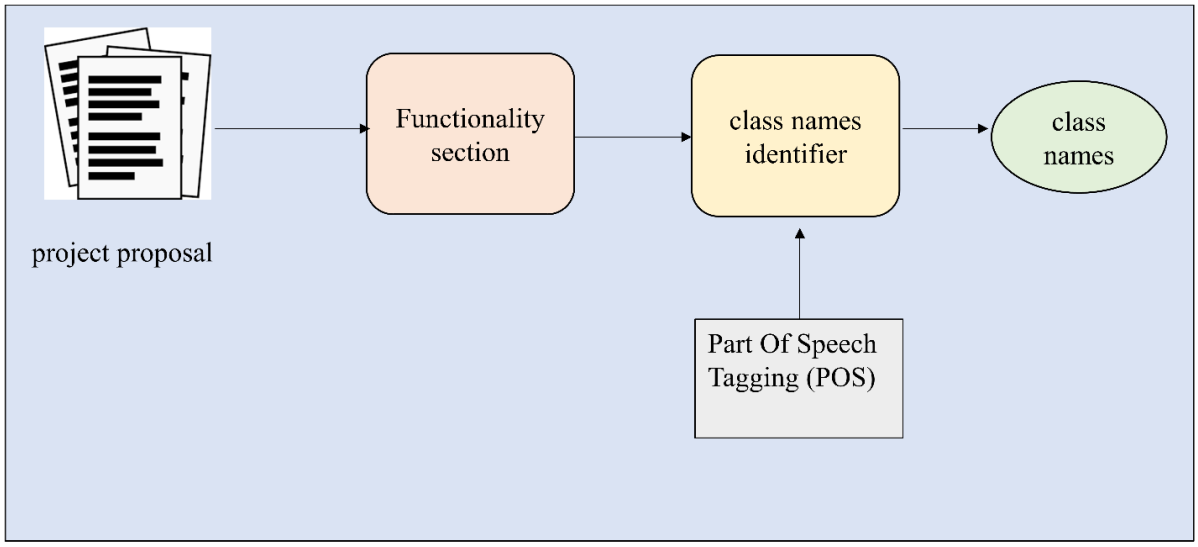


Figure 7. Overview of Requirements Analyzer Component

Usually, requirements of any software are documented in the project proposal document. Hence, the proposal document is taken as the input source of obtaining the requirements. However, reluctance of a standard format of the proposal document has created the problem of locating the exact requirements in a computational background. Therefore, a common format of generating the project proposal document has been suggested as shown in Figure 8.

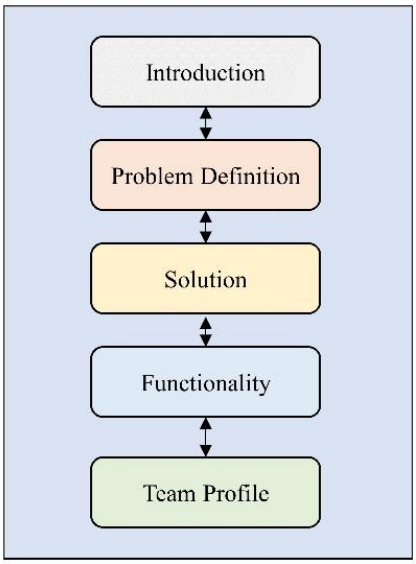


Figure 8. Proposed Project Proposal Document

The content of the proposal document has been divided into five categories namely the Introduction, Problem Definition, Solution, Functionality and Team Profile. The section of Introduction implies the preface of the problem to be addressed by the software which is going to be implemented. A detailed description of the problem is indicated inside the Problem Definition section. The Solution section implies the flow and the mechanism of implementing the solution. The Functionality section indicates the requirements intended by the system to be implemented. Finally, the Team Profile section maintains the information about the software development team which addresses the solution. Therefore, the expectation is to locate the exact requirements through the Functionality section of the proposal document. However, detecting the same content inside the Functionality section as the requirements through proposed component is impractical, and the expectation of using them to comprehend the source code logic and for the development of other software development phases cannot be accomplished. Hence, the necessity of representing the requirements inside the Functionality section as another form has been analyzed. According to the phases followed in Software Development Life Cycle (SDLC), the logical diagram generation is performed once the requirements analysis have been finalized [79]. The source code implementation is performed by referring to the logical diagrams which have been generated earlier. Therefore, the concept of presenting the requirements in a form that they can be used for the diagram generation has been followed. For that scenario, requirement identification as sentences or part of sentences cannot be used. Therefore, identification of system requirements as class names have been explored with the possibility of applying them for the diagram generation process as well. It has been found that the existence of class names as nouns in majority of the scenarios, so that identifying the nouns inside the Functionality section has been studied [80] . To achieve it, Part Of Speech (POS) tagging has been used, which converts each sentence in a document to a list of words and allocates a corresponding tag [81]. The words captured with the tag of "*NN*" have been identified as the nouns inside the Functionality section. Identifying all the nouns inside the specified section would be another problem as it would create unnecessary complexions through the diagram generation process. It is therefore, decided to control the number of class names through the number of times that it has been repeated inside the specification. In other words, the highest frequency class names were suggested to be identified, where the frequency level can be controlled through Python script running inside the class names identifier. The internal logic of class names identifier is addressed through *classIdentification.py* file. The logic of *classIdentification.py* is written in Algorithm 1.

Algorithm 1. *classIdentification.py* to Generate Class Names

Algorithm 1: *classIdentification.py* to generate class names

Input: The text file contains the *functionality* specification
Output: Class names (mostly recurrent nouns with the threshold value given)
1: Let file objects are *f* and *file*
2: *f = open*("//include the path of functionality.txt")
3: Let *lines* represents the content read by *functionality.txt*
4: *lines =f.read()*
5:
6: Let *tokenized* represents each word inside the *lines*
7: *tokenized = nltk.word_tokenize(lines)*
8: **foreach** *word* ϵ *lines* && *if is_noun(pos == 'NN')*
9: Let *counter* is the frequency of each noun
10: *counter = collections.Counter(nouns)*
11: **end foreach**
12: Let *x* is the threshold counter for mostly available nouns
13: *counter.most_common(x)* //gain the most common x nouns
14: Let *listToStr* is the combination of all the class names
15: **foreach** *elm* ϵ *counter.most_common(x)*
16: *listToStr =' '.join(str(elm))* //convert *elm* into a string and join each class name with a white space
17: **end foreach**
18: *file=open*("//include the path of the file to be written", *"w")*
19: file.write(listToStr) //write the content to the *file* object
20: file.close() //close the *file* object

The output of Algorithm 1 is the class names identified through the Functionality section inside the proposal document, which have been repeated more than given frequency level (*x*). Therefore, those class names are known to be the requirements of the software, which can be inputted to the diagram degeneration phase. Hence, obtaining the requirements in a way that they can be useful for the other phases can be verified through this component. Moreover, the effectiveness of these class names as requirements to reduce the cognitive complexity should be then analyzed. Even though, the logical flow of an implemented source code cannot be understood by merely referring to these class names as requirements, the idea of having a high possibility of appearing these class names inside the source code logic can be given to the user through this component. Then, along with these class names as requirements, the logical diagrams have to be generated to assist the user to comprehend the logical flow reduce the cognitive effort. Therefore, class names together with visualization technique can then be

considered as an assistive mechanism to make the comprehension process easy, thereby reducing the cognitive complexity.

3.4.2 Assisting for Necessary Cognitive Load: Logical Diagram Generating Component

The visualization of a source code logic has already been described as a way of supporting to the users' cognitive load to increase the understandability and to ensure a less cognitive complexity. The visualization component of the proposed system (Figure 6) is capable to generate the diagrams of demonstrating the source code logic in two ways. One type of visualization is performed without the source code referring, while the other type of visualization is performed by referring the source code. Generally, UML diagrams are used to demonstrate the internal logic of a source code [82]. Although there is a variety of UML diagrams, class diagram, Entity-Relationship (ER) diagram, sequence diagram, object diagram and component diagram are widely used for the visualization [83]. As the preliminary step, the visualization of logical diagrams without referring the source code has been explored. The reason behind this visualization technique is to automate the order of SDLC, where the diagram generation process is performed by inputting the requirements that have been found. That process is basically achieved manually by using the specifications and project proposal documents. The accuracy of manually drawn diagrams tend to be frequently lesser due to the human errors involving with the transformation process from requirements to visualization and the comprehension issues regarding with diagram logic [84]. Hence, it can cause to generate the diagrams with inaccurate logic, which can create unnecessary misunderstandings. Further, the logical comparison issues between inaccurate diagrams and source code with correct logic can increase the cognitive complexity, which goes beyond the expectation. Therefore, the necessity of deriving a methodology to generate the logical diagrams using the requirements as per the SDLC performs has been implemented as shown in Figure 9.

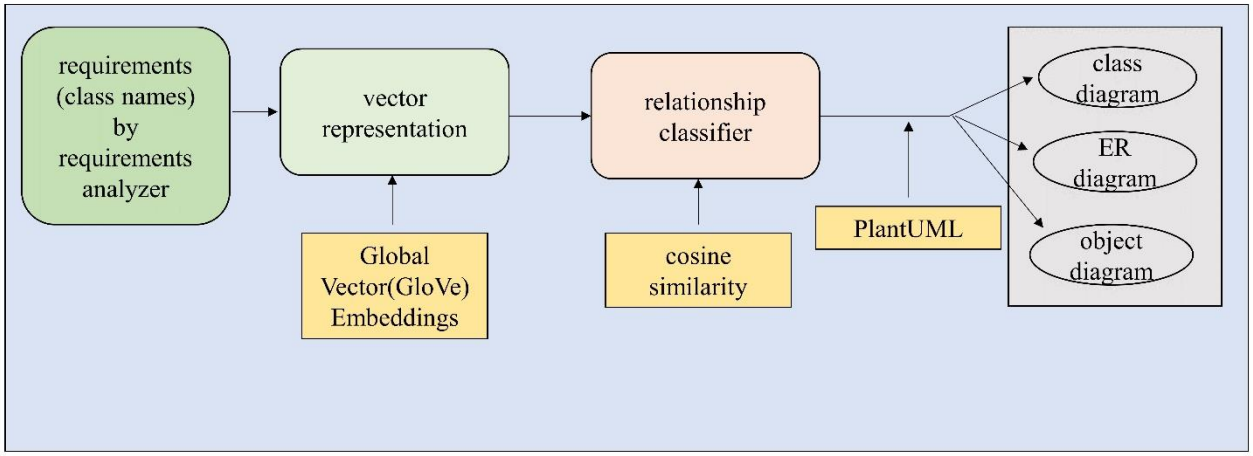


Figure 9. Overview of Visualization Component without using the Source Code

As discussed in requirements analysis component, the diagram generation of this component has also been stated as accurate, due to the impossibility of attaining a source code with correct logic using incorrect logical diagrams. This component can generate the logical diagrams based on the class names outputted by the requirements analysis component as requirements. Further, this component is capable of generating class diagram, ER diagram and object diagram for a software. It is essential to analyze the relationship of class names prior to diagram generation as the logic of diagram modelling is based on the relationships of each entity used inside the diagrams. Therefore, as the first step, Global Vectors for Word Representation (GloVe) [85] techniques has been used to convert each class names into a mathematical representation through a vector. It includes large datasets consisting with large number of tokens and represents with multi-dimensional spaces. The dataset used for the proposed component is glove.6b.100d, which is stored in a text file. It consists of six-billion-word tokens and each token is represented with a vector with hundred dimensions. It is crucial to use high dimensions as it can compare the vectors using multiple parameters and extract the most appropriate relationship between the input class names. TorchText library provided by the Deep Learning library PyTorch[3] has been used to embed with GloVe technology. Each words' vector representation is then compared mathematically using cosine similarity [86]. It represents the cosine angle between each of two vector representations. Two closer vectors denote a higher cosine similarity, and the distant vectors can be represented with a less cosine similarity value. Therefore, class names which are related with each other can be identified through higher cosine similarity, and the class names

[3] https://pytorch.org/

with less logical relations can be identified with less cosine similarity. A logical diagram should consist with different types of relationships which demonstrates the logical flow of its corresponding source code. Therefore, the necessity of finding the correct relationship of each class name pair has raised. To demonstrate the different types of relationships within the diagram generation, following threshold values have been proposed with respect to cosine similarity values as listed in Table 9.

Table 9. Cosine Similarity Values with Relationship Types

Cosine Similarity Level (CS)	*Mapped Relationship Type*
80 <= CS <= 100 (100 similarity implies the same word is compared)	Inheritance
70 <= CS <= 80	Composition
60 <= CS <= 70	Aggregation
25 <= CS <= 60	Association
CS < 25	No relationship

The class names with high similarity levels imply more common behavior, which can be considered as parent-child relationship [87]. Hence, the relationships with highest cosine similarity values are considered under the inheritance category. The composite relationship indicates a parent entity, which owns the child entity with a stronger association. Nevertheless, its behavior is not stronger than inheritance [88]. The aggregation indicates a parent, which maintains a relation with its child with a weaker association level, so that the similarity value scale has to be lesser than the similarity range used for composition [89]. Association relation maintains a weaker connectivity among the classes, thereby the similarity range has to be lesser than inheritance, composition and aggregation [90]. The class names lesser than 25 similarity level have been considered as unnecessary relationships, which do not imply a stronger connection to be applied in the diagrams. PlantUML syntaxes are implemented based on the relationship types to generate each diagram, and inputted to PlantUML library [83]. The different methodologies to construct each of the diagram have been indicated as follows.

3.4.2.1 Class Diagram Generation

The class diagram is generated through this component is capable of demonstrating the possible class names with their relationship types. The attributes and the methods inside each class have not been included in this class diagram generation as this is generated without referring to the corresponding source code. Moreover, the possibilities of having mismatch issues of attributes and methods in both diagram and source code have been analyzed, so that the class diagram generation has been implemented using the class names only. The relationship types use in the class diagram have been uniquely presented using the symbols listed in Table 10.

Table 10. Proposed Symbols for Relationship Types

Relationship	*Notation*
Inheritance	<|--
Composition	*--
Aggregation	o--
Association	--

Each of the relationships identified through Table 9 are then mapped into the corresponding notation using Table 10. Then, they have been initiated as relationship objects and constructed relevant PlantUML statements. Finally, output class diagram has been generated by executing these PlantUML statements. The internal logic used to construct the class diagram through *generateClassDiagram()* can be found in Algorithm 2.

Algorithm 2. *generateClassDiagram()* to Generate Class Diagram

Algorithm 2: *generateClassDiagram()* to generate class diagram
Input: The list of class names obtained through requirements automation **Output**: Class diagram 1: Let *Relationship* is a separate class created to handle pair of class names with their similarity values 2: Let *reList* is the list created from *Relationship* class 3: Call function *assignRelationshipType(reList)* //refer Table 9 for relationship types 4: Let *classA* and *classB* represent the class names in *reList* 5: Let *plantUmlSource* be the object created from *StringBuilder* class 6: *plantUmlSource.append("@startuml\n")* //start writing the *plantUML* syntax 7: **foreach** *r* ϵ *reList* 8: Append *classB*, relevant notation, *classA* and *"/n"* to *plantUmlSource* // refer Table 10 for notations 9: **end foreach** 10: Convert the diagram structure to a string to generate the diagram 11: Store the diagram inside the path specified as a PNG file

Based on Algorithm 2, class names identified through the requirements analyzer have been stored inside the *Relationship* class and the corresponding relation for each pair of class names has been discovered by referring to Table 9. The class name pairs with relationship are then stored inside *classA* and *classB* and appended with the related relationship notation through PlantUML statements by referring to the Table 10. Generated PlantUML statements have been converted into a string to generate the class diagram and saved as a Portable Network Graphics (PNG) file in a specified location for later reference.

3.4.2.2 ER Diagram Generation

In order to generate ER diagram, the type of the relationships and the respective multiplicity levels of each entity should be known. Table 11 summarizes the selected relationship type and the corresponding multiplicity mapped in ER diagram.

Table 11. Relationship Types and ER Multiplicity

Relationship	*ER Multiplicity*
Entity A (whole) aggregation Entity B (part)	A – one or zero B – zero or many (Zero to many)
Entity A (whole) composition Entity B (part)	A – only one B – one or many (One to many)
Entity A (whole) association Entity B (part)	Many to many

Based on the identified relationship and the multiplicity levels, following notations have been used for the visualization process in ER diagram as shown in Table 12.

Table 12. Proposed Notations for Relationship Types

Relationship	*Notation*
Association, Direct Association, Dependency	| o—o|
Composition	|o—|{
Aggregation	|o—o{

The Algorithm 3 specifies the logic used to construct ER diagram for a given software specification using *generateERDiagram()*.

Algorithm 3. *generateERDiagram()* to Generate ER Diagram

Algorithm 3: *generateERDiagram()* to generate ER diagram

Input: The list of class names obtained through requirements automation
Output: ER diagram
1: Let *Relationship* is a separate class created to handle pair of class names with their similarity values
2: Let *reList* is the list created from *Relationship* class
3: Call function *assignRelationshipType(reList)* //refer Table 11 for relations
4: Let *classA* and *classB* represent the class names in *relist*
5: Let *classes* is an object of <String> arraylist
6: **foreach** *r* ϵ *relist*
7: **if** inheritance **and** implement in each relation in *classes* are not existed
8: **if** *r* in *classA* **or** *classB* are not existed and the similarity level < 40
9: Add the *classes* into *classA* and *classB*
10: **end if**
11: **end if**
12: **end foreach**
13: Let *plantUmlSource* be the object created from *StringBuilder* class
14: *plantUmlSource.append("@startuml\n")* //start writing the *plantUML* syntax
15: **for** *c* ϵ *classes*
16: *plantUmlSource.append("entitiy").append(c).append("{}\n")* //draw entities
17: **end for**
18: **foreach** *r* ϵ *reList*
19: append *classB*, relevant notation, *classA* and *"/n"* to *plantUmlSource* // refer Table 12 for ER notations
20: **end foreach**
21: Convert the diagram structure to a string to generate the diagram
22: Store the diagram inside the path specified as a PNG file

Apart from the procedure followed in Algorithm 2 to generate class diagram, following conditions have been used in Algorithm 3. Firstly, the class name pairs with *inheritance* and *implement*, and the relationships lesser than 40 with the similarity level have been removed. The rest of the class names have been added to *classA* and *classB* and then appended with the related notation in Table 12. The ER diagram is constructed by executing PlantUML statements, which have been generated by appending *classA*, identified notation and with *classB*, and the possibility of saving it to a PNG file is also provided.

3.4.2.3 Object Diagram Generation

The object diagram generated by this component illustrates the objects of the classes which have been identified through the requirements analyzer. Nevertheless, the inheritance relationship has not been considered, since this diagram displays only the view of the objects of the source code. The logical behavior of visualizing the object diagram by *generateObjectDiagram*() can be viewed in Algorithm 4.

Algorithm 4. *generateObjectDiagram()* to Generate Object Diagram

Algorithm 4: *generateObjectDiagram()* to generate object diagram

Input: The list of class names obtained through requirements automation
Output: Object diagram
1: Let *Relationship* is a separate class created to handle pair of class names with their similarity values
2: Let *reList* is the list created from *Relationship* class
3: call function *assignRelationshipType(reList)* //refer Table 9 for relations
4: Let *classA* and *classB* represent the class names in *relist*
5: Let *classes* is an object of <String> arraylist
6: Let *inheriClasses* be a String hash to gain inherited class pair names
7: **foreach** *r* ϵ *relist*
8: **if** *r* in *classA* **and/or** *classB*
9: check the existence of inheritance and implements in r
10: Add the *classes* into *classA* and *classB*
11: **end if**
12: **end foreach**
13: **for** *s* in *classes*
14: **if** *inheriClasses* contains a class pair
15: Remove the super class
16: **end if**
17: **end for**
18: Let *plantUmlSource* be the object created from *StringBuilder* class
19: *plantUmlSource.append("@startuml\n")* //start writing the *plantUML* syntax
20: **for** *c* ϵ *classes*
14: *plantUmlSource.append("object").append(c).append("{}\n")* //draw entities
15: **end for**
16: **foreach** *r* ϵ *reList*
17: append *classB*, "- -", *classA* and *"/n"* to *plantUmlSource*
18: **end foreach**
19: Convert the diagram structure to a string to generate the diagram
20: Store the diagram inside the path specified as a PNG file

To generate the object diagram, class names of each pair is added to two different classes called as *classA* and *classB*. The object name identification is commenced while identifying the relationship types of each class pair. The class pairs identified with *inheritance* and *implements* have been inserted to another class called *inheriClasses*, while removing their super class names. Then, all the other relationships have been encoded with PlantUML statements to be visualized with "- -" from one object to another object.

Therefore, the user is given the opportunity to refer to the class diagrams, ER diagrams and object diagrams generated by this component in a problematic scenario of understanding the logic of a source code. Thereby, the diagrams are expected to provide the details of the classes and their relationship types, so that users can compare the connectivity among those inside the source code. Therefore, they can be stated to assist for user comprehension which cause to reduce the cognitive complexity. Nevertheless, the exact flow of the source code cannot be attained through these diagrams as they are unable to illustrate the logical flow as in control flow diagrams, activity diagrams or sequence diagrams. Even though comprehending the logic of a source code includes identifying the major components like classes and their relationship types, the cognitive complexity is mainly focus for the comprehension of the source codes' logical flow. Herein, the necessity of having UML diagrams which are generated by referring to the source code has been arrived as another assistive mechanism for users' cognitive load to achieve a less cognitive complexity. As a solution, the visualization component of the proposed component has been included with another sub visualization component which can generate UML diagrams by using the source code with correct logic.

The component of visualizing UML diagrams is capable of modelling the sequence diagram and the class diagram to demonstrate the logical behavior of a source code. Surprisingly, some recent automated tools can be found to generate the UML diagrams, but majority of them have to be drawn by user after comprehending the logic [84]. However, the possibility of generating sequence and class diagrams without user intervention is the major significance and the advantage of the proposed component. The overview of the diagram generator by using the source code can be observed in Figure 10.

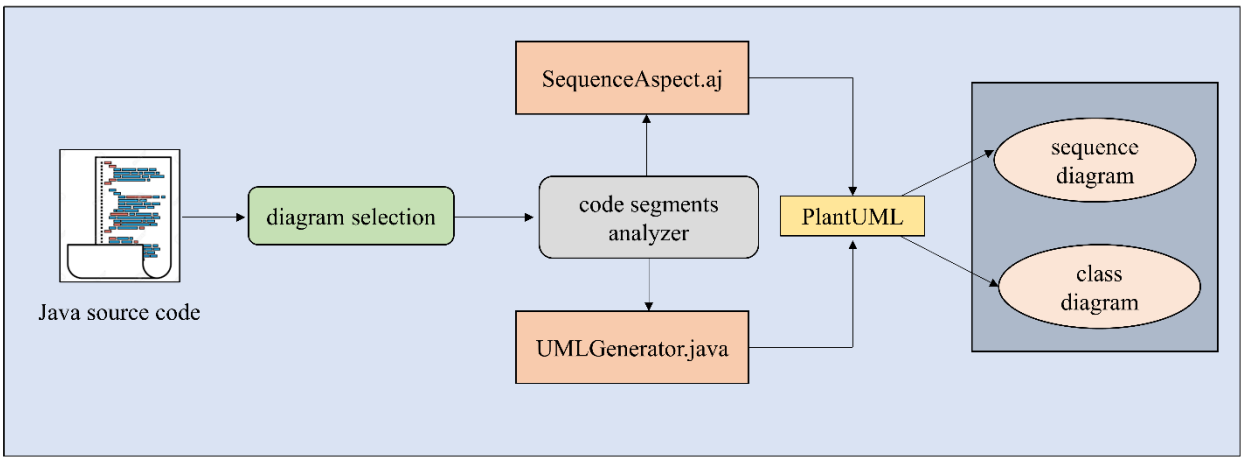


Figure 10. Overview of Visualization Component using the Source Code

According to Figure 10, Java source code/s in which the diagrams have to be modelled is the input for this component. The user is given the opportunity to select the appropriate diagram to be generated by the component. Once the diagram selection has been done, the appropriate code segments related to each diagram visualization are going to be selected by the code analyzer. It uses two source files, where *SequenceAspect.aj* is used for the sequence diagram generation and *UMLGenerator.java* is used for class diagram generation. After necessary information has been processed by the corresponding source file, PlantUML statements are generated according to the syntax required by each diagram. Then, the execution of these statements outputs the required diagram and possibility of saving it as a PNG image file has also been provided.

In the scenario of generating the sequence diagram, the inputted java source files should be configured and converted to an AspectJ project since *SequenceAspect.aj* file is responsible for sequence diagram generation. The logic inside *SequenceAspect.aj* has been denoted in Algorithm 5.

Algorithm 5. *SequenceAspect.aj* to Generate Sequence Diagram

Algorithm 5: *SequenceAspect.aj* to generate sequence diagram

Input: Java source codes implemented for a software
Output: Sequence diagram
1: Execute all java files in the current directory
2: Obtain the project location using current join point
3: Let *source* and *target* represent the source class name and the target class name respectively
4: **if** the class name in current join point == null
5: set *source* into current class name
6: **else**
7: set *source* into main
8: Let *methodSign*, *returnType*, *parameters*, *methodName* and *message* represent the element of a message structure
9: Set *methodSign* by getting the signature of current join point
10: Set *returnType* by splitting *methodSign* with (" ") [0]
11: Set *parameters* by getting substrings of *methodSign* enclosed with "(" and ")"
12: Set *methodName* by getting the name of *methodSign*
13: **if** *parameters* != null
14: Set *message* using "(" + *parameters* + ")" + ":" + *returnType*
15: Let *messages* represents the whole message structure
15: Set *messages* by appending *source, target* and *message*
16: **end if**
17: Let *source*, *target*, *messageString* represents the sections of *messages*
18: **for** *message* in *messages*
19: Set *source* to *message[0]*
20: Set *target* to *message[1]*
21: Set *messageString to message[2]*
21: Let *messageStr* be the message string passed as UML string
23: Set *messageStr w*ith *source* + "->" + *target* + ":" + *messageString*
24: **end for**
25: Create *builder* object from *StringBuilder* class with *"@startuml\n"*
26: **foreach** *umlmessage* in *messageStr*
27: Append *umlmessage* with *"\n"* to *builder*
28: **end foreach**
29: Convert the diagram into PNG image and save it inside the current project folder

The sequence diagram requires necessary class file names and message callings passed invoked them. Those contents have been categorized by calling *SequenceAspect.aj*, which has been implemented through Algorithm 5 and execute relative PlantUML statements to generate the corresponding sequence diagram.

On the other hand, the class diagram generation has been performed by *UMLGenerator.java* in which its logic has been implemented through Algorithm 6.

Algorithm 6. *UMLGenerator.java* to generate the class diagram

Algorithm 6: *UMLGenerator.java* to generate the class diagram
Input: Java source codes implemented for a software **Output**: Class diagram 1: Let *umlRelations*, *umlAssociations*, *umlClasses* are arraylist objects to represent relationships, association types and class names 2: Create *builder* object from *StringBuilder* class with *"@startuml\n"* 3: Let *umlClasses* in an arraylist type object to represent all the class names provided 4: Add all the class names into *umlClasses* 5: **foreach** *umlclass* in *umlClasses* 6: Gain all class names and interface names into *builder* 7: Gain all constructor declarations into *builder* 8: Gain all field names into *builder* 9: Gain all the method names into *builder* 10: Remove duplicate associations in *umlclass* 11: Remove duplicate relationships in *umlclass* 12: Add relationship types and association types to *builder* 13: **end foreach** 14: Close *builder* by appending "\n\n" 15: Convert the diagram into PNG image and save it inside the current project folder

It is noteworthy that Algorithm 6 presents the abstract form of logic followed by *UMLGenerator.java* to generate the class diagram. To describe it furthermore, this algorithm is capable of obtaining the class names and interface names and to return PlantUML notations for them in which their fields are *public* and *private*. It can return relationships of classes while checking for duplicates of relationships and associations. It has the capability of getting the class and interface by name, and to check one to many relationship (1 → M) existence for a given class. The class diagram generation is also comprised with avoiding the method duplication in child classes if the method is already available in its parent class or interface. Further, checking whether a method is either a setter or a getter method, existence of getter or setter methods for a given field can also be performed prior to class diagram generation.

Thereby, the assurance of visualizing the source code logic can be stated through sequence and class diagrams, as both of diagrams have been implemented by analyzing the logical flow inside them. Along with those, class diagram, ER diagram and object diagram generated by other visualization component can provide an overview of the source code logic as they are generated without the source code. Therefore, by giving the opportunity for users to refer UML diagrams generated in two ways, their ability to understand the logic of a source code can be developed. Accordingly, the visualization component can be considered as an aid to expand the necessary cognitive load of a user, which tends to reduce the effort allocated for understanding process. However, current tools supported for UML diagrams generation within NetBeans IDE cannot be ignored. The Visual Paradigm[4]and EasyUML [91] are commonly used as UML diagram generation techniques which can be integrated with NetBeans IDE. Visual Paradigm tool supports for a platform for user to model the relative UML diagram by comprehending the source code logic, which is not the exact requirement to reduce the cognitive complexity. Because it does not act as an assistive mechanism for the comprehension process, as it should be generated by users after the logical comprehension. Surprisingly, EasyUML supports for diagram generation based on the source code without the user intervention as the proposed component does. However, it can be achieved with a proper integration and installation of EasyUML inside NetBeans IDE. The proposed component solves the integration and installation issues inside NetBeans IDE as it has already been implemented through NetBeans. However, the limitation of the types of UML diagrams supported by the component to class diagram, ER diagram, object diagram and sequence diagram exists over the types supported by current visualization techniques. Therefore, the expectation is to achieve a considerable comprehension level by the proposed component over the current visualization mechanisms with respect to the supporting UML diagrams and to achieve a less cognitive complexity.

[4] https://www.visual-paradigm.com/

3.4.3 Reducing Unnecessary Cognitive Load: Defects Tracing Component

A bug or a coding defect is an unusual or unexpected event, which occurs the entire system in a non-functional state [92], [93], and it has been found that the defects can cause to increase the cognitive effort of users by utilizing unnecessary cognitive load which have been assigned for the actual source code logic comprehension [40], [41], [94]. Therefore, detection of coding defects has been introduced as a component of the proposed system under the expectation of reducing the comprehension effort of users. It is evident that the human comprehension is essential for handling the coding defects. It includes identifying the type of the bug and verifying the necessary correction to be applied. If the coding defect is not understood by a user, the effort utilized to solve that issue will be high, which demands for a higher cognitive complexity. There are numerous bug trackers available to trace the coding defects inside a given source code, but the success is limited with the number of bugs that each bug tracker is capable of identifying [95]. Also, some bugs are runtime defects which the bugs trackers cannot identify in the debugging time. The expectation is to trace the bugs which are not supported to the bug tracing plugins available in NetBeans IDE. In the proposed system, the plugin of FindBugs bug tracker has been installed [96], so that the bugs which are not identified by it are suggested to be handled by the proposed component. The overview of the bug tracker component can be viewed in Figure 11.

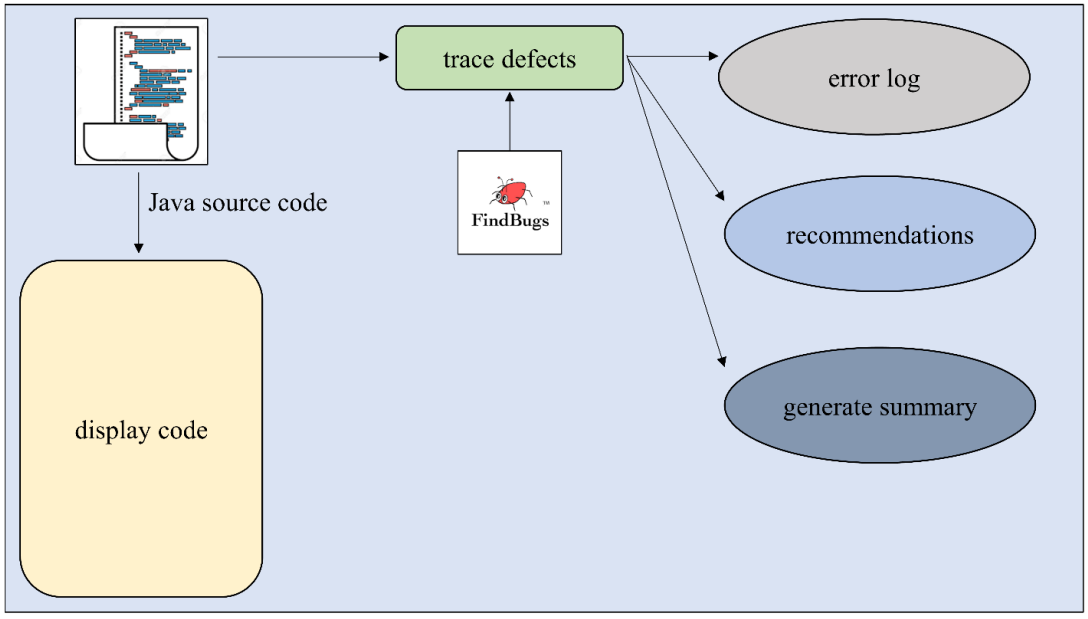


Figure 11. Overview of Defects Tracing Component

The source code is the main input for defects tracing component. Since the proposed system is a java-based system, it should be noted that the defects tracing component capable of handling only java source codes. In order to gain the source code for defects detection, the option of file browsing has been implemented as the first functionality. Then, the uploaded source code is displayed with the corresponding LOC value to track the source code segments easily by its location. Tracing the possible defects is the next functionality of the component. The types of coding defects that the component is capable to handle are listed in Table 13.

Table 13. Coding Defects Handled by Defects Tracing Component

Identified Coding Defect	*Description*
Division by zero	The denominator of any mathematical division is zero, which derives infinity as the answer
Multiple return statements	Multiple return statements in a single method
Array storing error	Store incompatible values inside an array (the data types of the array and the values do not match)
Unsupported operations	Add new values to a list, which is already assigned with values
Illegal states error	.next(), .set() and .remove() methods are called in incorrect order as follows next(), remove(), set() set(), next(), remove() set(), remove(), next() remove(), next(), set() remove(), set(), next() set(), next() remove(), next() remove(), set() set(), remove()
Number formatting errors	Incompatible values are passed to the parse methods
Illegal arguments	Null parameter is passed as a parameter in any method
Illegal thread states	Attempt to start or try to call sleep method to the same thread multiple times
Illegal monitor state	Call thread.wait() method many times for the same method
Unsupported cloning	.clone() method is used without implementing the Cloneable interface
Class casting error	Create an instance of the parent class
Illegal thread run method	thread.run() method is called to the same thread multiple times

Once a possible match detected, it will be prompted as a defect. It includes displaying the identified defect type, its location and the recommendations to fix it. Each recommendation also includes the error description, the location to track the defect and the procedure to correct it, so that the users' unawareness of the defect will not be an issue. Since the recommendation procedure is provided in a more convenient and understandable way, comprehension issues of the error messages generated by existing bug trackers have been solved [95], [97]. Finally, the summary of the coding defects can be received, which includes the error count, LOC value, percentage of error occurrence based on LOC and a description on errors.

The significance of this component lies with the recommendation procedure of fixing a coding defect. It has been already discussed that providing recommendations and guidelines cause to assist the user with related logic, so that it can assist to reduce the cognitive complexity. The recommendation procedure of this component is comprised with a detailed description which can be understood by the users. Therefore, the effort required by the user to handle a coding defect will be minimized not be fallen into the extraneous cognitive effort, as the necessary information is supplied by the component. Thereby, the assurance of having a proper defects tracing and guidance component under software development and maintenance phases of software can be validated to achieve a lesser cognitive complexity.

3.4.4 Reducing Unnecessary Cognitive Load: Code Quality Optimization Component

The next component introduced under extraneous cognitive load reduction and to increase the necessary cognitive load is the optimization of code quality issues. The quality of a source code is found to be a parameter of determining its complexity as a high-quality source code directs to a less complexity and a less quality source code directs to a high complexity [98]. The source code quality is measured with multiple dimensions, and the ease of understanding is one factor among them. The understandability of a source code can be expressed through cognitive complexity [99]. Hence, the source code quality can be considered as a parameter to control the cognitive complexity. Moreover, maintaining the quality of source code is another determinant process behind any software development process, as it assists to increase the usability and the maintainability [75], [100], [101]. Accordingly, the applicability of cognitive complexity inside

the software maintenance to verify the source code quality can be stated again. By considering all these scenarios, a quality source code can be denoted as easily comprehensive than a less quality source code. Furthermore, source code quality is reduced with respective to its improper structure. The reason to have an improper structure inside a source code is the appearance of code quality smells, which creates the source code difficult to understand. In other words, a source code with bad designs is difficult to focus and comprehend, so that users should have high effort on the comprehension process. Hence, it is essential to eliminate the code quality smells inside the source code, which in turns output a source code with lesser cognitive complexity. Eliminating a code quality smell includes identifying the issue and recognizing the steps of correcting it, which is not related with the source code logic. Therefore, code quality optimization takes a considerable amount of unnecessary cognitive load which reduces the capacity of necessary cognitive load required for the understanding process of source code logic. As a solution, the proposed component has been introduced to assist the user with optimizing the code quality smells, so that it helps to reduce the unnecessary cognitive load that user should utilize without a quality optimizer. Thereby, it helps to give more priority for the necessary cognitive load along with the comprehension process of the source code, so that the source code would be easier to understand, which can gain a less cognitive complexity.

There are numerous applications built to detect the code quality smells and fix them to ensure the source code quality. SonarQube and SonarLint are some of the recent services provided to validate the source code quality by detecting code smells and defects, and to compute the complexity using existing software complexity metrics. However, the necessity of hosting the source codes of these services is existed. Hence, it is expected to achieve code smells detection without hosting the source codes through the proposed component. The component is constructed to distinguish Java code smells[5] and to provide recommendations to fix them. Therefore, it can be treated as an assistant mechanism to fix unknown quality issues and to reduce the users' unnecessary cognitive load and effort. As in the defects tracing component, the recommendations have been supplied in detail to easily understand by each user as an assistance procedure to reduce the unnecessary cognitive load. Moreover, the feature of auto fixing of several quality issues has been introduced within the component to assist the user, which is reflected as another assistance mechanism to reduce the extraneous cognitive load. Therefore, it can be specified that the code quality optimization component is capable to assists with the users' necessary cognitive

[5] https://rules.sonarsource.com/java/type/Code%20Smell/

load while reducing the unnecessary cognitive load to reduce the cognitive complexity of source code. The methodology of processing the code quality issues optimization can be viewed in Figure 12.

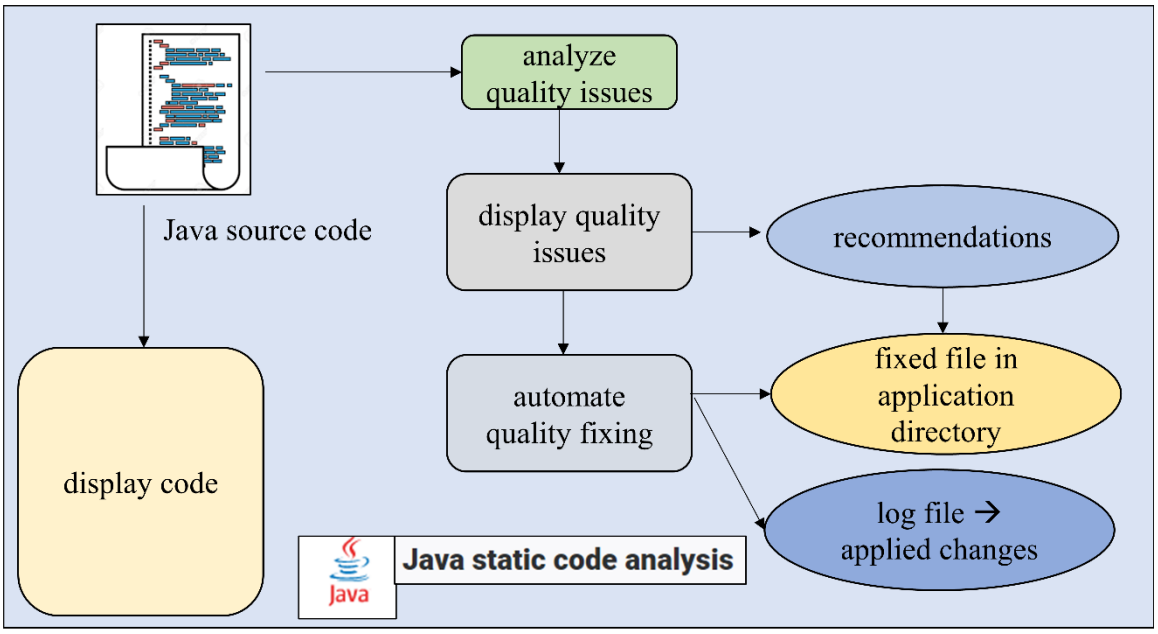


Figure 12. Overview of Code Quality Optimization Component

The java source code that required to be analyzed to detect the code quality smells is the input for this component. A copy of the source code is maintained by the component to navigate to each line and remove comments and excessive white spaces. Then, the component refers to each line again and identify the required syntaxes and elements. Java keywords (reserved key words), identifiers, numbers, operators, escape characters, variables, classes and methods are denoted as the code elements inside a source code. By analyzing the how Java key words have been used inside the source code, the rest of identifiers, numbers, operators, escape characters, variables, classes and methods are stored inside different collections for future reference. Through those, the source code is verified against the code smells handled by the component. The code quality smells handled by the proposed component have been listed in Table 14.

Table 14. Code Smells Detected by Code Quality Optimization Component

Code Smell	*Description*
Unused import detection	Import statements which have not been used inside the source code. Can automatically fix by the component.
Redundant modifiers usage in interfaces detection	Interface access modifier is used again for the declarations inside the interface. Can automatically fix by the component.
Invalid usage of generics in constructor detection	Generics are declared in constructors when the declaration also expressed as generic. To address this, the diamond operator can be used. Can automatically fix by the component.
Ternary operator detection	Usage of ternary operators can be replaced with if-else conditions
Nested ternary operator detection	Usage of nested ternary operators has to be minimized
Invalid modifier declaration order detection	Modifier declaration has to be performed in the following order. public, protected, private, abstract, static, final, transient, volatile, synchronized, native, strictfp

Based on the above code smells, the component is coded to identify the matching patterns by analyzing the syntax and the code elements of the inputted source code. If a particular source code segment is observed to be aligned with a quality issue handled by the component, it will be emphasized by providing recommendations to fix it correctly. Further, the auto fixing features are also be activated for a certain number of quality issues, so that the user should not have any processing with the extraneous cognitive load, which can be considered as the best scenario of demonstrating the cognitive complexity reduction. A fixed file after correcting code quality issues either by following the recommendations given by the component or by automated feature is saved inside the same project directory. It consists of the modified source code after applying the modifications by user. Moreover, a log file containing the changes which have been applied can also be observed inside the same directory. Therefore, this component can also be stated as a

mechanism of reducing the cognitive complexity concept inside the software development and maintenance processes, and to assist to reduce unnecessary cognitive load of users

3.4.5 Reducing Unnecessary Cognitive Load: Refactoring Component

As it is already described in the code quality optimization component, it is evident that a structured source code can attain a less cognitive complexity while an unstructured source code leads to a high cognitive complexity. Currently, the refactoring techniques are being used to make an unstructured source code into a structured source code. Therefore, a component of handling the certain number of refactoring techniques has been introduced in the expectation of reducing the cognitive complexity. This component is aligned with the code quality optimization component as detecting code smells and refactoring are related concepts. Code quality smells are the situations of implying the bad design of a source code and refactoring techniques can be applied to enhance the design of a source code by altering the behavior of it without effecting to its outputs [102]. Since the source code is altered into a structured source code after applying the refactoring techniques, the users' unnecessary cognitive load utilize with the unstructured source code will be mitigated, so that the reduction of the cognitive effort and the cognitive complexity can be mentioned. Therefore, the assistance of the refactoring techniques to increase the source code structure and its quality can be considered to optimize the cognitive complexity associated with it. The procedure of the refactoring automation can be observed in Figure 13.

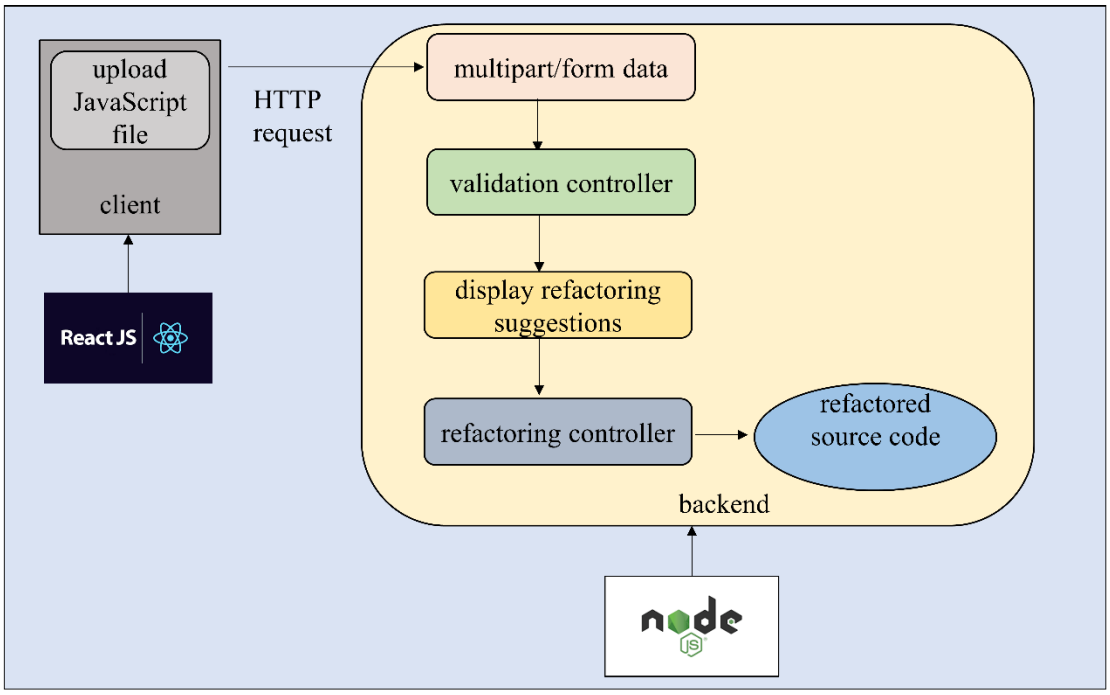


Figure 13. Overview of Refactoring Component

According to the Figure 13, the functionality of the refactoring process has been implemented as a client process and a backend process, and the purpose of implementing it is to provide the accessibility from outside through a public locator. The client and backend processes are communicated with Hypertext Transfer Protocol (HTTP) requests, and all the HTTP requests are handled by Axios library. The client and the backend processes have been implemented through ReactJS and NodeJS respectively in terms of the of performance and synchronization made through several client queries. The user is supposed to act as the client process. The user should upload the unstructured source code as a JavaScript file to be refactored by the component. Therefore, the unstructured source code is acting as the input for this component. Multer library is used for the file uploading process due to its high flexibility and the efficiency with multipart/form data. Accordingly, the uploaded source file is sent as a multipart/form data to the backend by the client process. The Multer middleware converts multipart/form data and attaches it to the HTTP request for easy access. Then, the structure of the uploaded source code is verified against the refactoring techniques handled by the component. The refactoring techniques supported by the component are listed in Table 15.

Table 15. Refactoring Techniques Supported by Refactoring Component

Refactoring Technique	*Description*
Unused variables refactoring	variables that have been declared and initialized, but not used inside the processing should be avoided
Unused methods refactoring	methods which have not been called or used for the processing should be avoided
Empty methods refactoring	methods in which their method body is empty should be avoided
Unreachable code blocks refactoring	code blocks which cannot be accessed (code segments appeared after *return* statement) should be avoided
Empty lines refactoring	unnecessary empty lines should be avoided
Unnecessary comments refactoring	unnecessary comments, which are not relevant to source code logic should be avoided

In order to validate the refactoring techniques listed in Table 15, Pure JavaScript-built methods and Putout opensource libraries are used. Firstly, the file upload controller saves the file into the disk space. In order to detect the possible refactoring technique, a validation controller has been designed for each refactoring technique, which returns a boolean value to the client process. If the boolean value returned from the validation controller is true, the detection of that corresponding refactoring technique can be stated. Therefore, boolean values obtained from the validation process are responsible for displaying all the related refactoring techniques. Then, the user is given the opportunity to select the appropriate techniques to be applied. Once the user has selected the refactoring techniques, another set of controllers have been defined to apply those refactoring techniques. Moreover, the opportunity to view the difference between the unstructured and the corresponding refactored and structured source code has been implemented through a split view. After the refactoring process is completed, the ability of downloading the structured source code has also been provided.

Although the refactoring mechanism is automated in most of the IDEs, the facility of recommending which refactoring technique is used inside a particular code segment cannot be observed. However, the proposed system component is capable of demonstrating the possible refactoring techniques along with the possibility of applying them computationally without a user intervention. This can be treated as a vital achievement, since the user's comprehension effort for the refactoring technique identification and application is reduced by this component to lessen the

unnecessary cognitive load of users and their cognitive efforts, thereby ensuring a reduction of the cognitive complexity of source code.

3.5 Introducing a Meaningful Cognitive Complexity Metric

One section of the proposed system has addressed the applicability of using the cognitive complexity concept inside the processes involving with software development and maintenance processes and to achieve a less cognitive complexity. However, the usage of the cognitive complexity concept should be performed in more convenient way. An expression in a form of metric is easy to use and compare, so that presenting the expression of cognitive complexity in a form of metric has been raised. Although there are numerous cognitive complexity metrics introduced by previous research works, the problems of their standardization and validity are existed [53]. Therefore, the possibility of introducing a new cognitive complexity metric has been analyzed, which can solve the drawbacks of previous research works. As the preliminary step, the drawbacks of cognitive weights in previous works were thought to be addressed. The major drawback of previous cognitive weights is the assumption-based weightage assignment, which has not been performed through a validated framework [33], [35]. Accordingly, the problem of using them for the cognitive complexity computation in real applications has been raised. Therefore, assigning cognitive weights through a valid framework has been performed. Since previous cognitive weights have been allocated for different source code categories under the architectural aspect of a source code, selecting a proper source code category was the next task. It has been observed that the usage of BCS in majority of research works for the cognitive weight allocation [1], [25], [28]. Furthermore, BCS define the fundamental and essential flow mechanisms, which builds the logical architecture of a given source code [4]. They are basically categorized into sequence, selection and iteration, in which the sequence structure has simple statements with no control statements execution, selection has one or more decision statements execution and iteration statements with the combination of both sequence and selection statements. Therefore, a methodology was designed to assign valid cognitive weightages for preliminary BCS categories.

3.5.1 Cognitive Weightage Assignment for Preliminary BCS through a Valid Framework

The expectation of this weightage allocation mechanism is to assign valid cognitive weights for *if-else*, *switch-case* statements under section criteria and *for* loops, *while* loops under iteration criteria. Furthermore, the cognitive weights for nested control structures have also been considered, since the nested control structures are found to increase the cognitive complexity of a source code [31]. Hence, the cognitive weights for *nested for* and *nested while* loops have also been considered for the weightage assignment through the same methodology. The procedure of assigning the cognitive weights for these BCS performed through an experimental background as followed in [34]. As the experimental background, a questionnaire has been arranged to analyze the comprehension level taken for above mentioned BCS categories among a selected user group. The problem of confining the cognitive weights generated through this questionnaire to the selected user group is still existed but, it is expected to analyze their applicability for entire user population using the statistical operations. The entire research work is aimed for the comprehension of a source code in which its logic is correctly implemented. Therefore, the comprehension effort allocated to gain the correct logic has to be considered for the cognitive complexity determination. The same concept has been mapped into the scenario of questionnaire, so that the comprehension effort of the correct logic can be implied using the marks that has been obtained. Further, the duration to comprehend a given logic can imply the level of the comprehension as a higher duration taken to understand a logic expresses a high cognitive complexity while a lesser duration taken to understand a logic implies a less cognitive complexity [34]. Accordingly, marks obtained from the questionnaire and duration taken to complete the questionnaire have been used as the parameters to assess the comprehensibility.

The questionnaire is consisted with five questions representing each BCS category. The questions under similar categories have been selected as same to make the comparison process easier with other controllers, and it has been included in Appendix D. As the target user group, a computing related BSc final year university student group was considered based on higher expertise and aptitude level in computing field than the other students. The students were tested against their coding expertise levels through another questionnaire as an aptitude test, and the students who scored more than 60% have been selected to the user group. The aptitude test given to selects users has been attached in Appendix E. As such, 500 students were selected as the target user

group, and the BCS questionnaire was conducted among them through a Moodle framework. The purpose of using a Moodle environment is to gain the marks and the duration efficiently, so that the inconveniences occurred with respective to manual monitoring will be minimized. It should be noted that each of BCS category has been tested by keeping one week duration gap to avoid the experience of dealing with the same question again, as experience effects to duration and marks when dealing with the same question. The recorded data has been statistically analyzed through Statistical Package for the Social Sciences (SPSS) software [103]. Mean of the recorded data was used to perform the statistical analysis as it can represent the entire dataset. Hence, mean time and mean marks were compared to observe the understandability of BCS. Accordingly, the assignment of cognitive weightages has been performed based on the mean values obtained for marks and duration. Then, the applicability of using the same weightages for generic users has been tested statistically.

Although the validity issue of cognitive weights has been solved using this mechanism, the problem of not addressing the entire architectural aspect is existed as BCS is only a single parameter of it. As it has already explained, factors of the architectural aspect are unlimited so that expressing the comprehension level associated with them to represent it is unachievable. Therefore, the necessity of expressing the cognitive weights in more convenient manner has been studied. Since this research work is emphasized the involvement of personal profile over the source code factor of the cognitive complexity determination, the inclusion of personal profile along with the cognitive weightage concept has been explored. Therefore, a cognitive weightage mechanism which highlights personal profile associated with the logical comprehension has been introduced.

3.5.2 A New Cognitive Weightage Assignment Emphasized on Personal Profile

It is expected to introduce a cognitive weight which highlights personal profile to denote the comprehension level of each user for a given source code using a numerical value. Since the comprehension levels of each individual are varying, the effectiveness of the parameters of personal profile should be varied as well. Therefore, the proposed cognitive weight should be varied from each user, which can emphasize the subjectivity of the cognitive complexity. As

such, the proposed cognitive weight has been generated as a predictive measurement by considering several personal and source code base factors which are interconnected with each other in terms of user comprehension level. It is noteworthy that the number of factors of taken from personal profile for validation are significantly higher than source code factors due to the emphasis given to the personal base throughout this work. The procedure followed to generate the person based cognitive weight can be observed in Figure 14.

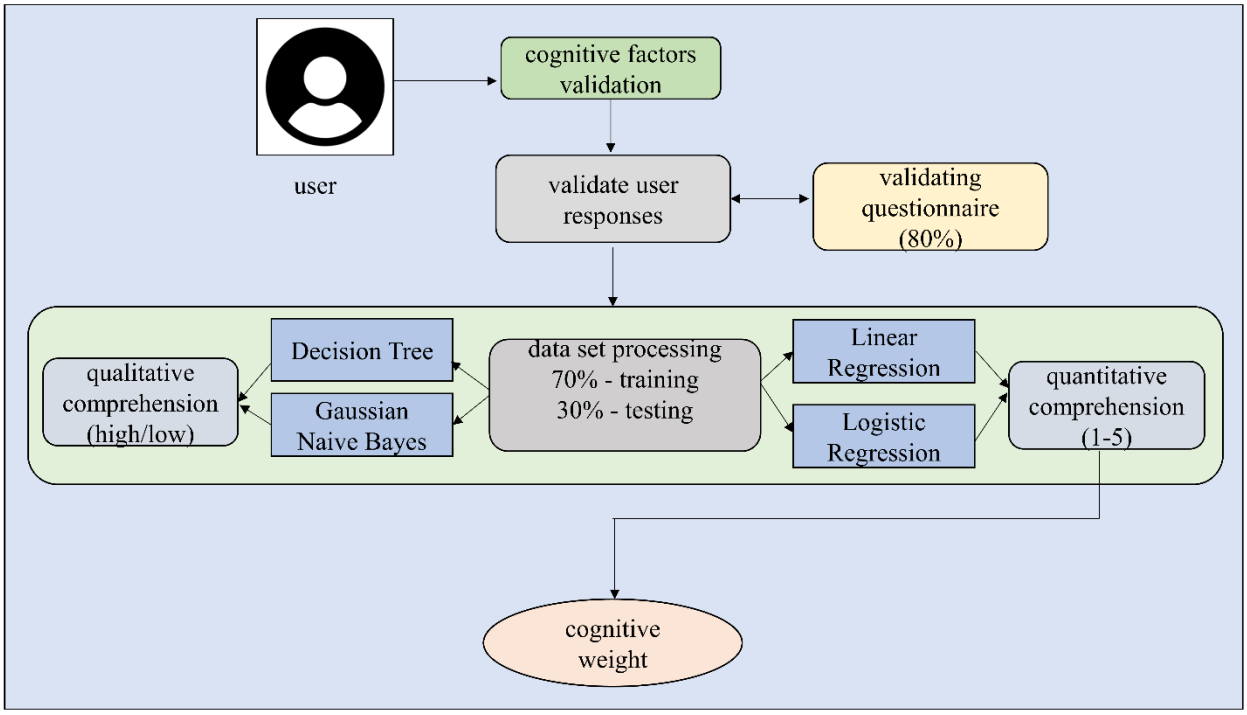


Figure 14. Overview of Cognitive Weightage Assignment Component

Based on Figure 14, the validation of the factors selected for the cognitive weight determination is preliminary performed from the user end. The factors selected for the validation purpose have been obtained from [104] and they have been listed as follows.

- The developer age – The age of the developer is associated with the individuals' experience of the computing environment [105]. It is particularly believed that a higher aged user can have more experience in coding environment than a less aged user. Nevertheless, the age cannot be considered as a direct parameter for the human comprehension as there can be several circumstances of gaining a high comprehension by less aged users.
- Familiarity of Java programming language (1-5) – The programming language is a symbolic representation of addressing a problem in a coding environment [69]. It is predominantly

different from the way that a native language is used, so that the awareness of a programming language is a vital factor behind the software development. The familiarity of a particular coding language deviates with the understandability level of a source code. The comprehensibility of a given source code becomes low when a users' analytical level of its programming language is high. On the other hand, if the problem is addressed using an unfamiliar programming language, it results with a low comprehensibility. Understanding an unfamiliar programming language consumes a considerable amount of extraneous cognitive load which does not relate with the program logic. Then, the lack of intrinsic and germane cognitive load applies to the understanding process of its actual logic, which outcomes a higher cognitive complexity. Since the experience cannot be expressed as a numerical value, a scale of 1-5 has been taken to denote it. The highest familiarity on Java language has been considered as rate 1 and rate 5 is dedicated for the least familiarity on the same. Noteworthy, the aptitude level of Java programming language has been considered, because the proposed system has been implemented through NetBeans IDE.

- Familiarity of Java coding standards and design patterns (1-5) – The quality of a source code is found to be a determinant factor of determining the users' cognitive complexity [106]. The quality of a source code can be attained by maintaining a proper coding structure inside the source code, so that each user can adhere to a common and a standard structure and refer to its internal logic easily. The structure of a source code can be maintained with respect to the way that its logic and coding have been arranged. The ways of arranging the logic can be performed by adhering to the design patterns [107] and its coding can be done with respect to coding patterns [108]. Thereby, source codes are tended to be aligned with standard designs and coding patterns to maintain the consistency of coding. Even though the expectation of maintaining proper coding standards and design patterns is to lessen the cognitive complexity, the status of the user experience on these factors has to be analysed. A user with high familiarity on the coding standards and design patterns can easily understand a given source code which results a lesser cognitive complexity. On the other hand, a user who does not have enough knowledge regarding these standards and patterns should consume a high cognitive effort, which tends to achieve a high cognitive complexity. Therefore, the user familiarity on coding standards and design patterns has been considered as a factor of determining the cognition level in this component. The rate 1 and 5 have been applied for the highest and lowest familiarity ratings regarding these factors respectively.

- Familiarity of software architecture and frameworks (1-5) – The background that a software is implemented and supported functionalities assist to understand a source code logic easily comparing to a source code implemented without a proper supportive background. The facilities supported by the computational environment consists with the software architecture and related frameworks. The expectation of these supporting services is to implement the software logic in more consistent manner. Nevertheless, the users' familiarity regarding these services cannot be neglected, since it effects for the comprehension level of source code logic. A user who has a better experience and aptitude level on these factors can comprehend the source code with less effort, and a high comprehension effort should be taken for the same for a user who does not have required aptitude and familiarity levels. Similarly in other ratings, rate 1 directs to the highest familiarity to on these parameters and rate 5 implies the least familiarity on same.
- Developer experience rate (1-5) – The experience on a computational background directs to the familiarity of the same context so that the comprehension effort of well experienced individual is lesser comparing to a less experienced user [56]. In other words, a developer with high experience in the coding background is more likely to understand a given source code than a developer with a less experience on the same background. As in other ratings, rate 1 indicates the highest experience and rate 5 denotes the least experience.
- Number of long project activities – A project with higher duration for its completion indicates its high number of functionalities as well as its complexity. This indicates the effort that each individual has to utilize is high, and it tends to gain a higher cognitive complexity. On the other hand, a project with a less completion duration indicates a lesser number of functionalities, which can be understood using less effort comparing toa high duration project. Therefore, the number of long project activities has been considered as a proportional indicator to determine the user comprehension level.
- Number of project activities – Generally, a developer in a software team has not been assigned for a single software processing. Further, the number of projects determine the workload that an individual should process, thereby it directs for the duration and attention utilized for the comprehension. As an exemplification, a lesser number of projects that each individual deals creates an opportunity to comprehend a particular source code accurately with more attention than dealing with high number of projects. Thereby, a lesser number of projects could create a

lesser cognition effort utilized for the logical comprehension, while a higher cognition effort is gained with high number of projects assigned for a particular individual.

- Number of parallel project activities – Regardless of the numerous project activities, the number of parallel project activities that a user has been assigned lays a huge impact on the cognitive effort. The number of activities that a user deals simultaneously may lower the attention level due to the lower duration and cognitive load applied for each task, so that the understandability becomes low. Therefore, the cognitive complexity becomes high due to the high effort that the user should acquire. On the other hand, a smaller number of parallel project activities can allocate more duration and attention with a higher cognitive load for the comprehension, so that the cognition effort becomes lower. Further, it is noteworthy that attaining a lower duration and a low cognitive complexity is based on the experience that the user has gained. Therefore, the number of parallel project activities has been taken as another factor of determining the comprehension level.
- Size of software – It has been found that the spatial capacity of a source code is a direct indicator of cognitive complexity [19]. The size of a software is generally denoted using LOC [109]. A source code with higher LOC tends to be more complex than a source code with a lesser LOC value. Nevertheless, there can be a certain number of circumstances where a higher LOC source code addresses a simple logic, and a lesser LOC source code indicates a complex logic. Accordingly, a direct relationship among the LOC and cognitive complexity cannot be stated. Moreover, the spatial capacity can be referred to the distance between a module/method call to its actual implementation, and the connectivity of the it to the cognitive complexity has been built in terms of cognitive load. If the distance of a module/method call to its implementation is less, the user requires a less effort for its memorization, since the cognitive load utilized for the memorization process is low. Further, a higher distance between a module/method call to its definition implies a high comprehension effort that the user should acquire due to the higher amount of cognitive load utilization. Subsequently, the distance between a module/method call to its implementation also indicates a vital role of determining the cognitive complexity under spatial capacity.
- The developer status (Undergraduate/Employee) – It is evident that the experience is a vital determinant behind the cognitive complexity, and it can be attained by the status of each individual [110]. Hence, current status of the user has been added as a personal factor of determining the cognitive level based on the experience gained. The status of each user has

been included with respect to the undergraduate and employee. A person who employs within a computing environment is known to be more experienced and comprehensive than an undergraduate. However, there are certain situations where this scenario does not exist, as an undergraduate is more experienced and comprehensive than an employee who works in a computing field.

The overview of the factors considered for cognitive weight determination from personal profile and source code factors can be observed in Figure 15.

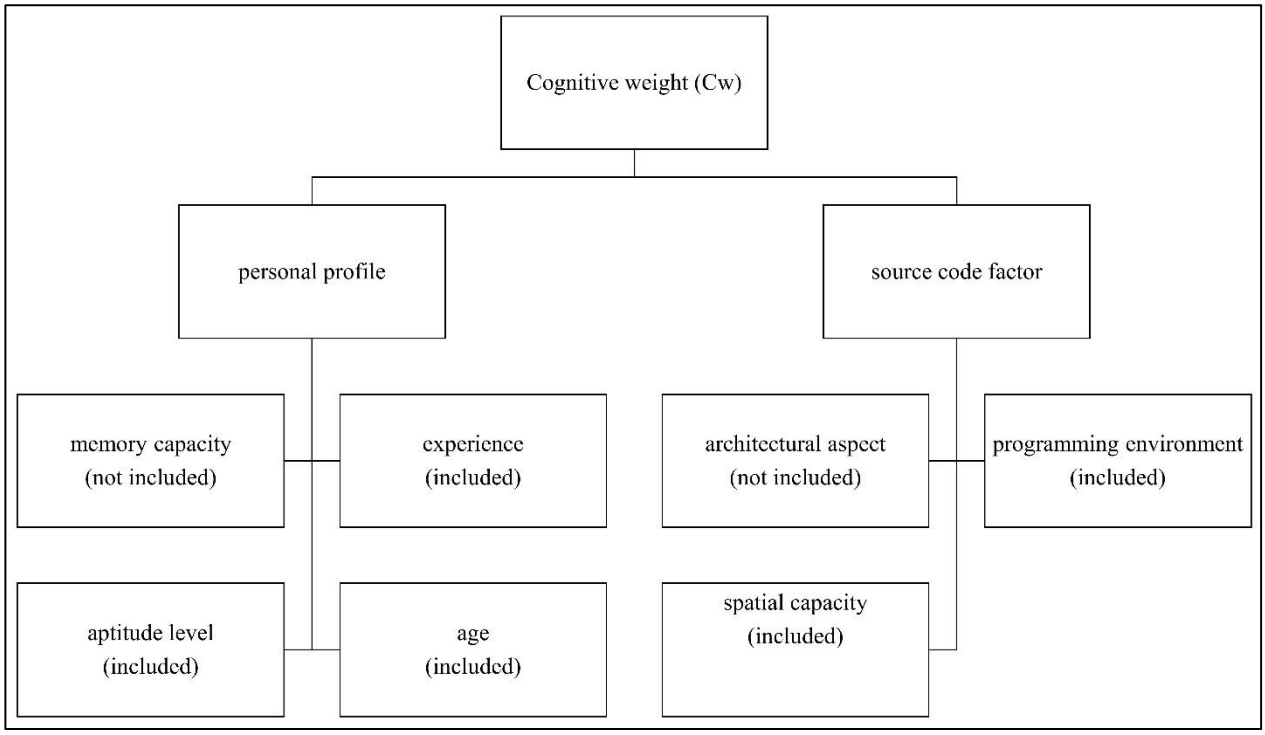


Figure 15. Factors Considered for Cognitive Weight (*Cw*)

It has been already described (section 3.2) that personal profile and source code factors are the main two factors that cognitive complexity can be described. Therefore, the effectiveness of the factors considered for the cognitive weight determination from these two factors should be studied. Even though the cognitive weight of this research work is emphasized with personal profile comparing to the source code factor, the avoidance of the source code cannot be accepted as the comprehension effort of users are monitored through the source code. As in Figure 15, users' experience of the computing background and the age have been included as the parameters to evaluate the cognitive weight. The users' aptitude level is included with respect to the familiarity of Java programming language, coding standards and design patterns, software

architecture and framework. Although the familiarity expresses the experience, the direct relationship of the aptitude level and the experience cannot be neglected, as a users' improved experience implies the growth of aptitude level. Hence, the aptitude level can be measured using the familiarity of the above parameters. The other subfactor of personal profile is the memory capacity of a user. The section 3.2 outlines the effectiveness of a persons' memory capacity and the cognitive load, and the overall system of reducing the cognitive complexity has been implemented on that basis. However, it is difficult to include the users' memory capacity as a factor of determining the cognitive weight due to the incapability of indicating it numerically. Since the procedure of handling the memory capacity and cognitive load has been discussed with the proposed system, the inclusion of the memory capacity has been deducted for the cognitive weightage allocation.

Then, the involvement of the source code factor for the cognitive weight allocation should be analysed. The architectural aspect has not been included for the cognitive weight allocation, since it is supposed to include as a major parameter inside the metric computation. The size of source code in terms of LOC has been included for the cognitive weightage allocation to demonstrate the inclusion of spatial capacity. The programming environment is already included along with the parameters used to determine the users' aptitude level as coding standards, design patterns, software architecture and frameworks are several features of a programming environment. Along with these parameters, the inclusion of the effectiveness of both personal profile and source code factor for the cognitive weightage assignment process can be verified although a higher impact is allocated for personal profile. Furthermore, the involvement of the project activities along with the parallel execution projects and long duration projects should be highlighted as the parameters considered through personal profile. Therefore, the combination of all these factors can be used to obtain an accurate cognitive weight assignment.

In the validation process, the user is given the opportunity to rate the qualitative factors within 1-5 range and to provide applicable values for the other factors as necessary. For each user rating for the qualitative factors, the component raises a related question to validate it to reduce the possibility of attaining an inaccurate cognitive weight by inaccurate user responses. The verification questions are based on Java programming language, Java coding standards, design patterns, software architecture and Java frameworks. The user rating validation is accepted for 80% thresholding marks obtained through the questionnaire. The validated user ratings are gathered as a .csv file and stored in MySQL database for later usage. Then, the dataset is trained

and tested to predict the future cognitive weight for each user. The data pre-processing has been performed to remove unused columns, noisy data and duplicate rows and labelling un-labelled columns have also been performed. Along with that, the correlations between the cognitive weight factors have been validated through Spearman method [111]. Then, 70% of data has been used for the training dataset and remaining 30% has been used as the testing data. The component has been designed to predict the cognitive weight in both qualitative and quantitative manners. The qualitative cognitive weight presents with *high* and *low* cognition labelling, while the quantitative cognitive weight uses 1-5 scale to impress the user comprehension level. The rating of 1-3 is considered as low cognitive weights, while 4-5 is considered as high cognitive weights. Consequently, weight 1 is denoted with the least cognitive level, which leads to a lesser cognitive complexity, and weight 5 is expressed with the highest cognitive level to be directed with higher cognitive complexity. The expectation of this bi-directional prediction is to avoid the challenging confirmations through the qualitative ratings. The ability to compare the predictions obtained through both qualitative and quantitative manners is another ultimate advantage of this bi-directional approach. Since the dataset consists with multiple data variables, the algorithms supported for multivariate classification have been used to train the dataset [112]. As such, Decision Tree and Gaussian Naïve Bayes algorithms have been used for qualitative cognitive weight prediction as they are classification algorithms [113], [114]. Both of Linear Regression and Logistic Regression algorithms have been used to predict the cognitive weights in a numerical range, as they can output a quantitative measurement [115]. Herein, the proposed cognitive weight mechanism can be stated as a way of emphasizing the effectiveness of personal profile regarding the comprehensibility of a source code logic. Furthermore, the capability of using it to demonstrate the subjectivity associated with the comprehension level can also be observed due to the variation of the user responses obtained for the validation factors. Therefore, the drawbacks of current cognitive weights namely the lack of personal profile involvement, limited applicability and the invalid mechanism followed to assign the weights can be verified as solved through the proposed cognitive weightage allocation mechanism.

3.5.3 A New Cognitive Complexity Metric Based on Personal Profile and Source Code Aspects

Majority of previous research works have been conducted to express the cognitive complexity as a metric, which expresses the human comprehension effort as an objective and quantitative value. Nevertheless, the lack of subjectivity signified from these metrics has created the problem of their validity in real applications [21]. Furthermore, usage of cognitive weightages along with these metrics can be observed to assist the cognitive complexity quantification by addressing the user comprehension level. Hence, the drawbacks associated with cognitive weights also existed within these metrics. As such, the necessity of deriving a proper cognitive complexity metric has been arrived. Therefore, as the solution, following mechanism has been followed to propose a new cognitive complexity metric which can be observed in Figure 16.

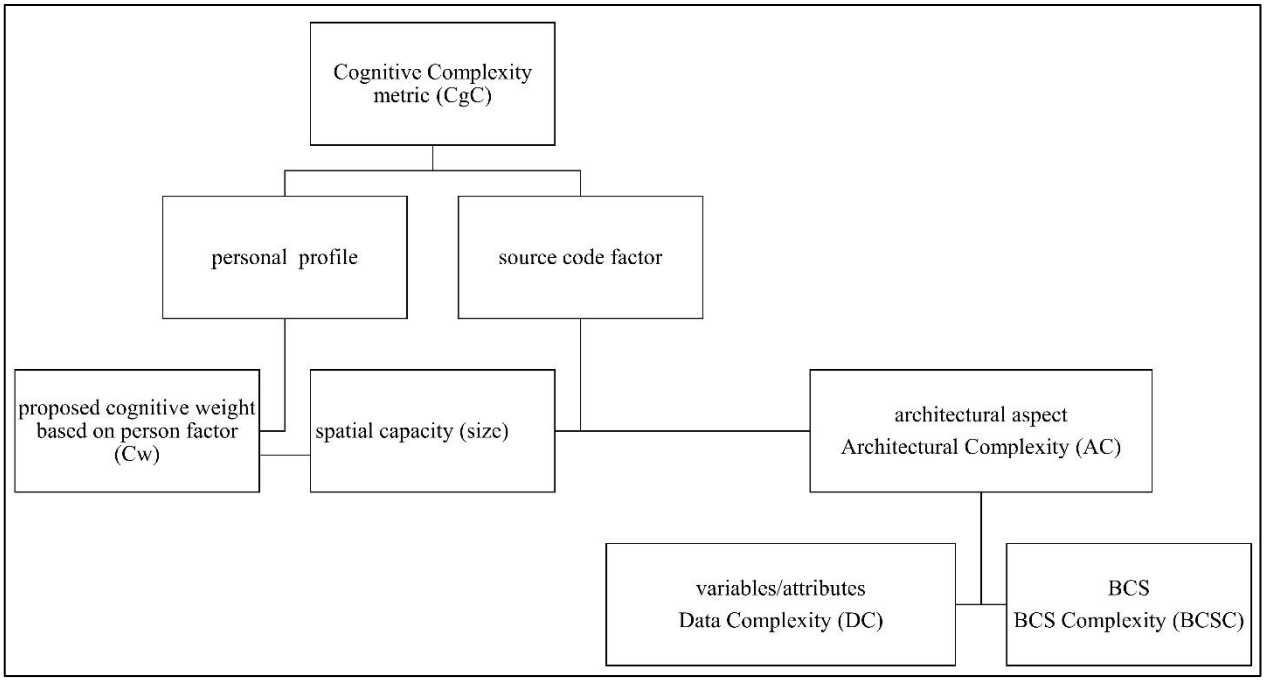


Figure 16. Overview of Cognitive Complexly Metric (CgC)

As described in section 3.2, the cognitive complexity has been divided into two major aspects namely personal profile and source code factor. Therefore, that involvement should be considered for the cognitive complexity metric as well. The involvement of personal profile has already been performed by initiating a subjective cognitive weight which has been described in section 3.5.2. Therefore, the consideration of source code factor for the metric introduction should be analysed. According to Figure 16, the quantitative effectiveness of the source code factor has been denoted

through spatial capacity and architectural aspect. Although the involvement of the programming environment comes under the source code factor as per Figure 4, it has not been considered for the metric, as it has been already considered for the cognitive weight determination. Similarly, the involvement of the spatial capacity has also been addressed in cognitive weight determination by considering the size of the source code. Hence, the inclusion of the size in terms of LOC has not been included as a parameter in the metric computation process. Generally, a real source code contains a high LOC, so including it as a parameter of computing the cognitive complexity would unnecessarily increase the complexity value, which cannot be granted as a better practise of obtaining the complexity value. The remaining sub factor inside the source code factor which is the architectural aspect has not been included in cognitive weight prediction process, so that it should be included inside the cognitive complexity metric computational process.

The architectural aspect of a source code defines the amount of information inside the source code generated for a software, and it has proven to be a major factor of determining the cognitive complexity [2]. This can be further elaborated based on the capacity of memory and the cognitive load that each user should process with the amount of information scattered within the source code. The variables in functional programming, attributes in object-oriented programming, input output parameters, operators and operands are some of the information/data items included inside a source code [22], [25], [30], [34]. One reason behind multiple introductions for cognitive complexity computation by previous research works is the observation of these data categories using different contexts. To describe it furthermore, the data is presented as number of variables while another work is aimed for input output parameters as data inside the source code. Moreover, the evaluation of cognitive complexity along with inheritance, coupling, cohesion under object-oriented programming concepts can also be observed [20], [30]. Along with these possibilities, it can be stated that the aspects of data that is applied inside a source code cannot be confined into a limited context, and if is attempted to express data by considering different categories, obtaining a single and proper cognitive complexity quantification cannot be achieved. Therefore, consideration of data scattered inside the source code has been obtained into a single context regardless of their subcategories. To explain it further, the incrementation of user comprehension effort of a source code with the increased amount of data items has been considered without navigating into its subcategories, because the concept of subcategories of data can be found within the whole data items when considering it as a single context. Therefore, all the data items have been considered as a common factor of determining the architectural aspect

without deeming their subcategories. Accordingly, each data item has been referred to a single unit of representing the architectural aspect and assigned the complexity weightage to one (1) [20], [36]. Hence, the complexity occurred due to the data items inside a source code is expressed by considering the number of data items that can be found inside the same source code. As such, the complexity of data items referred to *Data Complexity (DC)* obtained for *n* number of data items in a source code can be presented as in Equation (14.1).

$$DC = \sum_{i=1}^{n} 1 \tag{14.1}$$

Additionally, BCS are another type of information which controls the other type of information to determine the logical flow [15], [16], [37]. The consideration of the BCS has been performed by majority of research works by involving them into the cognitive weight assignment [19], [25], [30], [36]. However, the limitations of these weightages into a specific user group, the assumptions which have been made in the assignment process confines their usage in real applications. This issue has been attempted to solve in this research work by assigning the cognitive weights for BCS through a BCS related questionnaire among selected user group by analyzing their marks and durations. The methodology of that process has already been described in section 3.5.1, but its inability to represent the entire architectural aspect only by using BCS is still existed. Hence, the impossibility of using BCS with cognitive weightage assignment can be highlighted. However, consideration of BCS to determine the cognitive complexity is a mandatory factor with respect to the source codes' architectural aspect as their feature of controlling the flow of logic cannot be ignored for the user comprehension. As a solution, a process of including BCS concept to the metric computation procedure without analyzing the comprehension effort them through cognitive weights has been introduced. Subsequently, the concept of including the number of BCS inside a source code has been considered. Through that, the proportional relationship of the number of BCS and the user comprehension effort can be stated, as the comprehension effort of a user increments along with the number of BCS incrementation due to the expansion of the control flow by BCS. Additionally, the influence of nested BCS for the cognitive complexity has also been considered, since the involvement of nested BCS makes the source code understanding process more complex than having a single BCS [19]. Therefore, the concept of having a single and nested BCS inside a source code has been addressed by considering the level associated with them. To explain it furthermore, if a source code consists with two nested *for* loops, the outer *for* loop is set to be in level 1 and the inner *for* loop is set to be in level 2. Moreover, if a source code contains only a single *for* loop, its

level is assigned with level 1. Through that, the effectiveness of single and nested BCS for the BCS complexity can be determined, and thereby the variation of the user comprehension effort can be expressed. Therefore, the levels assigned for each BCS has been added for the metric computation. As such, the complexity derived by using BCS is denoted as *BCS Complexity (BCSC)* and computed as in Equation (14.2), where *m* is the total number of BCS appeared in a source code and k_i represents the BCS level assigned for i^{th} BCS.

$$BCSC = m + \sum_{i=1}^{m} k_i \tag{14.2}$$

As such, architectural aspect of a given source code has been defines as the combination of variables/attributes and BCS inside a given source code. Therefore, *Architectural Complexity (AC)* of a given source code can be considered as the complexities generated through *DC* and *BCSC*, which can be denoted as in Equation (14.3).

$$AC = DC + BCSC \tag{14.3}$$

Along with Equation (13.1) and (13.2), the possibility of stating the architectural complexity of a source code with respect to its data items and BCS can be claimed. Furthermore, the cognitive weight represents the involvement of personal profile, spatial capacity and factors of programming environment under source code factors. Therefore, all the factors associated with the cognitive complexity has been included in the metric computation as well. As such, the *Cognitive Complexity metric (CgC)* can be expressed as summation of the *personal cognitive weight (Cw)*, *AC* as shown in Equation (14.4).

$$CgC = Cw + AC \tag{14.4}$$

The reason of expressing *CgC* by getting the addition of *Cw* and *AC* should be clearly defined. One reason behind getting the addition of these two components is the consideration of cognitive complexity as a combination of personal profile indicated with the cognitive weight (*Cw)* and source code aspect indicated with architectural aspect (*AC*). It should be noted that the combination of *Cw* and *AC* cannot be considered as a form of multiplication. To describe it furthermore, in case of *Cw* and *AC* get multiplied, *CgC* value gets equal to the value of *AC*, if *Cw* is obtained as 1 for a particular person. Accordingly, effectiveness of complexity associated with *Cw* for cognitive complexity determination cannot be demonstrated properly. On the other hand, if *Cw* and *AC* get added, the effectiveness of complexity related with *Cw* can be demonstrated properly for the same example scenario, since the weight of 1 is added to existing *AC* as the value

obtained for *CgC*. Thereby, the necessity of getting the addition of *Cw* and *AC* should be maintained for *CgC* computation process.

Furthermore, Equation (13.4) can be further described by using Equation (14.1), (14.2) and (14.3), where number of variables/ attributes inside the source code is *n*, *m* is the number of BCS and k_i is the BCS level associated with i^{th} BCS and shown in Equation (14.5).

$$CgC = Cw + DC + BCSC$$

$$CgC = Cw + \sum_{i=1}^{n} 1 + m + \sum_{i=1}^{m} k_i \quad (14.5)$$

This equation of quantifying the cognitive complexity can be considered as a vital achievement due to the inclusion of both person and source code factors involving with human comprehension. More significantly, the subjectivity of the human comprehension effort has also been included inside the equation as the form of cognitive weight (*Cw*) with more emphasis on personal profile, which cannot be seen in previous works. Thereby, this achievement can be stated as more practical to express the user cognition effort in real applications.

3.5.4 Complexity Calculation through Standard Complexity Metrics

As an additional feature, a component of calculating the software complexity of a given source code with respect to a certain number of metrics has been introduced. The software complexity is defined as the extent of the difficulty level associated with a component, design or its source code [49], and to assess this property numerous software complexity metrics are being used in real applications. Similarly in current cognitive complexity metrics, the aspects considered for each software complexity metric computation are also different, so that different quantifications to expose the complexity of a given source code can be observed. Therefore, each of software complexity metrics can be considered as different types of indicators to define the software complexity. However, the definition of cognitive complexity can be related with software complexity, as both metrics attempt to evaluate the understandability level associated with a software using its source code in different aspects. As such, a component of calculating the software complexity using a certain number of complexity metrics has been included to the proposed system. Therefore, the system represents two types of complexity computations as shown in Figure 17.

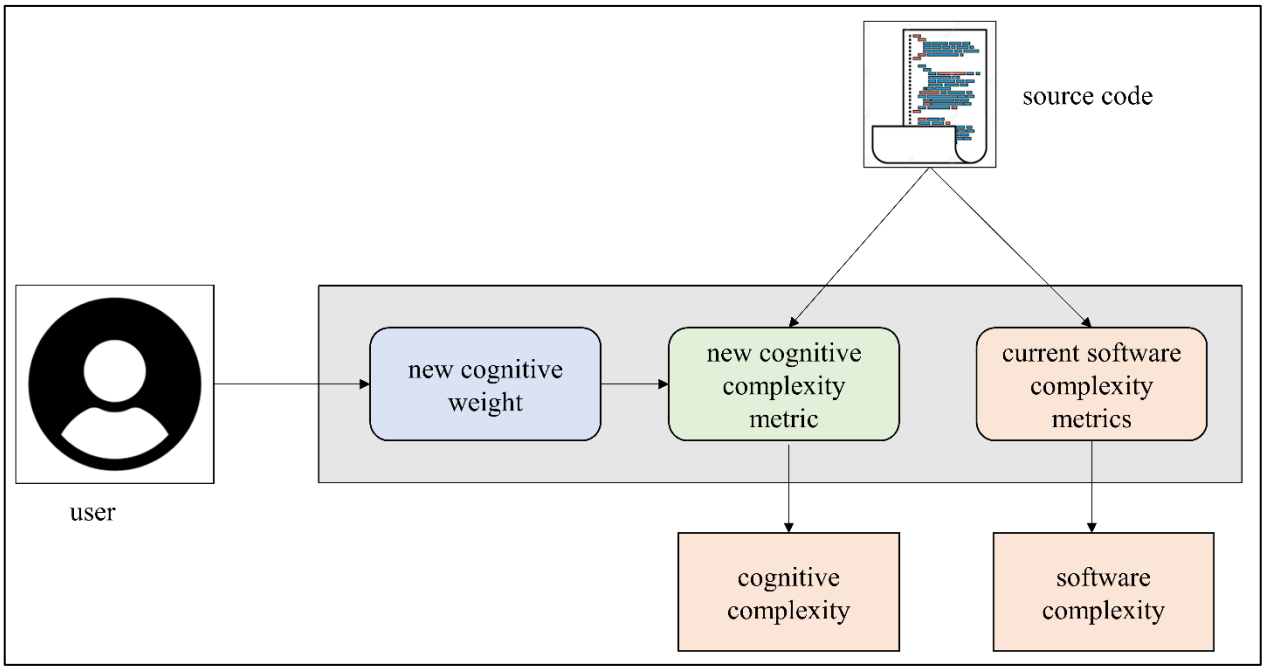


Figure 17. Overview of Complexity Calculation using *CgC* Metric and Existing Software Complexity Metrics

Based on the Figure 17, the system is capable of computing the cognitive complexity using the proposed cognitive complexity metric, while its software complexity is measured using the supported complexity metrics by the software complexity calculation component. Through that, it is expected to observe an existence of a relationship among two complexity measurements as per their definitions. The overview of the software complexity computation component can be viewed in Figure 18.

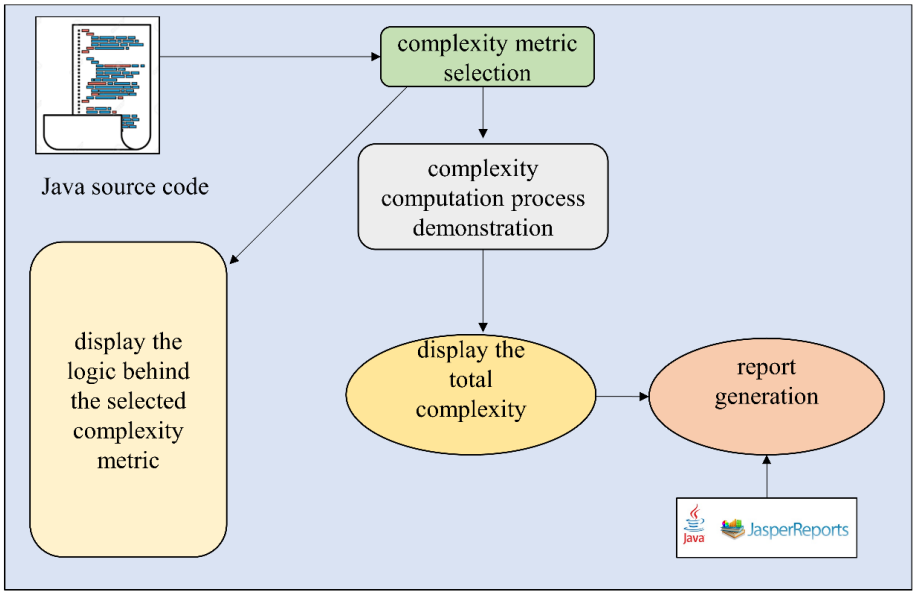


Figure 18. Overview of Complexity Calculation using Existing Software Complexity Metrics

According to Figure 18, java source code is the input for the complexity computation component. It is significant that the same procedure followed in code quality optimization component is directed to this component as well. As such, a copy of the inputted source code is maintained by the component to remove the comments and unnecessary white spaces. The purpose is to identify Java keywords (reserved key words), identifiers, numbers, operators, escape characters, variables, classes and methods in each line of code. All the identifiers, numbers, operators, escape characters, variables, classes and methods in each line of code is identified based on how Java key words are used inside the source code, and they are stored in different collections for future usage. Based on the users' selection of the complexity metric, the component is able to use the record of related code elements for the calculation process. Once user selects a particular complexity metric, its underlying logic is displayed for user to acknowledge about the calculation procedure. The software complexity metric supported for this component are listed in Table 16.

Table 16. Existing Software Complexity Metrics Handled by Complexity Computation Component

Software Complexity Metric	*Description*
Line of code and Comment percentage metric	Lines only with curly braces, single and multiple line comments are not considered as a line of code [116] Comment percentage is calculated by considering the single and multiple line comments and dividing it by the actual line of code value [117] Recommended to maintain the actual line of code value in 4-400 range in which the function length is for 4-40 program lines and file length is 4-400 program lines Comment percentage is maintained to have more than or equal to 30%.
Complexity due to code element size metric	Java reserved key words, identifiers, operators and operands are considered as code elements Access modifiers used as Java key words are not considered

Table 16 continued

	All the identified code elements are assigned with weight 1 and the key words “new”, “throw”, “throws” are assigned with weight 2 due to their usage in handling objects and exceptions
Complexity due to recursive methods metric	Lines applied with the recursion are assigned with weight 2 to observe the recursive components clearly [118] (size complexity of the same source code is also displayed through this metric selection)
Cyclomatic Complexity (CC) metric	Determines the number of linear independent paths inside a given source code, which is preliminary based on the number of BCS [37] using two equations $CC = d + 1$ d – number of decision statements in the control flow graph $CC = e - n + 2$ e – number of edges in the control flow graph n - number of nodes in the control flow graph Methods, conditional statements, looping structures are assigned with weight 1 Boolean operators such as conditional AND (&&) and conditional OR (||) are assigned to 1 Recommended to maintain the CC value less than 20 [119]
Cognitive Complexity metric by G. Ann Campbell [2]	Ignore structures that allow multiple statements to be readably shorthanded into one Increment weight by 1 for each break in the linear flow of the code Increment the weight when flow-breaking structures are nested

Table 16 continued

Halstead metric	Computes the software complexity in terms of the operators and operands inside the source code given [120] Reserved key words that specify the type, all identifiers which are not reserved words and constants are considered as operands Reserved key words that specify storage class, qualify types and reserved key words which do not identified as operands and conditional operators are considered as operators Use nine formulas for the calculation

Table 16 concluded

Further, the user is given the opportunity to observe the procedure of the calculation process applied for each line of code, and the total complexity is displayed at the last. The report of the complexity computation process can be converted into a format of report with the aid of JasperReports for the download purpose.

Therefore, the opportunity of computing the cognitive complexity using proposed cognitive complexity metric and the software complexity using the metrics listed under Table 16 for a same source code has been created by the system. Thereby, the possibility of obtaining the variation and relationship of cognitive complexity and the software complexity can be gained.

3.6 Summary

Throughout this chapter, the applicability of cognitive complexity inside software development and maintenance processes has been analysed in terms of reducing the cognitive complexity and computing the cognitive complexity by proposing a new cognition metric. Firstly, the main factors effect for the cognitive complexity has been analysed, and the subfactors associated with these main factors have been presented to describe the term cognitive complexity in numerous directions. Accordingly, cognitive complexity has been described using two main factors namely personal profile and source code factor. The reason behind these two factors selection is the definition of cognitive complexity which determines the comprehension effort of users for a given source code. The comprehensibility of a given source code is essential for two parties. Users who deal with software development and maintenance processes should be aware of its source code logic, since the success of these phases depends on how they can understand the logic behind it. Further, the comprehension of a source code by computer happens when the source code is debug and executed to produce the output. However, it has been mentioned that this research work focuses about the users involving in software development and maintenance phases such as developer, software engineer and quality assurance engineer. As the next step, subfactors associated with these users' personal profiles and source codes have been analysed. The comprehension process is considered as a process involved with the memory capacity of a particular individual, so that capacity of memory of a person has been considered as one factor inside personal profile. Further, memory capacity has been found to be categorized into long term, short term and working memory, where long term memory refers to the long-lasting memory, short term memory is for the information kept for a shorter duration, while working memory is associated with the duration that the user deals with comprehending the given source code. Thereby, the significance of handling the working memory at the time of dealing with a source code with the aid of long term and short-term memories has been highlighted. Managing the working memory has been described using cognitive load which has been categorized into intrinsic, extraneous and germane. Intrinsic cognitive load determines the amount of memory required to identify the exact logic of a source code, which refers to the actual complexity at the end. Extraneous cognitive load refers to the memory allocated for excessive distractions while germane cognitive load refers to the capacity of linking with new ideas along with the information of long-term memory. Furthermore, aptitude level on programming has been

considered under personal profile for cognitive complexity determination. Under that, preference on a particular programming language and the problem analytical skill have been taken. Further, the users' experience and age have also been added as the subfactors under personal profile. The other main factor to express the cognitive complexity is the source code. The first subfactor taken under source code is the architectural aspect which expresses the amount of information inside a source code. This is the factor considered by majority of previous research works to represent the cognitive complexity with different dimensions. Based on the previous works, the consideration of architectural aspect has been commonly categorized into variables/attributes, BCS, operators and operands, input output parameters and object-oriented concepts. Then, the spatial capacity has been introduced as another subfactor inside source code factor. It is also used by previous research works such that it refers to the size in terms of LOC, and the distance in LOC from a module call to its implementation. As the other subfactor of the source code, programming environment has been considered which includes the source code structure and the availability of comments. Therefore, cognitive complexity has been described with respect to memory capacity, aptitude level on programming, experience and age under personal profile, while architectural aspect, spatial capacity and the programming environment have been considered through source code aspect.

Then, the association of these parameters to handle the cognitive complexity inside software development and maintenance processes has been discussed. It has been stated that the understandability of a source code is the direct co concept behind cognitive complexity, so that the expectation has been described to have a high understandable source code which can utilize a less comprehension effort to achieve less cognitive complexity. Therefore, the procedures of reducing the cognitive complexity have been explored. It has been highlighted that the consideration of personal profile over the source code factor to express cognitive complexity in this research, so that the reduction of cognitive complexity has been analysed in terms of handling the factors inside personal profile. Hence, the effectiveness of cognitive complexity has been described as a human comprehension process, in which the cognitive load plays a vital role inside it. Therefore, the procedures of handling the cognitive load to achieve a less comprehension effort have been studied. It has been found that reducing unnecessary (extraneous) cognitive load and assisting for necessary (intrinsic and germane) cognitive load can reduce the human comprehension effort. Accordingly, the mechanisms of reducing unnecessary cognitive load and assisting for necessary cognitive load have been explored. As such, using an

error free and structured source code have been found to reduce unnecessary cognitive load, while providing visualization, simulation, recommendations and guidelines for a source code logic have been supported for necessary cognitive load.

Demonstrating the procedures of reducing unnecessary cognitive load and assisting necessary cognitive load in a computational background has been implemented using the proposed system. Consequently, referring to the requirements of software and visualizing its logic have been added as the components under assistive cognitive load mechanism. The process of the visualization component has been categorized into two visualization procedures, where the logical diagrams are constructed with and without the source code. The visualization without the source code happens by referring to its requirements. Therefore, the user is given the opportunity to refer both types of diagrams of same source code though the comprehension process. As for the components under cognitive load reduction, defects tracing, code quality optimization and refactoring components have been included. The ultimate goal of these components is to reduce the effort utilized in the comprehension process and to make the functionality of software development and maintenance processes in a smooth way.

The introduction of a new cognitive complexity metric is the other significant outcome of this research, which has been already included as another component inside the proposed system. As the preliminary step, the concept of cognitive weight has been studied, which expresses the understandability level using a numerical value. Based on previous research works, this has been introduced based on the architectural aspect of a source code, and the consideration of BCS can be widely observed in majority of works. But, the limitations of the applicability these weights into a specific user group and the assumption considered for the weightage allocation has emphasized their invalidity. As a solution, a mechanism of assigning cognitive weights for BCS through a validated framework has been considered, in which the weightage assignment is conducted based on the results achieved through a questionnaire. However, involvement of personal profile to cognitive weights assignment process cannot be seen in this procedure due to the emphasis given to BCS through this research work. Hence, a new cognitive weight has been proposed as a predictive factor by analysing the factors of personal profile and source code which have been verified by both user and the system. The developer age, familiarity of Java programming language, familiarity of Java coding standards and design patterns, familiarity of software architecture and frameworks, developer experience, number of project activities, number of long project and parallel project activities, size of software in LOC and the developer

status have been considered for the cognitive weight allocation, which includes both of personal profile and source code factors. The inclusion architectural aspect under source code aspect has not been included as it has been already considered in the metric computation process. Also, a high effectiveness of personal profile involvement can be observed with respect to a high number of personal profile factors consideration over source code factors. Moreover, the capability of obtaining a cognitive weight which varies with each user is the significant achievement of this scenario, as it can demonstrate the subjectivity associated with the users' comprehension levels. Along with this, the process of initiating a subjective and quantifiable cognitive weight for the new cognitive complexity metric has been attained.

The proposed cognitive complexity metric also includes the association of personal profile and source code factor as it has already described as two major aspects of cognitive complexity. Since the involvement of personal profile is presented in proposed cognitive weight, it has been taken as the representation of personal profile inside the metric, while source code aspect has been added through a calculation process. The spatial capacity of source code in terms of size in LOC has already been considered with cognitive weight, so that it has been removed from the calculation. The effectiveness of architectural aspect has been initiated with respect to the variables/attributes and BCS in more abstract form to avoid inconsistences occurred with its expression with multiple dimensions, which has been observed as the reason behind numerous outcomes to express the cognitive complexity in literature. Consequently, the architectural aspect complexity is obtained as the addition of the complexities due to variables/attributes and the complexity due to BCS. Along with that, the addition of proposed cognitive weight, which includes personal profile together with source codes' spatial capacity included to the new cognitive complexity metric, which is capable of presenting the human comprehension effort as a quantitative measurement. It is significant that the inclusion of subjective cognitive weight effects to demonstrate the subjectivity of human comprehension effort through the proposed metric, which can be considered as a vital achievement over the previous cognitive complexity metrics.

Therefore, the initiation of procedures to lessen the cognitive complexity in software development and maintenance processes in a computational background and the introduction of a subjective cognitive weight allocation mechanism with a new cognitive complexity metric are the major achievements of this research project.

4.0 RESULTS AND DISCUSSION

4.1 Introduction

The first outcome of this research work is the introduction of several computational procedures, which can demonstrate cognitive complexity reduction inside software development and maintenance processes. The other outcome is the introduction of a new cognitive complexity metric based on person and source code aspects, which is capable of emphasizing the subjectivity associated with human comprehension levels using the proposed subjective cognitive weight. Even though the purpose of introducing the cognitive complexity reduction components can be validated logically and theoretically, they should be empirically validated as well to confirm their stability in real usage. The same condition should be applied to the introduction of a new cognitive complexity metric as well. It should also be validated theoretically and empirically to promote its applicability in real applications, while demonstrating its relativity to the definition of cognitive complexity through a numerical representation. Therefore, the evaluation procedures of both of these outputs have been discussed in this chapter.

The components implemented to address the cognitive complexity reduction should be verified to achieve a less comprehension effort on a given process. It has been found that the duration taken to process a given task implies the effort taken to understand its underlying logic [121], [122]. This can be further explained as a higher effort utilized to comprehend a given source code involves a considerable number of processing involved, which takes a higher duration. The understandability of a source code tends to be easy, when the comprehension effort is lower. Consequently, it involves with less processing which utilizes a less duration. Therefore, the duration applied to comprehend a source code can be considered to verify the comprehension effort, thereby the effectiveness of cognitive complexity as well [34]. As such, the duration has been considered as a major parameter to analyze the cognitive complexity reduction of the proposed system components. The procedure described in [34] has been followed to analyze the duration attained for a task completion by a set of users. Accordingly, the duration taken to perform a certain task inside a given source code without system components and by using system components along with the source code have been monitored and analyzed. Nevertheless, the familiarity obtained through repeating the same task in the same source code also effects to the cognitive complexity, as it changes with the duration. Therefore, a time gap of one week has

been maintained to repeat the same task with system components in order to measure the duration. Thereby, it has been assumed that users are novel to the source code so that their cognitive loads have not been filled with the previous event information, which they have referred one week prior. The same set of users considered for cognitive weightage assignment for BCS described in section 3.5.1 has been used for this evaluation criteria as well. Moreover, those users have been categorized into subgroups to lower down the users in a particular group and to create the facility to analyze the results more conveniently. As such, four different user groups, in which each user group consists with 100 users were selected for the study. The students were selected based on four different score levels obtained through the selection questionnaire such that 60%-70% scored students belongs to the first group (G1), 70%-80% scored students belongs to the second group (G2), 80%-90% scored students belongs to the third group (G3) and 90%-100% scored students belongs to the fourth group (G4). For the analysis process, thirty different software have been considered.

It should be noted that this research work analyzes the comprehension effort utilized for a source code which is implemented with a correct logic. Hence, the requirement of validating the accuracy of these components have been ignored as incorrect functionalities of these components cannot lead to a correct logic implemented inside the source code. On the other hand, the proposed metric should be undergone with a validation procedure to verify its applicability in software domain. The validation has to be performed theoretically with standard software complexity metrics frameworks. Further, it should be evaluated against the drawbacks of current cognitive complexity metrics.

4.2 Evaluation of Requirements Analysis Component

The purpose of the requirements analysis component is to gain the class names as requirements by analyzing the formatted project proposal document. The possibility of using these class names for the visualization process can also be highlighted as the logical diagram generation is performed after finalizing the requirements inside SDLC. The possibility of referring to the requirements of a software to make the source code understanding process easier is the core concept behind the implementation of this component. As it has been already described, the duration taken to accomplish a certain task is used to practically analyze the comprehension

effort of this component. Accordingly, the effectiveness of requirements analysis component to reduce the users' comprehension effort has been measured in terms of duration. The users were given the source codes implemented for thirty software along with their requirements generated by requirement analysis component. They were asked to perform a modification for each source code by referring only to the given source code. The same task has been repeated by providing the generated requirements as well. The modification includes implementing a new method by comprehending the logic of the whole source code, so that the time taken to implement the method can imply the comprehension effort utilized for it. This procedure has been monitored through a Moodle environment to gain the duration accurately. As such, the duration taken to accomplish this task by referring only to its source code and with its requirements have been recorded with respect to each subgroup maintaining one week time gap among both processing. However, there can be certain situations that users complete the modification inaccurately. In that scenario, users were asked to repeat the same process until the correct modification is outputted, so that the average duration is obtained as the duration taken to apply the modification. As such, each source code modification has been performed by four subgroups, so that the average duration taken by four subgroups for each source code has been computed to make it easy for use and analyze. Accordingly, the average duration taken by each of subgroup to apply a modification inside thirty different source code of software have been shown in Figure 19.

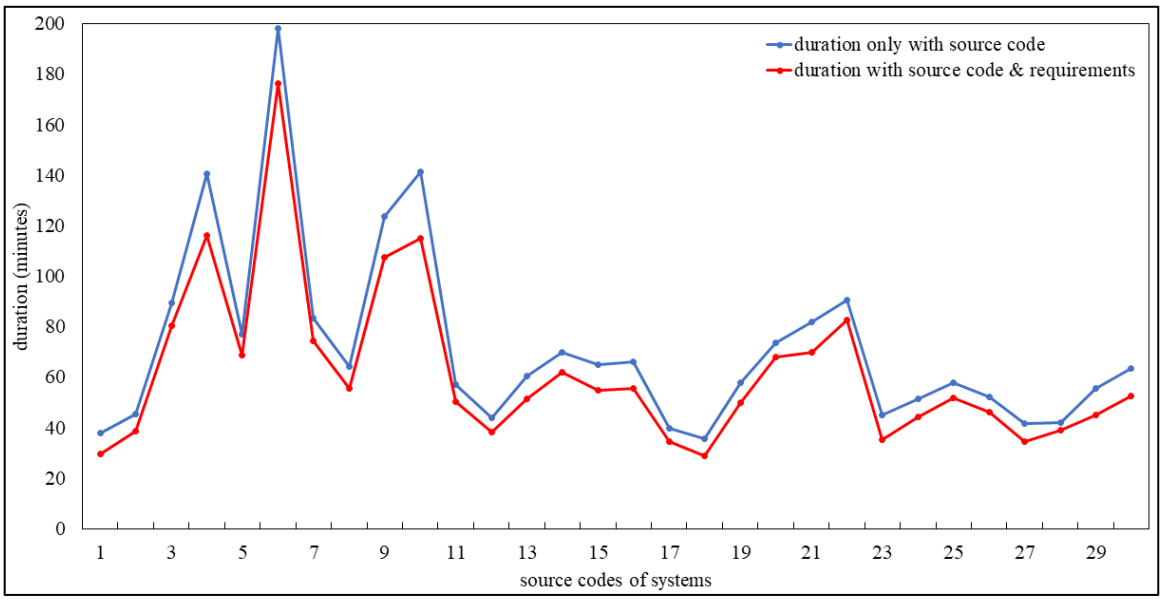


Figure 19. Average Duration with Requirements Analyzer

Based on Figure 19, average duration utilized to implement a new method inside given source code can be observed as less with the reference of requirements generated by requirements analysis component. To describe it furthermore, the average duration to implement a new method can be reduced by comprehending the logic of its source code and along with its requirements, as referring only the source code to perform the same process takes more time. Based on the recordings, it can be noticed that total average duration taken for this study by referring only to the source code has obtained 71.75 minutes, while the total average duration for same processing with additional usage of requirements has led to 61.9 minutes. Therefore, proposed requirements analysis component has gained 13.83% duration reduction comparing to comprehension process followed only with the source code. A lesser duration implies the reduction of the cognitive effort, thereby the reduction of cognitive complexity can be stated. Therefore, usage of above requirements analysis component can be stated as reducing the cognitive complexity of a given source code.

Furthermore, the assistance of requirements analysis component with respect to the duration reduction has been verified statistically. Since the experiment has been conducted through same user group for multiple scenarios, paired samples T test in SPSS has been performed for the comparison process. The expectation is to verify a reduction in cognitive effort in terms of lesser duration taken with source code and the requirements usage. The output of T test can be observed in Table 17.

Table 17. Output of Paired Samples T Test conducted for Requirements Analyzer

Paired Samples T Test

		Paired Differences					t	df	Sig. (2-tailed)
		Mean	Std. Deviation	Std. Error Mean	95% Confidence Interval of the Difference Lower	Upper			
Pair 1	Duration_Only_With_Source_Code – Duration_With_Source_Code_And_Requirements	9.86000	6.73151	.61450	8.64323	11.07677	16.046	119	.000

Based on this result, it is 95% confident that the true mean difference of durations is in between 8.64323 and 11.07677. It should be noted that all these analyses have been done with 5% significance level (α), so that confidence level has been taken as 95%. In order to analyze the results generated by paired sample T test, the hypothesis test has been performed as follows.

Let μ1 be the total mean time taken for source code comprehension only by referring the source code

Let μ2 be the total mean time taken for source code comprehension by using the requirements along with the source code

The null hypothesis (H_0) is defined as, H_0: μ1 <= μ2

The alternative (H_1) is defined as, H_1: μ2 < μ1

Rejection region (p value approach) of H_0

p value < significance level (α)

p value is the value under Sig. (2-tailed) in generated output. This test is generated as 1 tailed due to the comparison process performed for different means, however the result is generated for 2 tailed. Therefore, p value should be obtained as Sig. (2-tailed) /2.

0.000/2 < 0.05

0.000 < 0.05 → condition is true

Therefore, null hypothesis (H_0) is rejected, which in turns accepts the alternative (H_1).

Hence, it can be verified that the mean time taken to comprehend a source code with requirements can reduce the duration.

Along with the statistical analysis of paired samples T test, the duration taken to understand a source code logic is lesser with the assistance of it is requirements. It implies that the duration taken to understand a source code logic is higher without any assistance. A lesser duration implies the reduction of comprehension effort taken by users, so that it can be verified that the requirements analysis component can reduce the cognitive complexity as an assistive mechanism to users' necessary cognitive load. Moreover, the concept of referring to the software requirements in a situation where the source code is difficult to understand can also be verified again to make the comprehension process simpler.

4.3 Evaluation of Visualization Component

According to Figure 6, the visualization component is consisted with two visualization mechanisms, in which one subcomponent is responsible to generate UML diagrams without the source code and the other component generates diagrams by using the source code. A sample of class diagram, ER diagram and object diagram generations without using the source code and a sample of sequence diagram and class diagram generation by using the source code have been included in Appendix A and Appendix B respectively. The combination of both subcomponents has been used as a mechanism for assisting the necessary cognitive load of users' to reduce the cognitive effort and complexity. Accordingly, another thirty different source codes have been given to the same set of users to implement a new function by comprehending its logic. The duration taken to implement a new function correctly by using only the source code and by using their visualized UML diagrams along with the source code have been recorded separately. Supporting to the logical comprehension using visualization has been performed in two ways. The logical diagrams generated by an existing visualization technique EasyUML, which is supported for NetBeans have been given for the user along with source codes. The diagrams generated by Visual Paradigm have not been considered, as it provides user to generate the diagrams. Hence, the duration to process the same functionality with the diagrams of proposed component has also been recorded after. It should be noted that all these three processes have been performed by keeping one week duration gap to omit the familiarity related with logical comprehension. As it is performed in requirements analysis component, the durations taken for incorrect modifications have been recorded by computing the average duration of obtaining the correct modification. As such, the mean duration acquired for each group of users for three different scenarios are presented in Figure 20.

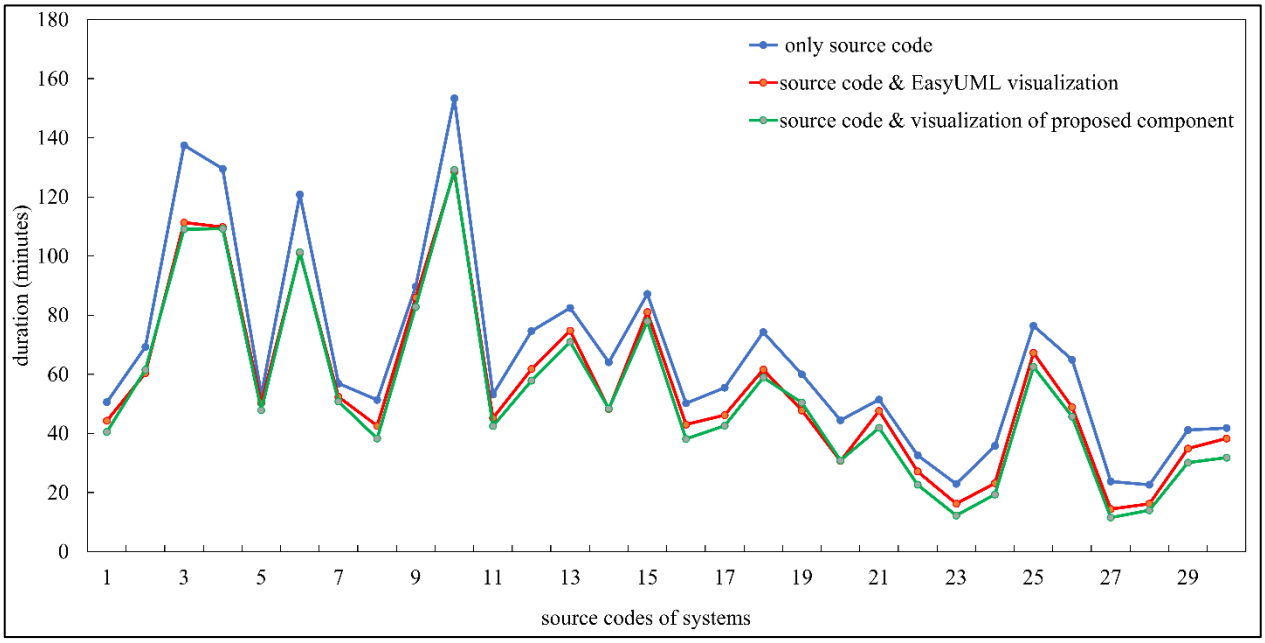


Figure 20. Average Duration with Visualization Component

Based on Figure 20, total average duration of applying the modification by comprehending the source code without any assistance results in 65.73 minutes. The total average duration for same processing with their logical diagrams using EasyUML and with proposed component have outputted 55.37 and 52.71 minutes respectively. Therefore, a significant duration reduction of 19.81% can be observed in the proposed visualization component comparing to the comprehension with no visualization, which implies the development of users' comprehension level with generated UML diagrams. Further, 4.81% duration reduction can also be observed in proposed visualization component comparing to the visualizations generated by EasyUML. As such, usage of the proposed component over EasyUML can be stated as a better way of illustrating the diagram logic, which tends to increase the user comprehension level comparing to the current visualization techniques. The comprehension level development of users tends to lessen the comprehension effort. Hence, the proposed visualization component can be stated as a way of reducing cognitive complexity effected with users.

The assistance of this component in terms of reducing the duration has been verified statistically using paired samples T test. The output of T test can be observed in Table 18.

Table 18. Output of Paired Samples T Test conducted for Visualization Component

Paired Samples T Test

		Paired Differences					t	df	Sig. (2-tailed)
		Mean	Std. Deviation	Std. Error Mean	95% Confidence Interval of the Difference				
					Lower	Upper			
Pair 1	Duration_Only_With_Source_Code - Duration_With_Proposed_Visualization	13.01850	7.34558	.67056	11.69073	14.34627	19.414	119	.000

Table 18 shows that the existence of true mean difference of the durations taken with source code and with component visualization is in between 11.69073 and 14.34627 along with 95% confidence. The hypothesis testing to check the reduction of comprehension duration with visualization component has been shown below.

Let $\beta 1$ be the total mean time taken for source code comprehension only by referring the source code

Let $\beta 2$ be the total mean time taken for source code comprehension of a source code by using the logical diagrams generated by proposed component

The null hypothesis (A_0) is defined as, A_0: $\beta 1 <= \beta 2$

The alternative (A_1) is defined as, A_1: $\beta 2 < \beta 1$

Rejection region (p value approach) of A_0

p value < significance level (α)

p value is the value under Sig. (2-tailed) in generated output. This test is generated as 1 tailed due to the comparison process performed for different means, however the result is generated for 2 tailed. Therefore, p value should be obtained as Sig. (2-tailed) /2.

$0.000/2 < 0.05$

$0.000 < 0.05$ → condition is true

Therefore, null hypothesis (A_0) is rejected, which in turns accepts the alternative (A_1).

Hence, it can be verified that the mean time taken to comprehend a source code with visualization can reduce the duration.

Therefore, the verification of the proposed visualization component can be introduced as a mechanism to reduce the comprehension duration of users for a given source code. Accordingly, the reduction of the users' comprehension effort can be emphasized with the support of UML diagrams, which in turns can assist to reduce the cognitive complexity. Hence, the guidance given by visualizing the source code logic in terms of UML diagrams can be concluded as a mechanism of reducing the cognitive complexity of a source code.

Furthermore, the possibility of using the proposed component for UML diagrams visualization over the existing visualization EasyUML has been determined statistically. However, it is noteworthy that verification of this component over other available visualizations is not applicable for all types of UML diagrams, as it does not support for all types of UML diagram generations. The paired samples T test has been conducted for the comparison process and its result is shown in Table 19.

Table 19. Output of Paired Samples T Test conducted for Visualization Component with EasyUML

Paired Samples T Test

		Paired Differences					t	df	Sig. (2-tailed)
		Mean	Std. Deviation	Std. Error Mean	95% Confidence Interval of the Difference				
					Lower	Upper			
Pair 1	Duration_With_Source_Code_And_EasyUML - Duration_With_Proposed_Visualization	2.66517	3.93022	.35878	1.95475	3.37558	7.428	119	.000

Based on Table 19, it is 95% confident that the true mean difference of the durations taken by the proposed component and EasyUML lies in between 1.95475 to 3.37558 minutes. Further, the procedure of hypothesis testing to imply the duration reduction of the proposed component has been shown below.

Let θ1 be the total mean time taken for source code comprehension of a source code using the diagrams generated by EasyUML

Let θ2 be the total mean time taken for source code comprehension of a source code by using the logical diagrams generated by proposed component

The null hypothesis (X_0) is defined as, X_0: θ1 <= θ2

The alternative (X_1) is defined as, X_1: θ2 < θ1

Rejection region (p value approach) of X_0

p value < significance level (α)

p value is the value under Sig. (2-tailed) in generated output. This test is generated as 1 tailed due to the comparison process performed for different means, however the result is generated for 2 tailed. Therefore, p value should be obtained as Sig. (2-tailed) /2.

0.000/2 < 0.05

0.000 < 0.05 → condition is true

Therefore, null hypothesis (X_0) is rejected, which in turns accepts the alternative (X_1).

Hence, it can be verified that the mean time taken to comprehend a source code with proposed visualization can reduce the duration comparing to the existing visualization EasyUML

Therefore, the reduction of duration through proposed visualization can be statistically verified over one of the current UML diagram generators namely EasyUML, such that users' comprehension level becomes easy with reference to the proposed visualizer. Hence, the usage of proposed visualizer to comprehend the source code logic can result a significant reduction of the comprehension effort comparing to the usage of source code without any diagrams and the usage of current UML diagram generators. Therefore, using a proper visualization technique is proven to assist with users' comprehension level to mitigate the difficulties occurred in the comprehension process and to achieve a less cognitive complexity. Furthermore, the concept of referring to the logical diagrams to understand the source code logic in an easy manner can be verified similarly with referring to requirements as well. As such, the issues related with understanding a source code can be lessened by navigating to their requirements and visual representations, which assists to reduce the cognitive effort and thereby to ensure a less cognitive complexity.

4.4 Evaluation of Defects Tracing Component

The verification of cognitive complexity reduction through defects tracing component has been introduced to reduce the cognitive load dedicated for defects handling, which assists to reduce unnecessary cognitive load with respect to source code logic. Accordingly, it is expected to expand the capacity dedicated for necessary cognitive load, which can assist with the actual program understanding. Hence, the reduction of the cognitive effort can be observed to achieve a less cognitive complexity. In order to check the variation of duration associated with this component, an experiment of maintaining a source code according to a new requirement has been performed. The same user groups were given source codes implemented for another thirty different software and requested to apply a modification for an existing method. It is evident that applying a modification to an existing functionality requires the logical comprehension entire source code. The source codes were inputted with coding defects listed under Table 13 and given them to users to apply the modification. The users have not been informed about the possibility of having coding defects inside given source codes, so that handling defects has to be performed along with the logical comprehension to apply the said modifications. The durations acquired by each user group to handle the identified defects through FindBugs plugin and to apply the correct modification have been recorded. The same procedure has been performed after another week through the assistance of defects tracing component and recorded the relevant duration. Moreover, the duration taken to apply the modifications without any bug tracker has not been recorded, since there is no possibility of checking the success of modifications and the expected outputs without realizing and handling the coding defects. It should be noted that the mean duration of applying correct modification has been taken in the scenarios of incorrect modifications, as it has been already followed in previous components' duration recording. Along with these, the average duration taken by each user group for both scenarios can be viewed in Figure 21.

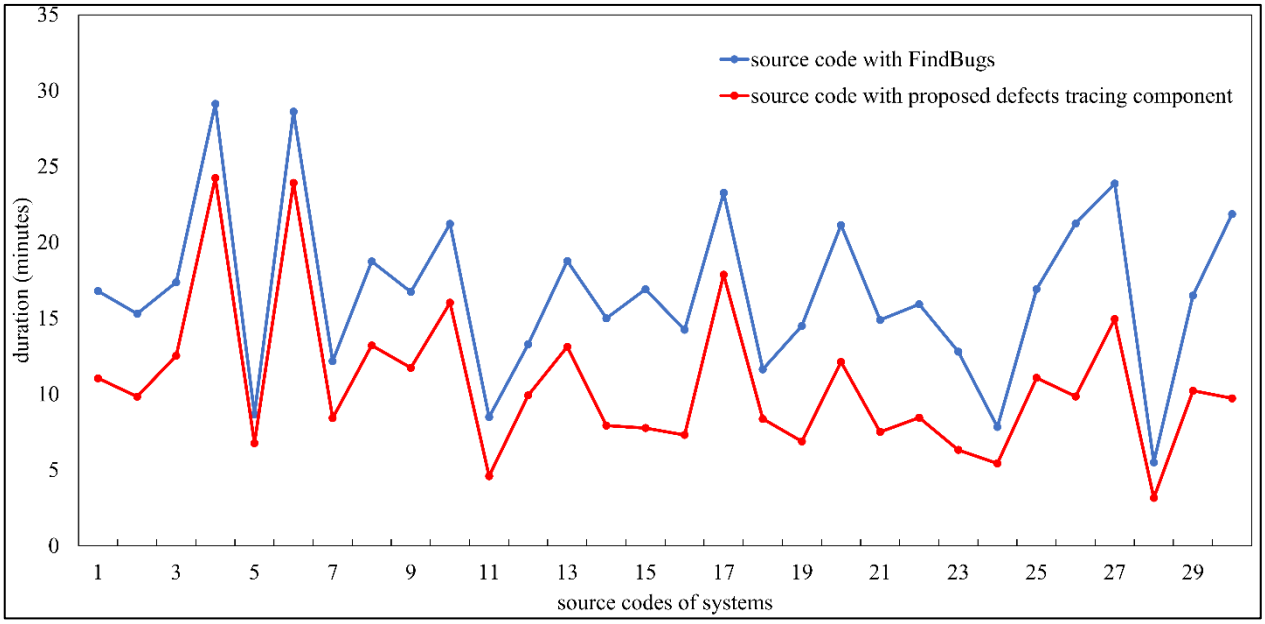


Figure 21. Average Duration with Defects Tracing Component

The Figure 21 illustrates that the total average duration required to apply the correct modification with FindBugs defects tracer is 16.65 minutes, while the total average duration for same processing with the proposed defects tracing component is 10.68 minutes. As such, 35.84% duration reduction can be observed comparing to the processing through proposed component over the current bug trackers. The reason for attaining a significant declination of duration can be stated as the background created by the proposed component to handle coding defects which are not detected by FindBugs tracking tool and the guidance provided to fix the defects, so that users' comprehension effort of handling those defects reduces.

Further, the possibility of gaining a duration reduction through this component has been analyzed statistically with respect to FindBugs bug tracker using paired samples T test. Its output can be observed in Table 20.

Table 20. Output of Paired Samples T Test conducted for Defects Tracing Component

Paired Samples T Test

		Paired Differences					t	df	Sig. (2-tailed)
		Mean	Std. Deviation	Std. Error Mean	95% Confidence Interval of the Difference Lower	Upper			
Pair 1	Duration_Of_Source_Code_With_FindBugs - Duration_of_Source_Code_With_Defects_Tracing_Component	5.96608	3.18463	.29072	5.39044	6.54173	20.522	119	.000

According to the outcome of Table 20, 95% confidence status can be applied for the existence of true mean difference with defects tracing component which lies in between 5.39044 and 6.54173 minutes. The hypothesis testing performed to observe the duration variation is as follows.

Let $\gamma 1$ be the total mean time taken for source code comprehension with FindBugs bug tracker

Let $\gamma 2$ be the total mean time taken for source code comprehension by using the proposed defects tracing component

The null hypothesis (B_0) is defined as, B_0: $\gamma 1 <= \gamma 2$

The alternative (B_1) is defined as, B_1: $\gamma 2 < \gamma 1$

Rejection region (p value approach) of B_0

p value < significance level (α)

p value is the value under Sig. (2-tailed) in generated output. This test is generated as 1 tailed due to the comparison process performed for different means, however the result is generated for 2 tailed. Therefore, p value should be obtained as Sig. (2-tailed) /2.

$0.000/2 < 0.05$

$0.000 < 0.05$ → condition is true

Therefore, B_0 is rejected, which in turns accepts B_1.

Hence, it can be verified that the mean time taken to comprehend a source code with proposed defects tracing component can reduce the duration comparing to the existing bug tracker FindBugs

Based on both types of analysis, it can be significantly observed that the logical comprehension of a source code effects with the coding defects inside it. Therefore, it influences with comprehension duration such that the reduction of comprehension duration can be achieved by having a proper defects tracing mechanism. To describe it further, the number of defects that a bug tracker can identify can changes the cognitive load, effort and the time. Consequently, the usage of proposed component can be verified to achieve a less comprehension effort comparing to the existing bug trackers such as FindBugs. Therefore, the reduction of cognitive complexity can be achieved through this component by creating the possibility to reduce unnecessary cognitive load effected with defects handling so that the source code logical understanding process becomes easier.

4.5 Evaluation of Code Quality Optimization and Refactoring Components

The other components introduced to reduce unnecessary cognitive load to achieve a lesser cognitive complexity are code quality optimization and refactoring components. As it has been already described, code quality issues are the symptoms of bad design of the source code, while refactoring techniques assist to improve the design of a source code. Hence, identifying code quality issues and applying refactoring techniques support to build a proper structure of a source code. The proposed system has two components to handle code quality optimization and refactoring, and their assistance to build a source code in a structured manner to reduce the cognitive complexity has been explored. The sample source codes generated for thirty different software have been tested with users in four groups to apply a modification to an existing function by understanding the source code logic. Firstly, they were given the source code implemented through NetBeans, so that they can use existing facilities available in the IDE to restructure the source code and continue the comprehension to apply the required modifications. Then, they have been asked to repeat the same process after one week duration with the aid of system components. The duration taken by users to correctly modify the source code have been recorded for both scenarios. Although a source code can be understood and modified without applying any refactoring technique, the feature of highlighting the possible code smells through the IDE cannot be ignored in the process of modifying the source code and execute. Therefore, the duration taken to understand the source code logic without any code smell identifier and refactoring component have not been considered for the analysis. Similarly in other components,

the average duration to obtain the correct outcome has been considered in the situations of incorrect results obtaining. Consequently, the average durations obtained by each user group for each source code have been plotted in Figure 22.

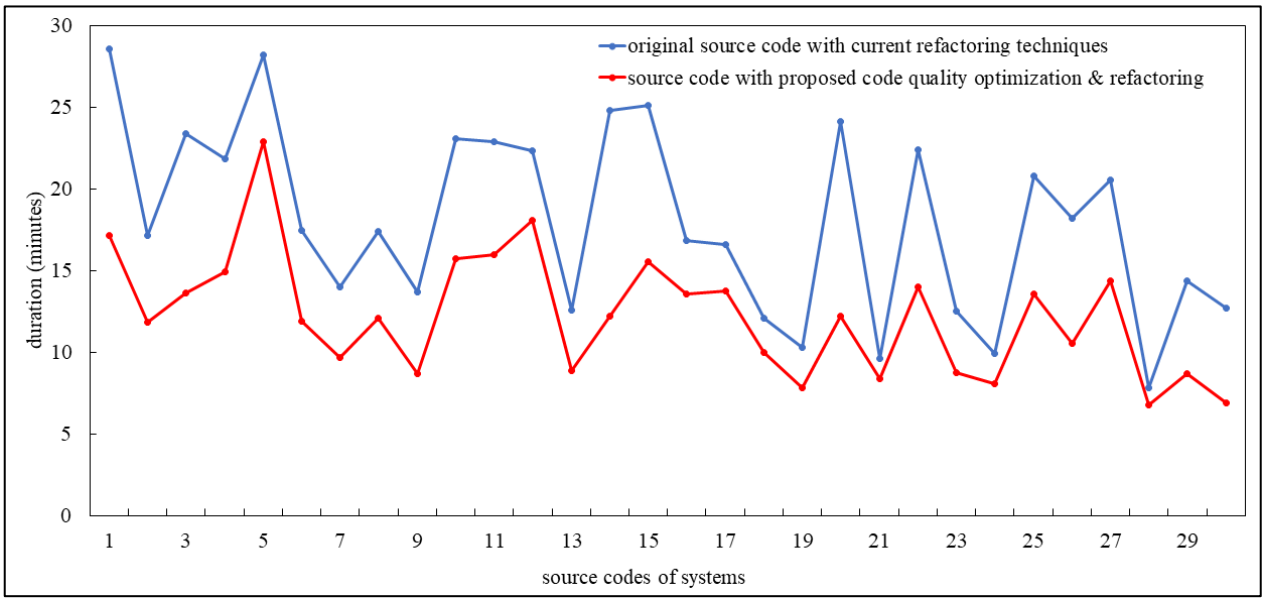


Figure 22. Average Duration with Code Quality Optimizer and Refactoring Component

According to Figure 22, the average total duration to comprehend the source code with available code quality solutions and refactoring techniques shows 18.05 minutes, while the same source code comprehension with proposed code quality optimizer and refactoring component outputs 12.22 minutes as the total average duration. Accordingly, a duration reduction of 32.29% can be observed in the proposed component over to the exiting features of handling code issues and refactoring techniques. As such, the declination towards the comprehension effort from the proposed component can be highlighted for the quality issues and refactoring techniques that the proposed components are capable of handling. The reason behind the duration reduction of proposed components is the procedure of giving suitable guidelines to fix the issues in a way that users can understand them easily, so that cognitive load and effort allocated with fixing quality issues and applying refactoring techniques are reduced comparing to the existing techniques of handling the same. Moreover, auto fixing feature introduced by code quality optimizer to fix some of the code quality issues results to lessen the cognitive load effected with fixing those, thereby to attain a high cognitive load capacity for the source code logic comprehension, which can reduce the cognitive effort. Therefore, the necessity of maintaining a proper structure of a

source code can be verified to make the logical understanding of a source code in an easy manner to reduce the cognitive complexity effected with it. Moreover, the applicability of a proper code quality optimizer and refactoring components has been indicated to further reduce the cognitive complexity with respect to the way that they have been handled. The statistical verification of the duration reduction of proposed components comparing to the available code quality and refactoring techniques using paired samples T test has been shown in Table 21.

Table 21. Output of Paired Samples T Test conducted for Code Quality Optimizer and Refactoring Components

Paired Samples T Test

		Paired Differences					t	df	Sig. (2-tailed)
		Mean	Std. Deviation	Std. Error Mean	95% Confidence Interval of the Difference Lower	Upper			
Pair 1	Source_Code_With_Current_Code_Quality_And_Refactoring_Techniques - Source_Code_With_Proposed_Quality_Optimization_And_Refactoring	5.82817	3.73234	.34071	5.15352	6.50282	17.106	119	.000

Table 21 summarizes of having 95% confidence interval of true mean differences of durations taken for source code comprehension with current quality and refactoring techniques and with proposed component between 5.15352 and 6.50282. Moreover, the hypothesis testing to observe the duration declination of proposed component is as follows.

Let $\eta 1$ be the total mean time taken for source code comprehension with current code quality and refactoring techniques

Let $\eta 2$ be the total mean time taken for source code comprehension by using the proposed code quality optimizer and refactoring component

The null hypothesis (Y_0) is defined as, Y_0: $\eta 1 <= \eta 2$

The alternative (Y_1) is defined as, B_1: $\eta 2 < \eta 1$

Rejection region (p value approach) of B_0

p value < significance level (α)

p value is the value under Sig. (2-tailed) in generated output. This test is generated as 1 tailed due to the comparison process performed for different means, however the result is generated for 2 tailed. Therefore, p value should be obtained as Sig. (2-tailed) /2.

$0.000/2 < 0.05$

$0.000 < 0.05$ → condition is true

Therefore, Y_0 is rejected, which in turns accepts Y_1.

Hence, it can be verified that the mean time taken to comprehend a source code with proposed quality structuring and refactoring components can reduce the duration comparing to the existing quality and refactoring techniques

Through the statistical analysis also, the duration reduction of the proposed component has been mentioned as true over the currently available techniques of solving code quality issues and refactoring techniques. In addition to that, the necessity of maintaining a proper structure of a source code is proven to have a less comprehension duration, which tends to make the comprehension process easier. Herein, including the features to solve code quality issues and to apply refactoring techniques can be stated to lessen the cognitive complexity effected with it.

After analyzing the durations obtained through all the components of proposed system, a significant reduction of the time taken to understand a given source code logic to process a certain task can be observed comparing to the current practices. As the duration has been expressed and verified as a direct parameter of determining the comprehension effort, a lesser duration has implied a lesser comprehension effort taken by a particular individual. So that, the

system components have been evaluated to obtain a cognitive complexity reduction, which are associated with software development and maintenance processes.

4.6 Evaluation of Cognitive Weightage Assignment for BCS

This has been conducted to overcome the issues related with current cognitive weights' validity, as majority of weightage assignments are based on experience related assumptions. Therefore, the weightage assignment process for BCS has been carried out through an experimental background. The expectation is to analyze the comprehension level associated with users in terms of marks and duration taken for the questionnaire. Since the cognitive weights are determined for BCS, the questions of the questionnaire have been categorized based on the type of BCS. As such, questions based on *if-else* and *switch-case* under selection category, *for* loop and *while* loop under looping category and *nested for* and *nested while* in nested looping category were included into the questionnaire. The same question was assessed through different BCS under same category, to compare the comprehension levels associated with different BCS under same category. The questionnaire was conducted in Moodle environment to accurately monitor the marks and the duration taken. The statistical data of the questionnaire has been obtained through SPSS. The statistical outcomes derived for all BCS categories have been included in Appendix C. Then, each category has been analytically studied based on mean and standard deviation of marks (out of 5) and durations (minutes) and compared to obtain the BCS which is easy to understand in each category. The data which has been obtained for each BCS are listed in Table 22.

Table 22. Statistical Values Obtained from BCS Questionnaire

BCS Category	*BCS*	*Evaluation criteria*	*Mean*
conditional	if	time(minutes)	6.8377
		marks (out of 5)	4.63
	switch-case	time(minutes)	3.4622
		marks (out of 5)	4.71
looping/iteration	for	time(minutes)	6.3616
		marks (out of 5)	4.2960
	while	time(minutes)	4.0950
		marks (out of 5)	4.3640
nested looping/iteration	nested for	time(minutes)	8.1471
		marks (out of 5)	3.6920
	nested while	time(minutes)	5.8110
		marks (out of 5)	3.9260

If a BCS can be easily understood, the duration taken for it becomes less and there is a high possibility that user can reach to the correct answer. Therefore, the BCS which obtain comparatively less durations and high marks reflect for high comprehension level, so that they should be assigned with less cognitive weights. Similarly, higher durations and less marks tend to obtain BCS with less comprehension level, which should be assigned with high cognitive weights.

By applying this concept along with Table 22 results, it can be clearly observed that high mean marks have been obtained with less mean duration in *switch-case* statements, which implies *switch-case* statement has high understandability comparing to *if-else* conditional statements under selection category. However, this situation may contradict with real scenarios as *if-else* statement can be seen in majority of applications than *switch-case* statements for conditional checking. For looping category, *while* loops tend to be more understandable, since it scores more mean marks within less mean duration with respect to mean values obtained for *for* loop. Similarly, *nested while* loops result with lesser comprehension effort with high means marks and less mean duration. Therefore, *switch-case* statement, *while* loop and *nested while* loop tend to be more comprehensive BCS with respect to selection, looping and nested looping categories

respectively, which their cognitive weights should be assigned as lesser than the other BCS in same category. Then, the necessity of denoting the comprehension level of each BCS through one value has been raised to make the comparison and assignment processes conveniently. As there are two parameters considered for the comprehension level determination, the multiplication of mean duration and mean marks has been considered as the parameter to evaluate each BCS with weightage assignment. The resulted mean values for both duration and marks, mean multiplication values and proposed cognitive weights are listed in Table 23.

Table 23. Proposed Cognitive Weights for BCS

BCS	*Mean (Total Time – minutes)*	*Mean (Total Marks – out of 5)*	*Mean (Time) * Mean (Marks)*	*Proposed Cognitive Weight*
if – else	6.8377	4.63	31.67	1.94
switch - case	3.4622	4.71	16.31	1
for	6.3616	4.2960	27.33	1.68
while	4.0950	4.3640	17.87	1.1
nested for	8.1471	3.6920	30.08	1.84
nested while	5.8110	3.9260	22.81	1.4

To assign the cognitive weights, the multiplicated mean values have been compared with respect to the minimum multiplicated mean value. The minimum multiplicated mean value is 16.31, which has been computed from *switch-case* statements. Accordingly, *switch -case* statement has been assigned to 1 as its cognitive weightage, considering 1 as the lowest cognitive weight. The other categories have been assigned by comparing with the lowest multiplicated mean (16.31) obtained for *switch-case* statement. Consequently, the ratio of other BCS multiplicated mean values in terms of *switch-case* multiplicated mean has been introduced as the cognitive weights each BCS. The resulted cognitive weights computed for each BCS have also been listed in the last column of Table 23.

As the next process, an existence of a relationship between the new cognitive weights and previous research works has been checked, even though they have been assigned through assumptions. Accordingly, the cognitive weights obtained through the questionnaire and past research works have been used as listed in Table 24.

Table 24. Comparison of Proposed Cognitive Weights with Previous Cognitive Weights

BCS	*Research* [35]	*Research* [19]	*Research* [25]	*Research* [30]	*Computed Weights from the Questionnaire*
if - else	1	2	2	2	1.94
switch - case	n (n cases)	3	3	2	1
for loop	2	3	3	3	1.68
while loop	2	3	3	3	1.1
nested for loop	n (n levels)	4	not discussed	3	1.84
nested while loop	n (n levels)	4	not discussed	3	1.4

It can be clearly observed that the previous cognitive weightages are bounded with whole numbers, while cognitive weights obtained from questionnaire dataset lies within 1-2 region with decimal values as well. The previous works has not introduced the decimal values on a basis of easy usage, comparison and computation, but it has led to unnecessary large quantifications for the complexity. Both of [19] and [25] has assigned 2 and 3 as the cognitive weightages for *if – else* and *switch – case* statements respectively, which *if – else* statement tend to be more comprehensive than *switch-case* statement. Hence, it has verified the practical situation of typical users to prefer more on *if-else* statements. Further, the weightage assignment of [35] can also be acceptable due to the higher weightage allocation for *switch-case* statement based on the number of cases associated with it. Nonetheless, the weightage allocation in [30] tends to be more problematic due to consideration of same cognitive weight allocation for both *if-else* and *switch-case* statements, which indicates the same comprehension level for both BCS. According to the

questionnaire dataset, *switch–case* statement tends to be more comprehensive than *if–else* statement, which is a complimentary observation with respect to other research works. The cognitive weights assigned for both *for* and *while* loops tend to be equal in literature, while weight of 2 has been assigned in [35] and others with weight 3. However, the comprehension level of a *for* loop and a *while* loop cannot be equal, so that both loops have to be assigned with different cognitive weights. Furthermore, any user is preferred to use one of the looping controllers as the preferred looping structure, which demonstrates the level of understandability and the comprehension effort of both loops have to be varied. Therefore, the cognitive weightage assignment in previous research works seem to be erroneous. The weightages assigned for both types of loops with questionnaire analysis seems to be quantitatively different such that *while* loops seem to be more cognitive than *for* loop. Therefore, it is a significant achievement to demonstrate the user cognition level and preference variations in a quantitative manner. The comprehension effort of nested looping categories has not been discussed in [25]. The same cognitive weightage, which is equivalent to 4 in [19] and n in [35] have been assigned to both *nested for* and *nested while* loops, which concludes the level of understandability of both has to be unchanged. It is practically unaccepted since different users has different comprehension efforts for different looping criteria, which has been discussed under the single looping criteria as well. Therefore, same weightage assignment for nested looping structure in [35] and [19] cannot be validated. The analysis of the questionnaire dataset introduces different cognitive weights for both nested looping structures such that *nested while* loop makes more understandable than *nested for* loop. Further, *while* loop and *nested while* loop have obtained a high level of understandability among single and nested looping criteria respectively, which signifies the accuracy of the test results as the low comprehension level associated with *while* loop should be continued to its *nested while* loop.

Accordingly, the applicability of majority of proposed cognitive weights can be stated as valid in terms of weightage assignment procedure and user preference in practical scenarios. However, these weightages have been generated based on a specific user group, so that their applicability is limited only for that user group. Thereby, the usage of proposed cognitive weights for general usage has to be performed, which is applicable for entire user population. As such, the mean values obtained for duration and marks of the questionnaire have been statistically analyzed to check the possibility of applying the same weights to general users. Since each BCS in same category has been tested using same user group, the paired samples T test has been performed for

both mean duration and mean marks obtained through each BCS. It should be noted that the confidence interval has been taken as 95% for the test. Through the test analysis, the applicability of the variations of mean duration and mean marks have been observed for entire group. If the same variation of a particular BCS can be verified to be applicable for the entire user group, the proposed cognitive weight can be stated as valid. Hence, paired samples T test has been conducted to verify the cognitive weights of *if-else* and *switch-case* statements in terms of mean duration and mean marks and the outputs can be viewed in Table 25 and Table 26 respectively.

Table 25. Output of Paired Samples T Test conducted for *if-else* and *switch-case* statements based on Time

Paired Samples T Test

	Paired Differences					t	df	Sig. (2-tailed)
	Mean	Std. Deviation	Std. Error Mean	95% Confidence Interval of the Difference Lower	Upper			
Pair 1 If_Total_Time - Switch_Total_Time	3.37557	6.56862	.29376	2.79841	3.95272	11.491	499	.000

Table 26. Output of Paired Samples T Test conducted for *if-else* and *switch-case* statements based on Marks

Paired Samples T Test

	Paired Differences					t	df	Sig. (2-tailed)
	Mean	Std. Deviation	Std. Error Mean	95% Confidence Interval of the Difference Lower	Upper			
Pair 1 If_Mark - Switch_Mark	-.082	1.155	.052	-.183	.019	-1.588	499	.113

Based on Table 25, the hypothesis testing for mean duration variation of *if-else* and *switch-case* statements for general users has been performed as follows. Although the questionnaire results imply more comprehension on *switch-case* statements, the impossibility of it with the current user preferences and previous cognitive weightage assignments has already been highlighted.

Therefore, the general expectation of deriving *if-else* statement as more comprehensive BCS has been analyzed through the test.

Let a1 be the total mean time taken to comprehend *if-else* statements

Let a2 be the total mean time taken to comprehend *switch-case* statements

The null hypothesis (T_0) is defined as, T_0: a1 >= a2

The alternative (T_1) is defined as, T_1: a1 < a2

Rejection region (p value approach) of T_0

p value < significance level (α)

p value is the value under Sig. (2-tailed) in generated output. This test is generated as 1 tailed due to the comparison process performed for different means, however the result is generated for 2 tailed. Therefore, p value should be obtained as Sig. (2-tailed) /2.

0.000/2 < 0.05

0.000 < 0.05 → condition is true

Therefore, T_0 is rejected, which in turns accepts T_1.

Hence, it can be verified that the mean time taken to comprehend a source code with *if-else* statements is lesser than the same source code with *switch-case* statements as per general users.

Similarly, the existence of high mean marks for *if-else* statements for general users has been analyzed by a hypothesis testing with the usage of statistical results listed in Table 26 as follows.

Let b1 be the total mean marks taken to comprehend *if-else* statements

Let b2 be the total mean marks taken to comprehend *switch-case* statements

The null hypothesis (R_0) is defined as, R_0: b1 <= b2

The alternative (R_1) is defined as, R_1: b1 > b2

Rejection region (p value approach) of R_0

p value < significance level (α)

p value is the value under Sig. (2-tailed) in generated output. This test is generated as 1 tailed due to the comparison process performed for different means, however the result is generated for 2 tailed. Therefore, p value should be obtained as Sig. (2-tailed) /2.

0.113/2 < 0.05

0.0565 < 0.05 → condition is false

Therefore, R_0 cannot be rejected.

Hence, the possibility of gaining less marks for if-else statements can be emphasized in terms of general usage.

Based on the hypothesis testing performed for comprehension level analysis of *if-else and switch-case* statements, a less mean duration has been observed with *if-else*, while a high mean mark has been observed with *switch-case* statements. However, the evaluation scenario of the questionnaire is to obtain the BCS which can obtain high marks within less duration to verify its comprehensibility over the other BCS, since the comprehensibility should be associated with correct logic identification. Surprisingly, the results of hypothesis testing do not imply that situation, as these two conditions are mapped with two BCS. Thereby, the impossibility of generalizing a cognitive weight for a BCS in selection category can be highlighted.

Then, the process of generalizing cognitive weights for looping criteria has been performed. Accordingly, outcomes of the paired samples T test for *for* loop and *while* loop with respect to mean time and mean marks have been listed in Table 27 and Table 28 respectively.

Table 27. Output of Paired Samples T Test conducted for *for* and *while* loops based on Time

Paired Samples T Test

	Paired Differences					t	df	Sig. (2-tailed)
	Mean	Std. Deviation	Std. Error Mean	95% Confidence Interval of the Difference Lower	Upper			
Pair 1 For_Total_Time - While_Total_Time	2.26657	5.46020	.24419	1.78680	2.74633	9.282	499	.000

Table 28. Output of Paired Samples T Test conducted for *for* and *while* loops based on Marks

Paired Samples T Test

	Paired Differences					t	df	Sig. (2-tailed)
	Mean	Std. Deviation	Std. Error Mean	95% Confidence Interval of the Difference Lower	Upper			
Pair 1 For_Marks - While_Marks	-.06800	1.41824	.06343	-.19261	.05661	-1.072	499	.284

The hypothesis testing conducted to observe the mean duration variation of looping criteria using Table 27 outcomes can be observed below. As per the questionnaire results, the influence of *while* loop over *for* loop has been considered as more comprehensive in hypothesis testing.

Let c1 be the total mean time taken to comprehend *for* loop

Let c2 be the total mean time taken to comprehend *while* loop

The null hypothesis (Q_0) is defined as, Q_0: c1 <= c2

The alternative (Q_1) is defined as, Q_1: c2 < c1

Rejection region (p value approach) of Q_0

p value < significance level (α)

p value is the value under Sig. (2-tailed) in generated output. This test is generated as 1 tailed due to the comparison process performed for different means, however the result is generated for 2 tailed. Therefore, p value should be obtained as Sig. (2-tailed) /2.

0.000/2 < 0.05

0.000 < 0.05 → condition is true

Therefore, Q_0 is rejected, which in turns accepts Q_1.

Hence, it can be verified that the mean time taken to comprehend a source code with *while* loops is lesser than the same source code with *for* loops as per general users.

Consequently, the analysis from the questionnaire results can be stated as true in terms of general users such that the comprehension duration for *while* loop is lesser than *for* loop. Similarly, same scenario applied for mean marks has been considered through another hypothesis testing referred to Table 28, since the comprehension effort should be allocated to identify the logic correctly. Therefore, the possibility of maintaining high mean marks for *while* loop for general users has been considered in hypothesis testing as follows.

Let d1 be the total mean marks taken to comprehend *for* loop

Let d2 be the total mean marks taken to comprehend *while* loop

The null hypothesis (S_0) is defined as, S_0: d1 >= d2

The alternative (S_1) is defined as, S_1: d2 > d1

Rejection region (p value approach) of S_0

p value < significance level (α)

p value is the value under Sig. (2-tailed) in generated output. This test is generated as 1 tailed due to the comparison process performed for different means, however the result is generated for 2 tailed. Therefore, p value should be obtained as Sig. (2-tailed) /2.

0.284/2 < 0.05

0.142 < 0.05 → condition is false

Therefore, S_0 cannot be rejected.

Hence, the possibility of gaining less marks for *while* loops can be emphasized in terms of general usage.

Similarly in conditional statements, looping criteria has also derived that the comprehension duration allocated for *while* loop is lesser with less marks, which is not the exact requirement for an accurate logical comprehension. Thereby, it is impossible to define the proposed cognitive weights for looping criteria to be used in terms of general users.

As the next process, the possibility of applying the proposed cognitive weights for nested looping criteria to use be used generally has been tested in terms of mean duration and mean marks. The outputs of T test for nested for and nested while loops are shown in Table 29 and Table 30 respectively.

Table 29. Output of Paired Samples T Test conducted for *nested for* and *nested while* loops based on Time

Paired Samples T Test

		Paired Differences					t	df	Sig. (2-tailed)
		Mean	Std. Deviation	Std. Error Mean	95% Confidence Interval of the Difference Lower	Upper			
Pair 1	NestedFor_Total_Time - NestedWhile_Total_Time	2.33610	8.45668	.37819	1.59305	3.07915	6.177	499	.000

Table 30. Output of Paired Samples T Test conducted for *nested for* and *nested while* loops based on Marks

Paired Samples T Test

		Paired Differences					t	df	Sig. (2-tailed)
		Mean	Std. Deviation	Std. Error Mean	95% Confidence Interval of the Difference Lower	Upper			
Pair 1	NestedFor_Marks - NestedWhile_Marks	-.23400	1.90118	.08502	-.40105	-.06695	-2.752	499	.006

Based on Table 29, a hypothesis testing has been performed to observe the mean duration variation among nested looping criteria. Similarly in single looping criteria, the questionnaire results of maintaining *nested while* loop as more comprehensive has been considered such that *nested while* loop should have less comprehension duration over *nested for* loop for general usage. Accordingly, the hypothesis testing performed in terms of duration can be observed below.

Let h1 be the total mean time taken to comprehend *nested for* loop

Let h2 be the total mean time taken to comprehend *nested while* loop

The null hypothesis (I_0) is defined as, I_0: h1 <= h2

The alternative (I_1) is defined as, I_1: h2 < h1

Rejection region (p value approach) of I_0

p value < significance level (α)

p value is the value under Sig. (2-tailed) in generated output. This test is generated as 1 tailed due to the comparison process performed for different means, however the result is generated for 2 tailed. Therefore, p value should be obtained as Sig. (2-tailed) /2.

0.000/2 < 0.05

0.000 < 0.05 → condition is true

Therefore, I_0 is rejected, which in turns accepts I_1.

Hence, it can be verified that the mean time taken to comprehend a source code with *nested while* loops is lesser than the same source code with *nested for* loops as per general users.

Based on the outcome of the hypothesis test, *nested while* loop tends to be more comprehensive than *nested for* loop in terms of comprehension duration for general usage. Further, another hypothesis test has been performed to evaluate the accuracy level obtained for nested looping due to the consideration of accurate comprehension. Since *nested while* loop implies high comprehensibility, its accuracy over *nested for* loop has been tested through the hypothesis testing using the values obtained from Table 30 as follows.

Let j1 be the total mean marks taken to comprehend *nested for* loop

Let j2 be the total mean marks taken to comprehend *nested while* loop

The null hypothesis (K_0) is defined as, K_0: j1 >= j2

The alternative (K_1) is defined as, K_1: j2 > j1

Rejection region (p value approach) of K_0

p value < significance level (α)

p value is the value under Sig. (2-tailed) in generated output. This test is generated as 1 tailed due to the comparison process performed for different means, however the result is generated for 2 tailed. Therefore, p value should be obtained as Sig. (2-tailed) /2.

0.006/2 < 0.05

0.003 < 0.05 → condition is true

Therefore, K_0 is rejected, which in turns accepts K_1.

Hence, it can be verified that the mean marks taken to comprehend a source code with *nested while* loops are higher than the same source code with *nested for* loops as per general users

Hence, the usage of *nested while* loop has shown more understandability than *nested for* loop in general applications, as it has been verified from both time and marks obtained to imply a correct comprehension. Therefore, the applicability of the proposed cognitive weights for nested looping category can be stated. Nevertheless, the outcomes of hypothesis testing conducted for selection and single looping categories are valid only for duration variation. As the outcomes have not been verified over the marks, the problem of maintaining an accurate understandability of these BCS exists. Hence, the applicability of proposed cognitive weights to the selection and single looping categories cannot be accepted.

A situation of applying a set of cognitive weights only to nested looping category through these outcomes creates a problematic scenario as there is no cognitive weightage assignment provided to represent other BCS categories. The user comprehension level should be associated with the whole source code, which does not confine only to BCS. Therefore, the possibility of confining the assignment of cognitive weights only to BCS creates another problem. With these problems,

it can be stated that assigning cognitive weights to BCS is not the expected outcome to denote the user comprehension level numerically. As such, a new procedure of assigning the values for cognitive weights to imply the user comprehension level related with a given source code has been followed.

4.7 Evaluation of Cognitive Weightage Assignment Emphasized on Personal Profile (Cw)

The proposed cognitive weightage assignment solves the problem of restraining the user comprehension level into a certain information type of source code as followed in previous research works. Instead of it, the proposed cognitive weight is comprised with factors related to personal profile and source code factor, as the cognitive complexity has been expressed using these two main categories. This research work is described to emphasize more on personal profile, so that personal profile factors considered for cognitive weight determination is comparatively higher than source code factors. The correlation of each parameter used for cognitive weight determination has been verified using Spearman method. The heatmap, which demonstrates the correlation of each parameter of the dataset can be observed in Figure 23.

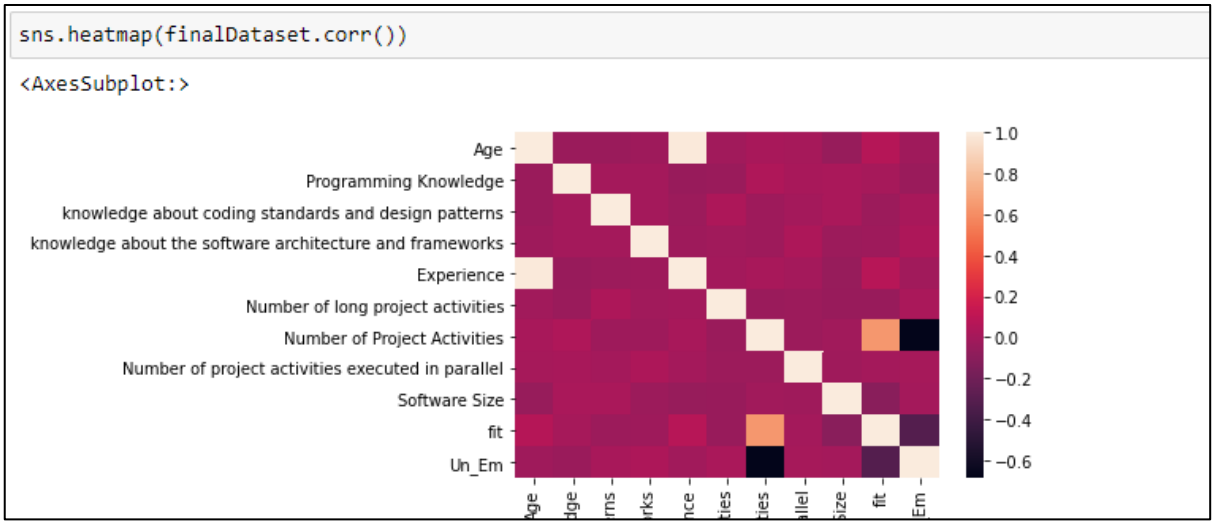


Figure 23. Heatmap Generated for the Parameters Considered for Proposed Cognitive Weight

The opportunity of rating each of cognitive weight parameters with respective to each user has been provided, and those values are verified through a questionnaire given for same user. The users' values are verified for 80% accuracy obtained through questionnaire. The verified data is

modelled and process to predict the cognitive weight for each user. The cognitive weight is predicted with quantitative approach to express it within (1-5) scale, and with qualitative approach to express the user understandability level using *high* or *low* terms. To express the cognitive weight quantitatively using (1-5) scale, Linear Regression and Logistic Regression algorithms have been used. The confusion matrix which summarizes the performance of Linear Regression algorithm can be observed in Figure 24.

```
from sklearn. metrics import classification_report

print(classification_report(Y_test, y_pred))
```

	precision	recall	f1-score	support
0	0.94	0.93	0.93	67
1	0.97	1.00	0.99	75
2	0.91	0.90	0.90	58
accuracy			0.94	200
macro avg	0.94	0.94	0.94	200
weighted avg	0.94	0.94	0.94	200

Figure 24. Confusion Matrix Generated for Linear Regression Algorithm

According to Figure 24, the Linear Regression algorithm to predict the cognitive weight quantitatively has achieved 94% accuracy. Since the same process has been performed with Logistic Regression algorithm, its accuracy and average accuracy have also been computed by generating its confusion matrix as shown in Figure 25.

```
from sklearn.metrics import classification_report
log_pred = model.predict(x_test)
print(classification_report(y_test, log_pred))
```

	precision	recall	f1-score	support
0	0.99	1.00	0.99	92
1	1.00	0.99	1.00	108
accuracy			0.99	200
macro avg	0.99	1.00	0.99	200
weighted avg	1.00	0.99	1.00	200

```
from sklearn.metrics import accuracy_score, precision_score, recall_score

print('Accuracy:', accuracy_score(y_test, log_pred))
```

```
Accuracy: 0.995
```

Figure 25. Confusion Matrix and Accuracy Generated for Logistic Regression Algorithm

Based on Figure 25, Logistic Regression algorithm provides 99% high accuracy and 99.5% high average accuracy on training data set which leads to a poor performance on testing data. Therefore, Logistic Regression algorithm leads with overfitting data as shown in Figure 26.

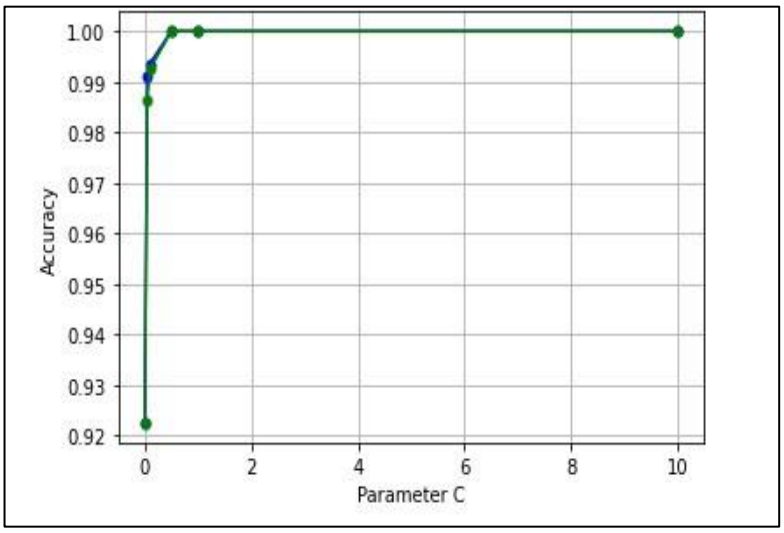


Figure 26. Overfitting Issue Caused by Logistic Regression Algorithm

The reason for this situation can be the number of parameters that have been used for data modelling. Since the majority of parameters of the data set is person biased, the reduction of parameters has not been considered as the solution. Therefore, Linear Regression algorithm has been selected to be used for cognitive weight quantification which does not show an overfitting over the training data set. To verify that, underfitting data graph of Linear Regression algorithm has been generated as shown in Figure 27.

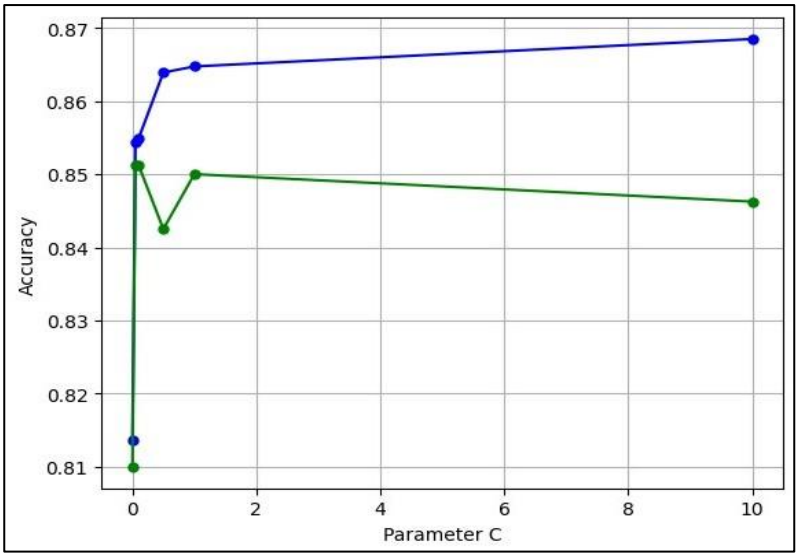


Figure 27. Underfitting Caused by Linear Regression Algorithm

According to Figure 27, Linear Regression algorithm can be stated as a better approach to be used to predict the cognitive weight using a numerical value. Then, selecting a suitable algorithm to predict the cognitive weight in a qualitative manner has been performed. Accordingly, Decision Tree and Gaussian Naïve Bayes algorithms have been used for that purpose. The accuracy and average accuracy of Decision Tree algorithm has been obtained through its confusion matrix as in Figure 28.

```
from sklearn.metrics import classification_report
log_pred = model.predict(x_test)
print(classification_report(y_test, log_pred))
```

```
              precision    recall  f1-score   support

           0       0.89      0.86      0.87        92
           1       0.88      0.91      0.89       108

    accuracy                           0.89       200
   macro avg       0.89      0.88      0.88       200
weighted avg       0.89      0.89      0.88       200
```

```
from sklearn.metrics import accuracy_score, precision_score, recall_score

print('Accuracy:', accuracy_score(y_test, log_pred))
```

```
Accuracy: 0.885
```

Figure 28. Confusion Matrix and Accuracy Generated for Decision Tree Algorithm

The accuracy and average accuracy obtained by Decision Tree algorithm is 88.5% and 89% respectively. Then, the same accuracies have been computes through Gaussian Naïve Bayes algorithm as shown in Figure 29.

```
from sklearn.metrics import classification_report
log_pred = model.predict(x_test)
print(classification_report(y_test, log_pred))
```

```
              precision    recall  f1-score   support

           0       0.89      0.92      0.91        92
           1       0.93      0.91      0.92       108

    accuracy                           0.92       200
   macro avg       0.91      0.92      0.91       200
weighted avg       0.92      0.92      0.92       200
```

```
from sklearn.metrics import accuracy_score, precision_score, recall_score

print('Accuracy:', accuracy_score(y_test, log_pred))
```

```
Accuracy: 0.915
```

Figure 29. Confusion Matrix and Accuracy Generated for Gaussian Naïve Bayes Algorithm

Hence, the accuracy and the average accuracy gained from Gaussian Naïve Bayes algorithm can be stated as 91.5% and 91% respectively. Since a higher accuracy over test data is obtained through Gaussian Naïve Bayes algorithm, it has been selected as the best algorithm to predict the cognitive weight qualitatively.

It should be noted that the qualitative cognitive weightage assignment has been introduced to build up a comparison and verification process between the quantitative cognitive weights outputted by the component. Moreover, the quantitative cognitive weight is going to be considered as one input for proposed cognitive complexity metric. Hence, more significance has been given to quantitative cognitive weights as it is easy to use and handle within the complexity computation process. As per the verification of proposed cognitive weights, they should be evaluated against the subjectivity associated with user comprehension level. In other words, for a given source code, generated cognitive weights for each user should be varied as the proposed cognitive weight is more emphasized on personal profile. However, this outcome has a significant variation with current cognitive weights as they have not been generated by considering personal profile. For the verification purpose, two sample GitLab[6] projects *sila_java* (project ID 4205706) with 509 LOC and *lox-java* (project ID 23403357) with 5153 LOC have been selected to be assigned with cognitive weights. The same group with 500 students, who were selected for BCS questionnaire was taken for this weightage assignment as well. They were requested to verify the parameters supplied from cognitive weightage component, and then the corresponding cognitive weight for each user is predicted after the users' responses have been verified again. As such, 500 cognitive weights have been generated by the component and they have been analyzed statistically to observe a variation. The statistical outcome generated by SPSS can be viewed in Table 31.

[6] https://about.gitlab.com/

Table 31. Cognitive Weights generated for Sample GitLab Projects

Descriptive Statistics

	N	Minimum	Maximum	Mean		Std. Deviation	Variance
	Statistic	Statistic	Statistic	Statistic	Std. Error	Statistic	Statistic
Cognitive_Weight_sila_java	500	1	5	2.52	.054	1.214	1.473
Cogntiive_Weight_lox_java	500	1	5	3.68	.051	1.148	1.318
Valid N (listwise)	500						

The status of changing the value of a variable can be identified by obtaining the variance of its data series. A non-zero value obtained for a dataset implies the possibility of changing their values along with the other parameter. If there is no change occurred in the data series, the variance of them becomes zero. According to the statistical analysis in Table 31, the variances of cognitive weights obtained for *sila_java* and *lox-java* by 500 users tend to be 1.473 and 1.318 respectively. Non-zero variances obtained for each source code imply the possibility of varying the cognitive weights from one user to another. Hence, the capability of demonstrating the subjectivity related with the understandability level can be verified by the proposed cognitive weight assignment. Furthermore, non-similar variances of cognitive weights generated for each source code imply the possibility of having different comprehension levels in different programs. This can be further analyzed by the frequency distribution of cognitive weights obtained by users for different source codes. The frequency distributions of cognitive weights generated by SPSS for *sila_java* has been shown in Table 32 and Figure 30 respectively.

Table 32. Frequency Distribution Table for *sila_java* Source Code

Cognitive_Weight_sila_java

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	1	120	24.0	24.0	24.0
	2	152	30.4	30.4	54.4
	3	111	22.2	22.2	76.6
	4	83	16.6	16.6	93.2
	5	34	6.8	6.8	100.0
	Total	500	100.0	100.0	

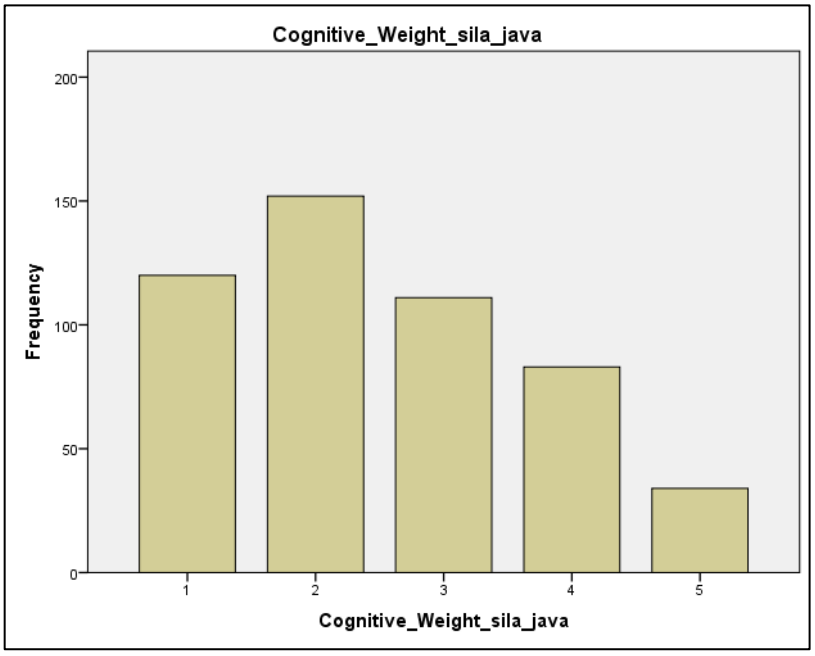


Figure 30. Frequency Distribution bar Graph for *sila_java* Source Code

According to the above cognitive weight distribution, majority of users' understandability level for *sila_java* source code lies in weight scale of 2. The weight of 2 belongs to a lesser cognitive weight category, which indicates a higher understandability level. Hence, the average understandability level of this source code tends to be in a higher state, so that it can be stated as a simple source code for majority of users. Similarly, the frequency distribution of the cognitive weights for *lox-java* source code has also been obtained through SPSS, and the results are shown in Table 33 and Figure 31.

Table 33. Frequency Distribution Table for *lox-java* Source Code

Cogntiive_Weight_lox_java

		Frequency	Percent	Valid Percent	Cumulative Percent
Valid	1	19	3.8	3.8	3.8
	2	61	12.2	12.2	16.0
	3	139	27.8	27.8	43.8
	4	125	25.0	25.0	68.8
	5	156	31.2	31.2	100.0
	Total	500	100.0	100.0	

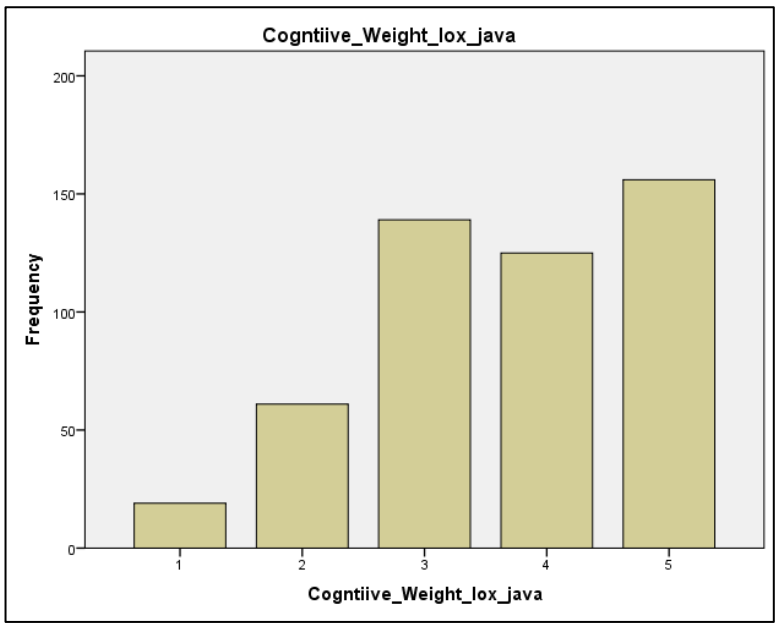


Figure 31. Frequency Distribution bar Graph for lox-java Source Code

Based on above results, the cognitive weight of 5 has been obtained by majority of users, which indicates the comprehension level of *lox-java* source code in a low state, since weight 5 is represented a higher cognitive weight category. Thereby, it can be stated as the average understandability level of this source code tends to be low such that majority of users find this source code logical comprehension process as complex. Moreover, the comprehension level decrement over the size of a source code can also be stated by considering these two scenarios. The size of *sila_java* in terms of LOC count is 509, while lox-java has 5153 LOC, in which lox-java is ten time more than the size of *sila_java*. Further, *sila_java* has denoted to be simple and lox-java tends to complex for most of users. Therefore, this situation can be taken to validate the concept of comprehension level decrement over the size of a source code, unless there are certain circumstances in which this concept cannot be applied.

Hence, the proposed cognitive weight can be specified as a way of demonstrating the effectiveness of personal profile to the understandability level of a given source code. It is capable to emphasize the factors of personal profile over source code factor to imply the subjectivity of user understandability such that cognitive weights proposed for a same source code varies with multiple users. This has not been addressed in current cognitive weightage assignments due to the emphasis given to source code factors, so that a significant reluctance of personal profile can be observed which outputs objective cognitive weights. Therefore, the mechanism of proposed cognitive weightage can be concluded as a vital achievement to highlight

the subjectivity associated with users' comprehension level, so that the subjectivity of cognitive complexity can also be validated through it.

4.8 Evaluation of the Proposed Cognitive Complexity Metric (CgC)

A cognitive complexity metric can be considered as an indicator of expressing the human comprehension effort in a quantitative manner. As cognitive complexity has been expressed with respect to personal profile and source code factors, the expression of its metric has also been expressed using these two main factors. The inclusion of personal profile has been addressed with proposed cognitive weight as it is mostly comprised with the factors inside personal profile. The spatial capacity under source code factor has already been included for cognitive weight determination, so that it has not been considered again for the metric computation process. The significant parameter of source code factor is its architectural aspect, which has been considered by all of current cognitive complexity metrics. As it defines the amount of information inside the source code, it has been introduced as the complexity attained by variables/attributes and BCS inside the source code. Subsequently, the new cognitive complexity metric (*CgC*) has been proposed as the addition of the proposed cognitive weight (*Cw*), complexity of variables/attributes (*DC*) and complexity of BCS (*BCSC*) as it is already derived through Equation (13.5). Even though the proposed metric can be proved in terms of the details expressed along with cognitive complexity definition, it has to be verified empirically and theoretically to prove its usage in real applications as an indicator of expressing the human comprehension effort associated with a given source code.

4.8.1 Empirical Validation of the Proposed Cognitive Complexity Metric (*CgC*)

Firstly, the proposed *CgC* measure has been verified in term of its applicability in real software applications using a practical framework. Kaner's validation framework has been used for this purpose, and it has been found that it is more practical and formal than the existing theoretical standard frameworks available for metric validation [12]. As such, the validation procedure of *CgC* measure through Kaner's framework has been performed as follows.

4.8.1.1 Practical Validation with Kaner's Framework

It consists with ten statements that the newly proposed metric should be capable of providing answers. Herein, the process of answering to those statements has been shown below.

Statement 1: Purpose of the measure

Purpose of this *CgC* measure is to indicate the cognitive complexity of a source code in terms of personal profile and source code aspects. The reason behind taking these two factors as the main factors behind cognitive complexity is its definition. The definition of cognitive complexity describes it as the amount of effort required for the source code logic comprehension. This research work is conducted to analyze the comprehension effort of a user in software development and maintenance processes, so that involvement of personal profile is mandatory factor to determine the cognitive complexity. Additionally, the comprehension effort is measured with respect to a source code implemented for a software, so that source code aspect should also be included for the cognitive complexity determination. Accordingly, the combination of both of personal profile and source code factors have been considered to express the cognitive complexity as well as for the quantitative representation through the proposed metric. Further, the metric is supposed to represent understandability and maintainability, as these two quality factors are found to be direct indicators to express cognitive complexity.

Statement 2: Scope of the measure

The proposed metric is focused to quantitatively represent the comprehension effort associated with the users involved in software development and maintenance processes. The emphasis of personal profile effecting for cognitive complexity has been highlighted over the source code aspects in this metric. Moreover, it is assumed that a source code with correct logical

implementation has been given to the user so that comprehension effort is allocated with understanding the correct logic of a software.

Statement 3: Identified attribute to measure

CgC measure is an indicator to express the user comprehension effort associated with a given source code. The user comprehension effort is mentioned as a direct indicator of expressing the user understandability and maintainability. Therefore, *CgC* measure addresses the user understandability and maintainability of a source code.

Statement 4: Natural scale of the attribute

It has been mentioned that the attribute expressed by proposed metric is user understandability and maintainability. The capability of understanding a given source code depends on one user to another resulting understandability as a subjective measurement. A scale of a subjective measurement cannot be defined so that *CgC* method does not consist with a scale.

Statement 5: Natural variability of the attribute

The user understandability and maintainability have been taken as the attributes that can be expressed from proposed cognitive complexity metric. These quality attributes are subjective measurements in which their variability cannot be defined. Moreover, there are numerous factors effected to determine understandability and maintainability over cognitive complexity so that defining a variability of it cannot be performed.

Statement 6: Definition of Metric

The *CgC* metric is used to indicate the cognitive complexity of a software, and the term cognitive complexity has already been defined as the amount of comprehension effort for a given source code by [1] [2].

Statement 7: Measuring instrument to perform the measurement

This metric computation can be performed for Java source codes and its computations procedure has been implemented in NetBeans IDE.

Statement 8: Natural scale of the metric

Since it indicates the user comprehension effort of a source code, it is difficult to express its scale as it derives with subjective measurement. However, obtaining higher values for *CgC* indicates

that the users' comprehension effort on that particular source code is high, which represents lesser understandability and lesser maintainability. Similarly, a lesser value obtained for *CgC* implies a less comprehension effort with high understandability and high maintainability. Therefore, it is recommended to maintain a source code which can attain a lesser value under *CgC* metric.

Statement 9: Relationship between the attribute and the metric value

The *CgC* metric determines the association of personal profile and source code aspects to indicate the user comprehension effort quantitatively and to express its understandability and maintainability. As it is already explained, a higher *CgC* value can direct to a lesser understandability and maintainability, while a lesser *CgC* value directs to a high understandability and maintainability.

Statement 10: Natural foreseeable side effects of using the instrument

This metric has been implemented in NetBeans IDE to facilitate the cognitive complexity of java source codes. As the CgC value is computed automatically after inputting the source code, there can be no side effects occurred in the background.

Accordingly, the proposed CgC metric is capable to describe all ten statements defined under Kaner's framework, which in turns verify its practicability. Apart from that, the process of computing the metric value for a given source code should be analyzed. Accordingly, the procedure of assigning the values for *DC* and *BCSC* for a sample source code has been illustrated as in Table 34.

Table 34. *DC* and *BCSC* Calculation for sample java source code - 1

Lime Number	*Line*	*DC*	*BCSC*
1	public class Class1 {	0	0
2	public static void main (String [] args) {	0	0
3	int n = 10, firstTerm = 0, secondTerm = 1;	3	0
4	System.out.println("Fibonacci Series till " + 5 + " terms:");	0	0
5	for (int i = 1; i <= n; ++i) {	1	1
6	System.out.print(firstTerm + ", ");	0	0
7	int nextTerm = firstTerm + secondTerm;	1	0
8	firstTerm = secondTerm;	0	0
9	secondTerm = nextTerm;	0	0
10	} //end of *for*	0	0
11	} //end of main	0	0
12	} // end of *Class1*	0	0
	Total	**5**	**1**

It should be noted that the information inside the class and main method declarations have not been considered for *DC* computation, as they are common for any type of source code. As such, the content of line 1 and 2 have been assigned as zero for both *DC* and *BCSC*. In line 3, 3 variables namely *n*, *firstTerm* and *secondTerm* have been declared and initialized, so that *DC* has been assigned with 3. Since there is no BCS involved in line 3, *BCSC* has been assigned as zero. Line 4 indicates a print statement which does not involve with any variable or BCS, so that both of *DC* and *BCSC* is assigned with zero. Line 5 indicates a *for* loop, which involves with 2 variables namely *i* and *n*. The variable *n* has been considered for *DC* calculation in line 3, so that it has been ignored for *DC* computation in line 5. Because, *DC* computes the number of variables/attributes inside a source code, so the frequency of each variable/attribute appearing inside the source code does not effect for *DC*. The variable *i* is new for the *for* loop, so that it has been considered for *DC* computation. Further, *BCSC* of line 5 is assigned with 1 as there is one *for* loop. Line 6 is a print statement, which has one variable *firstTerm.* Since it has been considered for *DC* computation in line number 3, *DC* is assigned with zero, while *BCSC* is also assigned with zero due to the unavailability of BCS in that line. Line 7 indicates 3 variables such

as *nextTerm*, *firstTerm* and *secondTerm*, in which *nextTerm* is declared for the first time. Therefore, *DC* is assigned with 1 and *BCSC* as zero, since there is no BCS involved with it. Both of line 8 and 9 have the variables which have been repeated before, so that *DC* is assigned to zero and the unavailability of BCS has led to *BCSC* as zero. Finally, both of line 10-12 do not involve with any variable/attribute or BCS, so that both of *DC* and *BCSC* have been assigned with zero.

Accordingly, *CgC* of the above source code can be computed as follows.

$CgC = Cw + \sum_{i=1}^{n} 1 + m + \sum_{i=1}^{m} k_i$ as per Equation (13.5)

The value for *Cw* should be derived from cognitive weightage assignment process for a particular individual. The total number of variables/attributes inside the source code is denoted as *n*, which is equivalent to 4 in this scenario. Since the source code contains only one *for* loop, the number of BCS is equivalent to 1, which is denoted using *m*. Further, its level is assigned with 1 as it does not involve with any nested criteria. As an exemplification, if a person obtains *Cw* as 3, *CgC* should be assigned as below.

$CgC = 3 + 5 + 1 + 1$

$CgC = 10$

Therefore, the quantitative indicator to express the comprehension effort of that particular individual results with 9.

Then, the cognitive complexity calculation process for a source code with a method and nested BCS has been computed. Firstly, the process of assigning the values for *DC* and *BCSC* has been denoted in Table 35.

Table 35. *DC* and *BCSC* Calculation for sample java source code - 2

Lime Number	*Line*	*DC*	*BCSC*
1	public class Class2 {	0	0
2	public static void main (String [] args) {	0	0
3	int mat[][] = { { 1, 2, 3, 4 },{ 5, 6, 7, 8 },{ 9, 10, 11, 12 } };	1	0
4	print2D(mat);	0	0
5	} // end of main	0	0
6	public static void print2D(int mat[][]) {	0	0
7	for (int i = 0; i < mat.length; i++) {	1	1
8	for (int j = 0; j < mat[i].length; j++)	1	1
9	System.out.print(mat[i][j] + " ");	0	0
10	System.out.println();	0	0
11	} // end of outer *for*	0	0
12	} // end of *print2D* method	0	0
13	}// end of *Class2*	0	0
	Total	**3**	**2**

As it has already explained, both of lines 1 and 2 have been assigned with zeros for *DC* and *BCSC* as they are class and main methods declarations. Line 3 indicates a two-dimensional array named as *mat,* which can be taken as one data attribute so that *DC* is set to 1. Since it does not contain any BCS, its *BCSC* is set to zero. Line 4 indicates *print2D* method calling in which its parameter is *mat.* Since it has already been considered in line 3, *DC* of line 4 is assigned to zero. Further, its *BCSC* is assigned with zero as it does not contain any BCS. Line 5 indicates the end of main method, which does not contain any type of information and BCS, so that both of *DC* and *BCSC* has been assigned with zero. Line 6 contains the method signature of *print2D* with *mat* passed as the parameter. The variable *mat* has already been considered in line 3, so its consideration in line 6 has been avoided. Line 7 has a *for* loop, in which its iterative variable is *i*. Hence, *DC* and *BCSC* set to 1 with respect to variable *i* and *for* loop. Line 8 also consists with another *for* loop, which is nested with the *for* loop in line 7. Its iterative variable is *j,* so that its *DC* is set to 1, and with the second *for* loop, *BCSC* is also assigned to 1. Both of line 9 and 10

indicate two print statements in which line 9 consists of *mat* as the parameter. Since *mat* has been considered earlier, its count is not included to *DC* in line 9. *BCSC* is line 9 is set to zero since there is no BCS involve with it. *DC* and *BCSC* of line 10 have been assigned with zero as there is no data and BCS involved with it. Line 11-13 indicate the end of *for* loop, *print2D* method and *Class2* respectively, in which its *DC* and *BCSC* are set to zero.

Consequently, assigning the values to the parameters of Equation (13.5) can be performed as follows.

$$CgC = Cw + \sum_{i=1}^{n} 1 + m + \sum_{i=1}^{m} k_i$$

In here, *Cw* is the cognitive weight introduced for an individual who refers this source code. The total number of variables/attributes (*n*) can be assigned as 3 based on the output of Table 35. The number of BCS inside the source code (*m*) can be assigned as 2 based on Table 35 outcomes. The level of the outer *for* loop is assigned with 1, and the level of inner *for* loop is assigned with 2 as it is nested with outer *for* loop. Thereby, the addition of levels generated by two *for* loops is equivalent to (1 + 2) = 3. Assume that *Cw* is gained as 2 for an individual, the value for *CgC* can be computed as follows.

$$CgC = 2 + 3 + 2 + 3$$

$$CgC = 10$$

Therefore, the comprehension effort of that source code can be expressed as 10 as a quantitative indicator to impress its cognitive complexity with respect to that individual.

As the next computation a sample source code with object-oriented concepts has been considered. The procedure of assigning values for *DC* and *BCSC* can be seen in Table 36.

Table 36. DC and BCSC Calculation for sample java source code - 3

Lime Number	*Line*	*DC*	*BCSC*
1	public class Main {	0	0
2	public void fullThrottle(){	0	0
3	System.out.pritnln(“The car is going as fast as it can!”);	0	0
4	} // end of fullThrottle method	0	0
5	public void speed(int maxSpeed){	1	0
6	System.out.pritnln(“Max speed is: “ + maxSpeed);	0	0
7	} // end of speed method		
8	public static void main(String [] args){	0	0
9	Main myCar = new Main ();	1	0
10	myCar. fullThrottle();	0	0
11	myCar.speed(200);	1	0
12	} // end of main	0	0
13	}//end of Main	0	0
	Total	**3**	**0**

According to Table 36, the sample source code does not involve with any BCS. Therefore, its *BCSC* should be equivalent to zero. Both of lines 1 and 2 indicates the class and *fullThrottle* method declaration, which does not contain any parameter. Hence, their *DC* has been assigned with zero. Line 3 consists with a print statement without any parameter passing. So that its *DC* has also been assigned with zero. The end of *fullThrottle* method is performed by line 4. Line 5 is coded with speed method implementation with one parameter called *maxSpeed*, so that *DC* value has been assigned to 1. Line 6 consists with another print statement which prints the value of *maxSpeed*. Since it has already been considered in line 5, *DC* has been assigned with zero. Since the mina methods declaration is common for any source code, *DC* of line 8 has been assigned with zero. Line 9 performs the creation of an object called *myCar* from *Main* method, so that *DC* is assigned with 1 by considering it as a data object. The *fullThrottle* method calling has been coded in line 10. Since it a parameter less method, it does not involve with *DC* computation. But, line 11 performs *speed* method calling by passing 200 as the parameter, so that *DC* has been

assigned with 1. Line 12 and 13 indicate the end of main method and Main class respectively, so that they do not contribute the compute *DC*.

As such, assigning values for *CgC* has been computed by referring to Equation (13.5).

$$CgC = Cw + \sum_{i=1}^{n} 1 + m + \sum_{i=1}^{m} k_i$$

The total number of data variables/attributes (*n*) can be obtained as 3 based on the output of Table 36. Since there is no BCS involved with this source code, the number of BCS (*m*) and nested level(*k*) of each BCS can be set to zero. Assume that a user has obtained the cognitive weight (*Cw*) as 4, so that *CgC* can be computed as follows.

$$CgC = 4 + 3 + 0 + 0$$

$$CgC = 7$$

Therefore, cognitive complexity of this source code with respect to that user can be resulted as 7.

Along with these mechanisms, the possibility of computing the cognitive complexity for a given java source code is supported by this metric. The inclusion of *Cw* as the parameter to assess the personal profile is capable of demonstrating the subjectivity associated with human understandability level, which can be considered as the predominant feature of the proposed metric. As such, possibility of expressing the cognitive complexity as a dynamic value can be highlighted, which have not been observed in previous computations. Therefore, the proposed cognitive complexity metric can be stated as more effective and practicable form of quantifying the human comprehension effort, which can be aligned with its definition as well.

4.8.2 Theoretical Validation of the Proposed Cognitive Complexity Metric (*CgC*)

Although the cognitive complexity computation is validated empirically using a set of source codes, its stability and practicability should be validated theoretically. A certain number of theoretical validation procedures have been introduced to validate software metrics. Among them, Weyuker properties, measurement theory introduced with Braind's framework have been used by majority of software complexity metrics [12], [123]. Therefore, it is expected to validate the proposed cognitive complexity metric with respect to both of these procedures. Firstly, the validation process under Weyuker properties has been performed. It has nine properties that the newly proposed complexity metric should adhere to estimate the accuracy [61]. Therefore, following these properties by a new complexity metric can lead to verify its usage to compute the software complexity in real applications. Accordingly, the verification of each property by the proposed cognitive complexity metrics has been performed as follows.

4.8.2.1 Theoretical validation with Weyuker Properties

In here, *P, Q* and *R* are stated as different source codes. |*P*| refers to *CgC* of source code *P,* while (*P;Q*) refers to the composition of *P* and *Q* source codes.

Property 1: $(\exists\, P)\ (\exists\, Q)\ (|P| \neq |Q|)$, where *P* and *Q* are two distinct source codes

The proposed *CgC* is composed with the parameters of personal profile and source code aspect. Therefore, a significant variation can be observed within personal profile and source code aspects with respect to two different source codes. To describe it furthermore, spatial capacity and architectural aspect of two different source codes are different, even the personal factor remains unchanged such that same person refers to two different source codes. Hence, *CgC* calculated for two different source codes should be different. Along with that, this property can be stated as true for *CgC* measure.

Property 2: Let *c* be a non-negative number, and then there are only finitely many source codes of complexity *c*.

All the source code implementations have limited number of classes and methods. The parameters under source code aspect for all source codes are limited such that their spatial capacities and the amount of information which is referred to architectural aspect cannot be

infinite. Therefore, there can be only finitely many methods that their cognitive complexity is equivalent to *c*. As such, this property is verified by *CgC* measure.

Property 3: There are distinct programs *P* and *Q* such that $|P| = |Q|$

This property states that there can be multiple source codes with same *CgC* value. As *CgC* is comprised with the addition of cognitive weight as the involvement of personal profile and spatial capacity, and the complexities of variables/attributes and BCS as architectural aspect, there can be certain situations that the addition of these parameters can end up with a same value. Therefore, the circumstances where two source codes can result with a same cognitive complexity cannot be ignored. Hence, the proposed *CgC* measure adheres with this property.

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

This indicates that two source codes of same functionality can result with different cognitive complexity values. This property can be directly applied with proposed measure as the parameters under source code aspects for two different programs are different although they compute the same functionality. The best exemplification can be considered with two source codes where a same functionality has been implemented using iterative and recursive methodologies. Accordingly, the spatial capacity and architectural aspect of both source codes end up with different values. Hence, *CgC* of both source codes get different. Therefore, *CgC* measure satisfy this property.

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

This property specified that the cognitive complexities of two distinct source codes P and Q are lesser than the source code which does both functionalities of P and Q. It is evident that the composition of two different source codes increase the spatial capacity and architectural aspect than the individual source codes, unless another structure is used to implement the combined logic. Therefore, it effects to *CgC* measurement even though the personal profile has kept unchanged. Hence, the possibility of gaining a higher *CgC* for a merged source code than its individual source codes can be stated. Thereby, the proposed *CgC* measure can validate this property.

Property 6:$(\exists\, P)(\exists\, Q)(\exists\, R)(\,|P| = |Q|)\&\, |P;R| \neq |Q;R|$

This property describes that if there are two source codes with same cognitive complexity composite with another distinct source code, the resultant cognitive complexities should be different. Appending another source code does not imply that its coding can be merged according to the way that it has been appeared earlier. In order to function it properly after merging, its information should be arranged in a proper way such that it effects to change the architectural aspect and spatial capacities. Therefore, two source codes with same cognitive complexity merged after another distinct source code cannot be achieved for a same cognitive complexity again. Consequently, the proposed *CgC* measurement satisfy this property.

Property 7: There are source codes *P* and *Q* such that *Q* is formed by permuting the order of the statements of *P*, and $(|P| \neq |Q|)$

If a logic of a source code should be transposed, it cannot be performed merely by reversing the order of statement executions of the original source code. It has to be performed in a way that it can produce the expected results as it is. Hence, the possibility of modifying the current amount of information and the spatial capacity can be existed. Therefore, a permutation of an order of execution of a source code effects to deviate the cognitive complexity from original source code. Thereby, proposed *CgC* measure helps to validate this property.

Property 8: If *P* is renaming of *Q*, then $|P| = |Q|$

The name of a particular source code does not have an impact for its cognitive complexity, as it is measured using personal profile and source code aspect. Therefore, the cognitive complexity can be changed with respective to the changes applied to personal profile and source code aspects. Herein, this property is also satisfied by the proposed metric.

Property9: $(\exists\, P)\,(\exists\, Q)\,(|P| + |Q|) < (|P;Q|)$ OR $(\exists\, P)\,(\exists\, Q)\,(\exists R)\,(|P| + |Q| + |R|) < (|P;Q;R|)$

If a source code is created by appending another two distinct source codes, the cognitive complexity of the composite source code tends to be higher than cognitive complexity addition of individual source codes. It is evident that the appended source code has more information and spatial capacity comparing to the individual source codes which have assisted to perform the merging. Furthermore, the increased amount of information and spatial capacity represent the combination of all information and spatial capacities of its individual source codes. However, a

certain amount of additional information and spatial capacity is required to arrange a proper functioning inside the merged source code. Therefore, it is evident that architectural aspect and spatial capacity of composite source code should be higher than the summation of those factors in their individual source codes. As such, the cognitive complexity of a merged source code tends to be higher than the cognitive complexity addition in corresponding individual source codes, which in turns satisfy this property.

Therefore, it can be stated that the proposed *CgC* measure satisfies all nine properties defined under Weyuker properties, which can be considered as a vital achievement to ensure its validity and the usage. The other procedure that the proposed metric has been validated is the measurement theory introduced through Braind's framework. It provides the definitions of representing a software system along with its modules and five properties to define the complexity of a software [63]. Herein, the satisfaction of these five properties by a new metric can imply the applicability of it to measure the complexity of a software. The same scenario has been applied to *CgC* measure such that satisfaction of these properties can denote the applicability of it to measure the cognitive complexity of a software. Accordingly, the process of evaluating the proposed *CgC* measure with five properties defined under Braind's framework is performed as follows.

4.8.2.2 Theoretical validation with Briand's Framework

The definition given by the framework to introduce a system with its modules as follows.

Definition (Representation of Systems and Modules): A system *S* is represented as a pair <*E*, *R*>, where *E* represents the set of elements of *S*, and *R* is a binary relation on *E* ($R \subseteq E \times E$) representing the relationships between *S*'s elements.

For the proposed metric *CgC*, *E* can be defined as a method inside a given source code and *R* is the control flow among one method to another. Hence, the combination of methods and their control flows can signify the whole source code *S*. Along with that, the definition provided for the complexity of a software is given below.

Definition (Complexity): The complexity of a system *S* is a function Complexity(*S*) that is characterized by non-negativity, null value, symmetry, modular monotonic and disjoint module additivity properties, which have been listed as follows.

Property complexity 1 (Nonnegative): The complexity of a system $S = <E, R>$ is nonnegative if Complexity $(S) \geq 0$.

Any source code created for a system exists with a certain cognitive complexity which is not equivalent to zero. It occurs with the existence of methods and their internal information flow. Therefore, the cognitive complexity computed for a source code should be non-negative. Herein, proposed *CgC* measure adheres with this complexity property.

Property complexity 2 (Null Value): The complexity of a system $S = <E, R>$ is null if *R* is empty. This can be formulated as: $R = \emptyset \Rightarrow$ Complexity $(S) = 0$.

This property indicates that a method with no control flow of information inside it does not consist with a cognitive complexity. A method with no information processing is inappropriate as methods are implemented to follow a certain set of instructions which process a set of information. Therefore, if a method does not include a control flow of information processing, there is no content that the user should understand inside it. However, the personal profile included with *CgC* measure should be highlighted. Even though the source code has empty methods, user should analyze the source code to comprehend it with existence of empty methods which consumes a certain effort of personal profile. Thereby, the cognitive complexity of a source code which does not have a control flow of information cannot be equivalent to zero. As such, the proposed cognitive complexity metric does not adhere with this complexity property. Furthermore, it is suggested that any source code should contain a certain cognitive complexity although it lacks with the internal information.

Property complexity 3 (Symmetry): The complexity of a system $S = <E, R>$ does not depend on the convention chosen to represent the relationships between its elements. ($S = <E, R>$ and $S^{-1} = <E, R^{-1}>$) $\Rightarrow$ Complexity(S) = Complexity(S^{-1}).

This property indicates that the cognitive complexity of a source code does not change with the alternations made for the execution order. This property contradicts with seventh Weyuker property as it denotes unequal cognitive complexities resulted with the permutation of order of execution of a source code. The proposed *CgC* is satisfied with that Weyuker property so that

satisfaction of this complexity property defined in Braind's framework cannot be existed. This can be further explained as permuting the order of execution has to be performed in a way that it can generate the expected output. Therefore, addition of extra information and spatial capacity cannot be ignored to ensure a proper functionality inside the permuted source code. Accordingly, the cognitive complexity of a permuted source code and its original source code tends to be different such that the proposed *CgC* measure does not adhere with this property.

Property complexity 4 (Module Monotonicity): The complexity of a system $S = <E, R>$ is no less than the sum of the complexities of any two of its modules with no relationships in common. If *m1* and *m2* denote two different modules, it can be defined as Complexity(S) ≥ Complexity (*m1*) + Complexity (*m2*)

This property describes that the summation of the cognitive complexities of any two distinct methods is not lesser than the cognitive complexity of its overall source code. If a source code consists with multiple methods, the cognitive complexities of all those methods are affected for the cognitive complexity of overall source code. This can be verified that the involvement of architectural aspect and spatial capacity to compute *CgC* measure, so that the involvement of both these aspects in all methods influences to the overall complexity of the source code. Therefore, the summation of cognitive complexities of a certain set of methods should be lesser than the cognitive complexity of overall source code. As such, the proposed cognitive complexity metric adheres with this property.

Property complexity. 5 (Disjoint Module Additivity): The complexity of a system $S = <E, R>$ composed of two disjoint modules *m1, m2* is equal to the sum of the complexities of the two modules such that Complexity(S) = Complexity (*m1*) +Complexity (*m2*).

This property highlights that the summation of cognitive complexities of two disjoint methods is equivalent to the cognitive complexity of overall source code, if the overall source code contains only with those two methods. Although the amount of information and the spatial capacity of those two methods should be added to form the overall source code, the association of personal profile to comprehend each of methods and the source code should be analyzed. In other words, it is difficult to express that personal profile association of these two methods is equivalent to the personal profile association of the overall source code. As an exemplification, a user can be able to understand each of methods individually regardless of understanding both of them together in the source code. In such a situation, it is difficult to express the effectiveness of personal profile

of whole source code as the addition of personal profiles in individual methods. Thereby, the proposed cognitive complexity metric cannot be applied to satisfy this complexity property.

Based on the verification process conducted with the properties defined under Briand's framework, the proposed *CgC* measure is capable of adhering only for two complexity properties, which can be stated as a poor satisfaction ratio. The rationale behind this situation is the measurement and modularity theory considered in this framework, so that the overall complexity of a source code is expressed as the summation of the complexities of its individual components. However, the possibility of assessing the user understandability level and source code factors effected with individual methods to whole source code have not been considered through this framework. Moreover, contradictory properties defined under this framework with Weyuker properties can be highlighted. As an exemplification, the symmetry property of Briand's framework contradicts with seventh Weyuker property. Further, disjoint module additivity property of Briand's framework contradicts with nineth Weyuker property. Since *CgC* measure adheres with all nine Weyuker properties, the adherence of symmetry property and disjoint module additivity property of Briand's framework cannot be accepted which has been already explained under their verification process. Thereby, a higher satisfaction ratio by Briand's framework for a new complexity metric cannot be expected if it has a higher satisfaction ratio through Weyuker properties. The same scenario can be applied with proposed *CgC* measure so that a lesser satisfaction ratio obtained by it through Briand's framework can be validated. Based on these situations, the proposed CgC measure to quantitatively imply the cognitive complexity can be theoretically validated to be used as a complexity metric in real software applications.

4.9 Evaluation of Proposed Cognitive Complexity Metric with Software Complexity Metrics

Rather than computing the cognitive complexity of a source code using proposed metric, proposed design is capable to compute the software complexity of the same source code using a set of current software complexity metrics. The expectation is to observe an existence of a relationship between both types of metrics as both of them attempt to define the user understandability of a source code. Accordingly, the complexities obtained for a same source code for both types of metrics among 500 users have been measured. As the software complexity metrics, cyclomatic complexity metric and LOC have been considered as it is widely used in practical scenarios to evaluate the software complexity [16]. Moreover, the cognitive complexity metric introduced by G. Ann Campbell [2] has also been used to compute the cognitive complexity to imply the usage of one of previous cognitive complexity metrics. Therefore, the possibility of analyzing the complexity values obtained with proposed *CgC* metric over currently available cognitive complexity and software complexity metrics can be stated. The users have been selected from the same user group used for the other analysis processes. Consequently, a source code of a sample GitLab project *sila_java* (project ID 4205706) has been used, which have already been used as a sample source code for cognitive weight determination as well. Then, the complexity values obtained for *sila_java* with respect to *CgC* metric, cognitive complexity by Campbell, cyclomatic complexity and LOC have been analyzed statistically and the results can be observed in Table 37.

Table 37. Statistics for Complexities Obtained for *sila_java* source code

Descriptive Statistics

	N	Minimum	Maximum	Mean	Std. Deviation	Variance	Skewness	
	Statistic	Statistic	Statistic	Statistic	Statistic	Statistic	Statistic	Std. Error
CgC_sila_java	500	83	87	84.52	1.214	1.473	.407	.109
Cog_Complexity_Campbell	500	25	25	25.00	.000	.000	.	.
Cyclomatic_Complexity	500	23	23	23.00	.000	.000	.	.
LOC	500	509	509	509.00	.000	.000	.	.
Valid N (listwise)	500							

According to Table 37, the variance of *CgC* measure has been obtained as 1.473, while other complexity measures' variance has become zero. The value obtained for variance implies the possibility of changing the measurement so that the values obtained for *CgC* metric can be stated as changing with respect to each user, while the other cognitive complexity metric, cyclomatic complexity and LOC values kept as unchanged. Accordingly, proposed *CgC* measure can be stated as subjective, while other complexity metrics tend to be objective. It should be noted that the reason behind CgC to demonstrate a complexity variation among each user is the usage of new cognitive weight into its calculation process. Further, by analyzing the complexity values obtained for *CgC* and Campbell's latest cognitive complexity metric, it can be clearly stated that the proposed *CgC* measure is capable of indicating subjective and quantified comprehension efforts, which cannot be observed in Campbell's metric and other available cognitive complexity metrics. Similarly, current software complexity values can also be indicated as the measurements limited to one or more quantifiable and objective software attributes, which are not capable of demonstrating users' subjective difficulty levels. Consequently, current metrics can be described as objective indicators to imply the complexity level associated with a given software in different contexts. Hence, the proposed *CgC* metric can be considered as a vital achievement to demonstrate the subjectivity related with users' comprehension level in a form of metric, which can be considered as a proper indicator of represent cognitive complexity. Since the above complexity computation has been performed for a single source code, the complexity computation using the same set of complexity metrics has been performed for another set of source codes to observe the variation among all metric values. Accordingly, another ten GitLab Java projects have been selected to be computed again these metrics. As such, the computed complexity values from *CgC*, cognitive complexity by Campbell, cyclomatic complexity and LOC can be observed in Table 38. It should be noted that the average complexity values have been computed for all the metric values to make it easier to demonstrate the complexity of 500 users.

Table 38. Complexities Obtained for Sample GitLab Projects

GitLab Project Name (Project ID)	*Cyclomatic Complexity*	*LOC*	*Cognitive Complexity by Campbell [2]*	*Average CgC*
sila_java (4205706)	23	509	25	84.518
skytec-test-java (38828805)	6	476	28	75.21
Java (12958040)	1	19	12	62.08
Simple Java (28693061)	1	29	59	75.23
gemseo-java (31201354)	7	623	22	63.61
lox-java (23403357)	65	5153	102	88.61
websitecarbon-java (24783461)	23	642	36	58.79
Approvals Java (25989825)	15	745	53	76.55
java-gradle (8368700)	1	88	23	28.79
java-threads (12257822)	8	546	11	58.96

Based on Table 38, the average complexity values obtained for cyclomatic complexity, LOC and currently used Campbell's cognitive complexity metric tend to be not varying, while the average cognitive complexity values obtained from proposed *CgC* metric tend to be varying for each user. As it has been already described, this situation is occurred due to subjective cognitive complexity computation performed in proposed metric. Thereby, the introduction of proposed *CgC* measure can be stated as meaningful in terms of indicating the subjective comprehension effort using a certain set of personal profile and source code aspects.

4.10 Summary

The computational design proposed by this research work is responsible to demonstrate the applicability of cognitive complexity concept inside software development and maintenance processes by demonstrating the mechanisms to follow to reduce the cognitive complexity. Accordingly, the processes of requirements analysis and visualization has been introduced as the procedures of assisting with human cognitive load to make the source code comprehension process easy. Moreover, handling bugs, code quality smells and introducing refactoring techniques have been proposed as the procedures of reducing the unnecessary cognitive load utilized with these processes when understanding a source code. Then, the assistance of these procedures implemented in proposed design has been verified over cognitive complexity reduction. As it has been already found, the duration taken to complete a process has been considered as an indicator to imply the effort on it, so that the comprehension effort has also been presented in terms of the duration taken to complete a task. All the components have been tested with respect to the duration taken to apply a said modification correctly inside a given source code by understanding its logic. If the modification has been done inaccurately, the users were asked to perform it repeatedly until it is received with correct output and recorded the average duration of it. The users to apply these modifications has been selected based on the marks obtained from a computing-based aptitude test. Accordingly, 500 BSc final year computing stream students who have scored over 60% from the above test has been selected as target user group. Consequently, the duration of applying a modification with the support of its requirements has been led to 13.83% duration reduction comparing to the same processing without using its requirements. Hence, providing the requirements along with a source code can be stated to reduce the comprehension duration, thereby ensuring a less comprehension effort to achieve a less cognitive complexity. Since the modification is applied to the same source code by same set of users in multiple times, the possibility of getting experienced with the same source code logic has been considered, and a time gap of one week has been maintained to reduce the users' awareness of the source code between two tests. It should be noted that the modification includes adding a new method or modifying an existing method by understanding the source code logic. Then, the duration to apply a modification to another source code with the aid of its logical visualization has been performed. Subsequently, a significant duration reduction of 19.81% has been achieved with respect to same processing without the aid of visualization. In addition to

that, same modification has been performed with the aid of EasyUML which is an existing visualization tool supported for java source codes. Surprisingly, the proposed visualization technique still supports for 4.81% duration reduction comparing to the comprehension related with EasyUML visualization. Based on both scenarios, a substantial duration reduction can be observed by proposed visualization technique with respect to the comprehension without any logical diagrams and the aid of EasyUML. Herein, the proposed visualization technique can be stated to achieve a less comprehension effort, thereby assuring a less cognitive complexity. Furthermore, it is noteworthy that the proposed visualization is comprised with logical diagrams generation without its source code and with its source code, which cannot be observed in other existing visualization techniques. As the next process, the verification of cognitive complexity through bugs tracing component has been carried out. The proposed defects tracker is capable of tracing the defects which are not supported by FindBugs bug tracker. Therefore, the users have been requested to analyze a source code logic and apply a modification with the aid of FindBugs and the proposed bug tracker and the duration has been recorded for both scenarios. A significant duration reduction of 35.84% can be seen from the proposed defects tracker over the comprehension with FindBugs tracker. Accordingly, the proposed bugs tracker can be indicated as another procedure to mitigate the comprehension effort and cognitive complexity. The components of detecting the code quality smells and applying refactoring techniques have also been tested based on the duration to perform a modification inside a given source code. In that situation also, 32.29% duration reduction can be observed by the proposed component over the facilities supported by NetBeans to detect the code smells and to apply refactoring techniques, which can verify a lesser amount of effort related with understanding a source code logic to gain a less cognitive complexity. Further, all the components have been statistically verified with their duration reduction using the hypothesis testing performed through paired samples T test along with 95% confidence interval. Herein, all the proposed components have been ensured to achieve a less comprehension duration over current practices and technologic, which can be used to reduce cognitive complexity.

The other major expectation of this research work is to propose a meaningful cognitive complexity metric which can be effectively used inside software development by adhering to its definition as well. As the preliminary step, defining valid cognitive weights to express the understandability level related with BCS has been carried out. The same user group has been evaluated against a questionnaire related with preliminary BCS and their comprehension levels

were monitored according to the marks and duration taken. The reason behind considering marks of the questionnaire is to observe the understandability level to identify its logic correctly. By statistically analyzing both of marks and duration, cognitive weights for each BCS have been introduced. Since these weights are confined for that user group, the applicability of using same results for entire user group has been hypnotically tested using paired samples T test along with 95% confidence interval. However, T test results have indicated the impossibility of using the same cognitive weights for entire user population. Therefore, the problem of verifying the cognitive weights assigned for BCS was raised, and as the solution, a new procedure of defining cognitive weights was analyzed. Consequently, a cognitive weight concept which emphasizes more on personal profile was suggested to be introduced as this research work elaborates personal profile than source code factors. The expectation was to predict and indicate users' comprehension level using a numerical value which differs from one person to another. A data set has been created based on the factors under personal profile and certain source code aspects verified by each user and cognitive weight assignment component. Then, the cognitive weight is supposed to be expressed in both quantitative and qualitative manners. To train and test the dataset for quantitative indications of cognitive weight, Linear Regression and Logistic Regression algorithms were used. Accordingly, Linear Regression algorithm has supported for 94% accuracy, while Logistic Regression algorithm have obtained 99.5% accuracy which derives overfitting problem. Therefore, Linear Regression algorithm has been used for quantitative cognitive weight prediction. On the other hand, Decision Tree algorithm and Gaussian Naïve Bayes algorithm were used to model the data set for qualitative weight prediction. Along with that, Decision Tree algorithm has shown 88.5% accuracy and Gaussian Naïve Bayes algorithm has shown 91.5% accuracy. Hence, Gaussian Naïve Bayes algorithm has been used to predict the cognitive weight qualitatively. It should be noted that more impact is given for quantitative cognitive weight as it is considered to be inputted as a parameter for new metric. The subjectivity associated with quantitative cognitive weights has been verified statistically by analyzing the cognitive weights obtained for two Gitlab project codes by same 500 users. Accordingly, the subjectivity of cognitive weights has been explained by exploring the variance obtained through descriptive statistics, and the possibility of indicating the average level of user understandability of a given source code has also been explained. Together with cognitive weight, a cognitive complexity metric (*CgC* metric) has been proposed which includes source code factors that have not been considered for cognitive weight determination. The calculation procedure of the metric

has been illustrated with several source codes, and it has been validated empirically and theoretically to prove its usage and stability in real software applications. It has been validated against Kaner's Framework to verify its usage in practical environment. For theoretical evaluation, Weyuker properties and Briand's framework have been used. It has satisfied all nine properties defined under Weyuker properties, while only two complexity properties have been satisfied by it. The complementary guidelines mentioned in Braind's framework with respect to Weyuker properties is the major reason behind achieving a lesser success rate by the metric. Therefore, the stability of *CgC* metric can be proven by considering 100% success rate with Weyuker properties. Finally, the possibility of indicating the subjective comprehension effort by *CgC* metric has been demonstrated using a set of Gitlab java project source codes. The complexity values computed by proposed *CgC* metric, a currently using cognitive complexity metric, cyclomatic complexity and LOC have been used for this purpose. By analyzing their variance through descriptive statistics, it has been shown that only proposed *CgC* metric is capable of existing with a variance, while all other complexity metrics produce objective complexities with no variance. Thereby, the applicability of demonstrating a meaningful calculation to quantitatively indicate the human comprehension level can be stated as achieved by proposed *CgC* metric.

5.0 CONCLUSIONS

5.1 Introduction

The proposed research work emphasizes the applicability of cognitive complexity concept in the field of software engineering. The cognitive complexity of a software determines the amount of effort required to understand the logic of its source code. The amount of effort to comprehend the source code logic can be defined as the understandability of a particular user with respect to a given source code. Hence, cognitive complexity can be used to express the understandability of a source code, which is a vital quality attribute in software developing. The understandability of a source code is useful in software developing process as a source code is developed by a set of people and it is modified by another set of people. The ability of applying a modification to a source code is referred to its maintainability, and it depends on how a user can understand the source code logic. Therefore, maintainability can be denoted using understandability and cognitive complexity is being a direct indicator of understandability, it can be stated that the concept of cognitive complexity can be used to describe both understandability and maintainability. The ability of comprehending a source code varies from one user to another user, as different users handle a same source code in different ways. Accordingly, cognitive complexity can be stated as a subjective measurement which is harder to quantify. Furthermore, the comprehension effort of an individual for a same source code gradually becomes lower with the growth of experience gained along with the time by dealing with it. Thereby, cognitive complexity is dynamically changed along with the time, so that it can be stated as a dynamic indicator of understandability and maintainability. Moreover, any source code of a software is expected to be easily comprehended so that it can be handled and maintained easily. Therefore, the expectation is to maintain the cognitive complexity of a source code in a reduced state, so that it will help to increase the understandability while assisting to maintain it without any failures.

Even though cognitive complexity is used to express the complexity of software in terms of comprehension effort, there are numerous software complexity metrics currently being used to evaluate software complexity according to numerous aspects. Among them, McCabe's cyclomatic complexity and LOC metrics are widely being used in industrial applications as well. McCabe's cyclomatic complexity computes the software complexity in terms of the number of linear independent paths inside the source code, while LOC computes the complexity with

respect to the size of source code. Therefore, the relationship of cognitive complexity with other complexity metrics has been explored. It is believed that a source code with higher cyclomatic complexity is difficult to understand due to high number of execution paths and test cases involved with it. Therefore, a high cyclomatic complexity source code can be stated as a source code with high cognitive complexity. Similarly, a higher LOC source code also considered as difficult to comprehend due to the length of its overall structure. Hence, a high LOC source code tends to be high in cognitive complexity. However, there are certain situations where a high cyclomatic complexity source code is easy to understand, and less cyclomatic complexity source code is difficult to understand. Similarly, situations such as a high LOC source code is easy to comprehend and a less LOC source code is difficult to comprehend also exists. Therefore, establishing a proper relationship between cognitive complexity and the metrics available to compute software complexity cannot be achieved. The reason for that scenario is the incapability of implying the actual difficulty level of users by current software complexity metrics. Herein, the problem of representing the actual user comprehension level by a proper mechanism can be stated.

Since the relationship of cognitive complexity with understandability can be directly established, addressing the understandability of a source code along with cognitive complexity has been taken as one of the significances of conducting this research work. Any software implementation is a teamwork, in which its team members have different comprehension levels. Further, the neglection of the understandability of team members would lead to a poor teamwork, which causes to have unnecessary failures. Therefore, a teamwork should be organized by identifying their comprehension efforts utilized for a given source code, which in turns aligned with cognitive complexity. Hence, the possibility of applying cognitive complexity in software project management is the other significance of this research.

In literature, there are numerous applications that cognitive complexity has been used. Among them, cognitive complexity metrics have played a vital role of quantifying the cognitive complexity. But, cognitive complexity is a non-quantifiable indicator as it is a subjective measurement. Nonetheless, numerous cognition metrics have been implemented by selecting several quantifiable factors affected with comprehension effort. It should be noted that the cognitive complexity metric is an indicator of cognitive complexity, which does not exactly represent the comprehension effort. In other words, cognitive complexity is denoted using a certain set of parameters through a form of metric. The consideration of source code factors for

all metric computations can be observed in literature, which the content of source code has been expressed by different categories such as input output variables, operators and operands, BCS, functions, spatial capacity, concepts with object orientation and many more. Accordingly, a variety of cognitive complexity metrics have been introduced, which makes difficult to standardize them into a single metric. Moreover, the concept of cognitive weight has been introduced with cognitive metric, which indicates the user comprehension level of a particular source code category with a numerical value in real usage. Even though the concept of cognitive weight has been introduced to represent the user involvement, it has not been conducted in a valid framework, since they have been assigned based on assumptions taken with respect to a particular user group. Hence, the problems of using these weights to indicate entire user population and the validity of these cognitive weights can be emphasized. As another significant approach, the applicability of cognitive complexity has been explained in a qualitative manner using a set of personal and software quality attributes. As such, the effectiveness of user expertise level, defects ratio of a source code, time consuming and complex tasks in a project, success rate of a project have been introduced to express cognitive complexity. Surprisingly, the usage of cognitive complexity in BPM, air traffic and aviation control systems, gaming development, career planning and theater-based applications emphasizes its applicability in other domains. Therefore, the concept of cognitive complexity has been considered as a vital factor of determining the progress of numerous contexts, so that its applicability should be a predominant factor to be determined inside software development domain as well.

Based on the subjectivity associated with each person with regard to cognitive complexity and the source code used to determine it, personal profile and source code factors must be considered as the major factors effecting for it. Nonetheless, the limitation of current cognitive complexity metrics being dependent only to source code factor can be highlighted. In other words, current cognitive complexity metrics do not imply the influence of considering personal profile correctly to express cognitive complexity. Moreover, the issues related with the relationship among understandability and maintainability with cognitive complexity should be considered. Even though cognitive complexity is said to be related with understandability and maintainability, there is no clear guidance given to connect and apply them within software development process. Therefore, to fill these gaps, the proposed research work has been carried out.

According to these considerations, the problem of this research work is mapped to observe the applicability of cognitive complexity to enhance software development and maintenance

processes. Based on that, following research questions have been generated to address the main research problem. The first research question is created to find the factors effecting for cognitive complexity to model it. Then, the second research question is generated with exploring the procedures to reducing the cognitive complexity. Finally, the last research question is comprised with introducing a meaningful cognitive complexity metric which can be used to facilitate the software development process. Accordingly, five research objectives have been defined to achieve the solutions to the research questions as follows. The first research objective is to identify the factors associated with cognitive complexity to model it properly, which addresses the first research question. Identifying the procedures of reducing the cognitive complexity has been considered as the second objective, which is aligned with second research question. The third research objective focuses to design a computational methodology to demonstrate the cognitive complexity reduction procedures, which addresses first and second research questions. Introducing a meaningful cognitive complexity metric which overcomes the drawbacks of current cognitive complexity metrics is the fourth research objective, and it also addresses first and third research questions. The last objective is comprised with evaluating the methodology used for the design and also the proposed metric, which addresses second and third research questions.

5.2 Achievement of Objective 1: Cognitive Complexity Factors Identification

The first objective of this research work is to identify the factors effected for cognitive complexity of a software. Through this research, main two factors have been found to influence with cognitive complexity namely personal profile and source code factors. It is evident that majority of research works have focused on source code factor to determine cognitive complexity, so that ignorance of it cannot be granted. As the cognitive complexity of a software is expressed through its source code, it must be included as a main factor to determine cognitive complexity. However, the definition of cognitive complexity indicates to consider the comprehension effort of a particular user for a given source code. This research work aims at users involved with software development and maintenance processes, so that the comprehension effort related to them has been analyzed as cognitive complexity. Further, it is a subjective measurement which depends on user who deals with a source code, thereby reluctance of

personal profile creates a considerable gap with modelling cognitive complexity. Hence, as the other main factor, personal profile has been considered.

With the expectation of describing cognitive complexity in a wider context, sub factors of these main factors have been analyzed. Accordingly, the capacity of memory has been considered as one subfactor inside personal profile as a logical comprehension of a source code is a process involved with the users' memory. The memory capacity can be divided into three categories based on the duration that each piece of information is working with the users' brain such as long term, short terms and working memories. Since the cognitive complexity is determined for a given source code within a certain duration, the effectiveness of working memory is higher. As such, this work has given more emphasis to the involvement of a users' working memory with cognitive complexity. Moreover, it has been found that the working memory has been categorized intrinsic, extraneous and germane cognitive load. Therefore, the effectiveness of these three types of cognitive loads under working memory has also been considered for cognitive complexity. Apart from memory capacity, the aptitude level of a person also determines the amount of effort required for a source code comprehension. The aptitude level has been further described as the preference given to a programming language and skill level associated with analyzing a problem. Then, as the next sub factor, the experience and age of an individual has been considered. Although experience of a person develops with the age, age has not been considered as a direct sub factor to express cognitive complexity as age has been found as a dynamic indicator to express the cognitive complexity. Therefore, the involvement of personal profile can be stated as the major reason behind resulting cognitive complexity as a subjective measurement. Moreover, personal profile can be concluded as the major parameter to specify the dynamic status of cognitive complexity. The other major factor to express cognitive complexity is source code factor, which all other research works have focused on. The amount of information which is referred to the architectural aspect of the source code has been considered as one sub factor under source code factor. Further, it is expressed with respect to variables/attributes, BCS, operator and operands, input output parameters inside a given source code and the influence of object-oriented concepts. The spatial capacity of source code is another sub factor, which refers to the size of source code and the distance between the module call to its implementation in terms of LOC. Moreover, the support given by the programming environment to implement the source code has also been considered as a sub factor of source code factor. It has further categorized into the structure of source code and the availability of comments. Hence,

the combination of all main factors and sub factors belongs to main factors have been considered as the indicators of expressing cognitive complexity, as all of these factors can be effectively applied to change the user comprehension level as expected. However, it should be noted that the factors associated with cognitive complexity cannot be limited, so that this research work has targeted only a certain set of parameters to denote cognitive complexity as it has already been mentioned above.

5.3 Achievement of Objective 2: Identify the Procedures of Reducing Cognitive Complexity

As it has been already described, the utmost expectation of any software development is to produce a simple source code which is easy to handle and maintain. To describe it further, a source code of any software should be easily understood so that it effects to achieve a less software complexity and high maintainability. A highly understandable source code directs with less comprehension effort utilized with users, so that its cognitive complexity should be less. Therefore, a source code should be implemented to achieve a less cognitive complexity. As such, the steps to be followed to achieve a less cognitive complexity have been explored through this objective.

A source code is said to be complex if it is difficult to understand. Similarly, a simple source code is easy to understand. This research work has emphasized more on personal factor to express the cognitive complexity, so that reasons of getting a source code to be complex are analyzed in terms of personal profile. As discussed under first objective, one of the main reasons to experience a high difficulty in understanding is that the user does not have enough memory capacity to understand it. Especially, it is due to the problems of working memory of a user such that working memory contributes more when comprehending a source code at a given time. Therefore, handling working memory properly which is capable to understand a given source code has been taken considered throughout this research. As such, the procedures of handling working memory of a person with respect to a source code understandability has been explored. The working memory is expressed using three types of cognitive loads, in which intrinsic and germane cognitive loads assist with the capability of maintaining necessary information and extraneous cognitive load is responsible to maintain unnecessary information that is not aligned

with source code logic. Therefore, reducing extraneous cognitive load and increasing intrinsic and germane cognitive loads is the optimal solution to reduce the cognitive complexity in terms of working memory. As such, the procedures of reducing extraneous cognitive load and increasing intrinsic and germane cognitive load have been studied to indicate the procedures of reducing cognitive complexity of a source code. Under extraneous cognitive load reduction, maintaining an error free and structured source code have been introduced as an unstructured source code with numerous coding defects is difficult to comprehend. Therefore, reducing coding defects and maintaining a proper coding structure can reduce unnecessary information involved with handling those, thereby the possibility of having more capacity to maintain necessary information can be increased. Under intrinsic and germane cognitive load incrementation, the procedures of assisting these cognitive loads have been analyzed. Consequently, the visualization of source code logic, simulation of functionalities or logical flow of source code and providing recommendations and guidelines to assist the user with handling maintaining the source code have been found. Therefore, initiating these procedures can be considered as the alternatives of assisting with source code logic, thereby they can guide the user to easily comprehend the source code. To conclude, the reduction of cognitive complexity has been achieved by maintaining the cognitive load effected with working memory. Through that, a certain set of procedures to reduce unnecessary cognitive load and to assist with necessary cognitive loads have been introduced with this research work.

5.4 Achievement of Objective 3: Design a Methodology of Demonstrating Cognitive Complexity Reduction Procedures

Based on the outcomes accomplished from the previous objective, a computational design has been implemented to assist with users to understand a given source code logic easily by producing a source code with less cognitive complexity. It has been designed in NetBeans IDE. Firstly, the possibility of assisting the cognitive load with necessary information processing has been analyzed. It has been already discussed that the visualization of a source code logic can assist the user to understand its logic easily. In addition to that, referring to the requirements of software has also been introduced as a mechanism to understand a source code logic easily. To describe it furthermore, a logic of a complex source code can be lessened by referring to its visualization and requirements. Consequently, two components have been introduced to the

proposed design, which can indicate the requirements and to provide UML diagrams based on source code logic.

Generally, the requirements specified for a software is documented in its project proposal document. Therefore, the project proposal document should be inputted to the requirements analyzer component to reiver the exact requirements. However, the lack of maintaining a common structure of proposal document makes it difficult to gain the requirements computationally. Accordingly, the proposal document has been converted into a common format which consists with five sections namely Introduction, Problem Definition, Solution, Functionality and Team Profile. The Functionality section consists with exact requirements that should be addressed through software implementation. Nevertheless, extracting the same content inside Functionality section is not the expected output of this component as it does not involve with any processing. Therefore, obtaining requirements has been performed which can be assisted to diagram generation process as well. Subsequently, it has been targeted to gain the class names out of Functionality section which can then be used to generate UML diagrams as well. The class names have been identified by recognizing the highest frequency nouns inside Functionality section, and the possibility of controlling the highest frequency has been provided through the component. To identify the nouns based on highest frequency, POS tagging has been used. Therefore, requirements analyzer is capable of generating the possible class names as requirements, which also helps to generate the diagrams to visually represent its logic.

As the other component to assist with the necessary cognitive load of user is to generate UML diagrams to represent the source code logic. The significance of this component is to generate UML diagrams based on requirements without referring to its source code as followed in SDLC. Further, it is capable of generating UML diagram based on its source code as well. Visualization of logical diagrams based on the requirements has been performed as follows. It is implemented to generate class, ER and object diagrams. The requirements have been obtained from the output generated by requirements analyzer, which are the possible class names that can be used to generate diagrams. Each of class names have been mapped with GloVe embeddings and the similarity of each pair of class names has been computed through cosine similarities. The similarity values have been categorized into five relationship types such as inheritance, composition, aggregation, association and no relation to be used to model class diagrams. In addition to that, each relationship type has been introduced with a notation, so that identification of them in the visualization process becomes easy. Accordingly, each pair of class name has

been verified through corresponding relationship type and assigned with related notation and coded with the statements to generate the corresponding class diagram. It should be noted that all diagrams have been generated as a result of constructing relevant PlantUML statements and executing them. In order to generate ER diagram, three types of ER multiplicity levels have been introduced based on the relationships identified earlier. Accordingly, the relevant notations have been introduced and constructed PlantUML statements to generate ER diagram. The object diagram represents a view of classes represented in class diagram, so that possible objects have been demonstrated after recognizing their relationship between them. However, the lack of information can be observed in these diagrams, as they have been generated without its source code. Hence, these diagrams can only provide a predictive visualization of the control flow of source code and the relationships between each information. As such, another sub visualization component of generating class and sequence diagrams has been introduced. The class diagram is generated through *UMLGenrator.java* file, which analyzes all class names, their relationship types, constructors, methods and their parameters. Based on these information, corresponding PlantUML statements have been generated and executed to obtain the expected class diagram. The sequence diagram generation is achieved through *SequenceAspect.aj* file, which gains all information through method signatures such as method name, return type, parameters and the messages which are passed from one method calling to another. Then, the execution of corresponding PlantUML statements have generated the sequence diagram. Especially, the possibility of saving these diagrams as a PNG image file has also been provided for future reference. Therefore, the mechanisms supported to generate logical diagrams of a source code logic have targeted to make the understanding process easier than referring only to its source code.

Along with necessary cognitive load assisting procedures, the proposed system consists with several components which assists to reduce unnecessary cognitive load of user to assist with easy comprehension of a source code. Tracing coding defects has been introduced as one option under that circumstance. NetBeans IDE is capable of tracking coding defects and there are existing bug trackers that can be installed as a plugin to detect bugs. However, there are certain bugs that cannot be traced using existing bug trackers. This research considers FindBugs bug tracker tool to be integrated as a plugin, and the bugs that is not handled by FindBugs bug tracker has been handled through the proposed component. Once the source code is inputted, the component is capable to list down all the identified bugs along with their line numbers by comparing the code

with the syntaxes of coding defects. If there is a defect, it recommends guidelines to fix it in more understandable manner. Moreover, it is capable to provide a summary on the defects that has been found and corrected. It should be noted that the component is capable to detect only twelve defects that cannot be identified by FindBugs bug tracker. Thereby, it is supposed to lessen unnecessary cognitive load by supporting to identify and fix coding defects. Consequently, the component is capable to generate a source code with less coding defects, which can achieve a less cognitive complexity.

The other component is responsible to maintain a proper structure of a source code. It has been achieved by handling six java code smells. The source code is the input for this component, and it is capable to identify the code elements of inputted source code and to compare them with syntaxes of code smells that the component can detect. Once a code smell is found, the component provides the guidelines to fix it in detail in a way that user can understand it properly. The facilities such as saving the modified source code in same application directory and providing a log file with the modifications applied can also be observed in this component. In addition to that, some of code smells have been provided with auto fixing feature, in which the user has no intervention on solving those. Hence, the reduction of unnecessary cognitive load by assisting the user to identify and fix code smells is targeted to be achieved by proposed component.

Along with code smells identifier, another component of applying refactoring techniques has been introduced to the system, since refactoring is considered to be the current trend of restructuring a source code to increase its understandability. It has been implemented as a hosting component which can be accessible by any user, and users do not require to have the other components of the design to function this refactoring component. The client and backend processes have been implemented with ReactJS and NodeJS respectively. The component is capable to prompt the possible refactoring techniques that can be applied, once the source code is uploaded. Then, the refactoring techniques are applied based on the users' selection on a particular technique. The client process is responsible to obtain the source code and backend process is capable to identify the possible refactoring techniques that can be applied. It should be noted that the component is capable to handle six different refactoring techniques. Each refactoring technique is handled with two controllers such that validation controller detects the structuring issue and refactoring controller can apply refactoring techniques without user intervention. The significance of this component over current refactoring tools is that it is

capable to list only the applicable techniques for a given source code rather than displaying the same set of techniques for any source code. Moreover, the option of downloading the refactored source code has also been provided in this component. Herein, the proposed component is expected to assist the user with utilizing unnecessary cognitive load to restructure the source code to make it easier to understand, thereby ensuring a lesser cognitive complexity.

5.5 Achievement of Objective 4: Introduce a Meaningful Cognitive Complexity Metric

Throughout this objective, it is expected to introduce a cognitive complexity metric which overcomes the problems of current cognitive complexity metrics. The traditional metrics to compute the cognitive complexity have been limited only with source code factors consideration. In other words, the involvement of personal profile cannot be observed in current metrics, so that the subjectivity associated with user comprehension level could not be presented in a meaningful way. Each metric has emphasized different source code aspects so that a variety of cognitive complexity metrics can be observed, which raises to have a standard cognitive complexity metric for usage. Moreover, a concept of cognitive weight which indicates the human comprehension level of a selected source category can be deemed as the involvement of personal factor to compute cognitive complexity. However, it represents only a specific user group and weightage assignment has been performed merely based on assumptions. Consequently, the impossibility of validating these metrics which are applicable for entire user population exists. Therefore, as the preliminary step, introducing a set of valid cognitive weights for a selected source code category has been performed. BCS have been selected as the source code category as they have been considered in majority of previous cognitive complexity metrics due to the behavior of controlling the logical flow. In order to avoid the assumptions taken for weightage assignment by previous metrics, the comprehension level of a selected user group was decided to be obtained through a BCS questionnaire. As the user group, 500 BSc computing related final year students have been selected after conducting another computing related aptitude test. The questionnaire was conducted in Moodle environment, and it is consisted with questions related with *if-else* and *switch-case* under conditional statements, *for* and *while* loop under looping criteria and *nested for* and *nested while* loop under nested looping criteria. The expectation was to monitor duration and marks obtained from each user and to assign cognitive weights statistically using mean of

duration and mean of marks. Although the weights have been assigned statistically, several contradictions have been seen which does not align with users' practices and preferences of using BCS. Moreover, the hypothetical testing conducted through paired samples T test has shown the impossibility of using majority of cognitive weights entitled for entire user population. Therefore, the mechanism of assigning cognitive weights for BCS is inclined to be invalid. Nevertheless, the concept of cognitive weights cannot be ignored as it can be used to represent the involvement of personal profile. Furthermore, this research work has given high impact on personal profile, thereby ignoring cognitive weights can create a lack of connectivity between its involvement with cognitive complexity. Therefore, a new cognitive weight (*Cw*) has been introduced which highlights more on factors associated with personal factor.

Cw has been introduced such that it can be derived as a predictive measurement based on each users' input to the factors associated with it. Also, cognitive weightage allocation component is capable to express *Cw* in both quantitative (1-5) and qualitative (*high* and *low*) manners. The quantitative weight 1 indicates a high comprehensibility and weight 5 indicates less comprehensibility. The factors considered to predict *Cw* is aptitude level on Java, experience and age, while spatial capacity is included as a parameter on behalf of source code factors. The accuracy of users' input and rating for these factors are verified from the component by prompting a relative question. Then, the gathered data set with verified ratings have been trained and tested to predict *Cw* quantitatively and qualitatively. Linear Regression algorithm was used to predict *Cw* in quantitative form, while Gaussian Naïve Bayes algorithm was used to predict *Cw* in qualitative manner. It should be noted that expressing *Cw* in qualitative manner is only to compare it with the quantitative value obtained for *Cw*.

Along with person biased cognitive weight, a meaningful cognitive complexity metric (*CgC*) has been introduced which involves both personal profile and source code factors as two main categorizations of cognitive complexity. The inclusion of *Cw* under personal profile involvement highlights the subjectivity of the metric, which can be considered as a significant achievement that cannot be seen in traditional cognition metrics. Since the value obtained of *Cw* is comprised with spatial capacity under source code factor, it has not been considered again for metric computation. But, the architectural aspect under source code factor have not been considered under *Cw* prediction, so that its complexity (*AC*) has been included for the metric computation process. It has been computed as the addition of the complexities derived from data variables/attributes and BCS. The complexity of variables/attributes (*DC*) have been computed

by obtaining the count of distinct variables/attributes by considering each of variables/attributes as a unit of information. To compute the complexity of BCS (*BCSC*), the number of BCS and the nested level of each BCS have been considered. Therefore, along with the addition of *AC* and *Cw*, *CgC* metric has been proposed to quantitatively indicate the cognitive complexity of a given source code. Based on the subjectivity associated with *Cw* and consideration of both personal profile and source code factor can emphasize the significance of its applicability comparing to existing cognitive complexity metrics.

In addition to that, the possibility of computing the software complexity using existing complexity metrics has been provided. Accordingly, six complexity metrics have been introduced to calculate the complexity. The user is given the opportunity to refer to the complexity computation process in each line of the source code along with the finalized complexity value. The expectation is to demonstrate the objectivity associated with current complexity metrics and the subjectivity supported by the proposed cognitive complexity metric (*CgC*).

5.6 Achievement of Objective 5: Evaluate the Methodology used in the Design and the Metric

The computational design of this research work has been implemented to demonstrate the procedures of reducing the cognitive complexity. As such, the components of analyzing the requirements and visualization have been introduced as assistive mechanisms for necessary cognitive load and tracing defects, code quality optimization and refactoring components have been introduced as the mechanisms of reducing unnecessary cognitive load of users. To evaluate these components to verify the ability to reduce cognitive complexity, the level of progress has been monitored with respect to a modification applied to thirty different software by a set of users. The modification applied to a software is regarded as a modification applies to an existing method or to implement a new method, in which both requires to comprehend the logic of corresponding source code. As for the user group, 400 students were considered from same student group who have been selected for BCS questionnaire under cognitive weight introduction process. These students have been divided into another four groups based on their marks obtained for the preliminary aptitude test, so that each group is entailed with 100 students. The duration taken to complete the said modification has been monitored in each component, since the

duration has been found to express the comprehension effort or the understandability of source code. It should be noted that the average duration to complete the task have been computed to represent all users' duration conveniently. All these evaluations have been performed in Moodle environment, so that obtaining the duration can be performed precisely.

Firstly, evaluation of requirements analyzer has been performed. Users have been provided source codes implemented for thirty different software and requested to apply a modification by referring to its requirements. Then they have been asked to repeat the same task by using the source code only. A time gap of one week has been maintained to ensure the unawareness of using the same source code repeatedly. The time taken by each user group for both scenarios was monitored and calculated mean duration. Based on mean durations, 13.83% duration reduction has been observed for comprehension with requirements comparing to the comprehension with only source code. The same procedure was followed to evaluate the visualization component such that mean time taken to comprehend a source code to apply another modification with the visualization component and without it were recorded. Consequently, 19.81% duration reduction has been obtained to the source code comprehension between the proposed visualization aid and the source code referring. Furthermore, mean time was computed to perform the same task by using the logical diagrams generated by EasyUML. A significant reduction of 4.81% duration has been observed in proposed visualization component over EasyUML, which concludes the possibility of assisting user to understand a source code logic more conveniently than existing diagram generation tools. Similarly, defects tracing component has also been evaluated against the mean time taken to comprehend and apply modifications along with the usage of FindBugs bug tracker. A significant 35.84% duration reduction has been observed in proposed defects tracker comparing to FindBugs bug tracker, which highlights its ability to support user to understand the source code logic. Finally, mean duration taken to process another modification using both code quality optimization and refactoring components was monitored along with the duration to process the same task without these proposed components. As in other components, 32.29% duration reduction has been observed from proposed components over the available tools and techniques provided by NetBeans for code quality optimization and refactoring. Therefore, all the proposed components can be stated to attain a considerable reduction in mean duration to apply a modification by understanding the source code logic, which in turns reflect to maintain the users' cognitive load in a more consistent way that would utilize a lesser comprehension effort. Moreover, all these scenarios have been hypothetically verified with paired samples T test,

such that all proposed components' mean time taken to comprehend a given source code have been observed as lesser than the duration acquired to comprehend the same source code without those components for entire user population. Hence, the proposed design can be concluded as a supportive mechanism which indicates possible mechanisms to reduce the cognitive complexity of a software.

Before evaluating the proposed cognitive complexity metric, the proposed cognitive weight (*Cw*) has been evaluated as it is the parameter of expressing personal profile involvement in the metric. The involvement of personal factor should be subjective, thereby cognitive weightage should demonstrate the subjectivity associated with user understandability. Even though cognitive weight has been indicated as a numerical value to make it easy to quantify cognitive complexity, the proposed cognitive weight has been predicted in both quantitative and qualitative manners to verify them in both ways. In order to predict cognitive weight in quantitative manner, Linear Regression algorithm has been used as it has shown an accuracy of 94.0%. The Logistic Regression algorithm has also been tested against the accuracy and gained 99.5%, which can be considered as overfitting to gain low performance of testing results. To predict the cognitive weight in a qualitative manner, Gaussian Naive Bayes algorithm has been used due to its accuracy of 91.5% over 88.5% accuracy gained by Decision Tree algorithm. Since the metric is comprised with the quantitative cognitive weight, more priority has been given to evaluate it in terms of its subjectivity. Accordingly, two Java source codes generated by two GitLab sample projects have been tested among same user group to obtain cognitive weights as per their input and ratings for *Cw*. Based on their descriptive statistics, a significant variation has been observed in *Cw* generated for both source codes. Since a positive variation implies a changeability of the weights gained by each user, the subjectivity associated with *Cw* can be highlighted, which cannot be observed in current cognitive weights. Moreover, the ability to indicate the comprehension level by analyzing the maximum cognitive weight frequency for a given source code has been emphasized. Accordingly, if a maximum *Cw* frequency lies in a lower range, it can be stated as more understandable for majority of users, while a higher range applied with *Cw* frequency indicates that the source code is difficult to understand by majority of users. Therefore, the practicability of proposed *Cw* to indicate the personal profile can be verified to be used in cognitive complexity metric.

The proposed cognitive complexity metric (*CgC*) consists with *Cw* and quantified Architectural Complexity (*AC*) with respect to the complexities derived by variables/attributes referred to as

Data Complexity (*DC*) and BCS Complexity (*BCSC*). Therefore, the addition of *Cw* and *AC* together has been proposed as an indicator of expressing the cognitive complexity of a particular source code. The applicability and the procedure of using and computing the value for the metric has been illustrated using several Java source codes. Moreover, the metric has been assessed against Kaner's validation framework to verify its practicability in real applications by addressing all ten guidelines mentioned in it. The theoretical validation of the metric has been performed through Weyuker properties and Briand's framework. The metric is capable of satisfying all nine properties under Weyuker properties and two complexity properties have been satisfied under Briand's framework. The contradictive properties defined in both frameworks are the major reason behind achieving a lesser satisfaction rate under Briand's framework. Therefore, by considering 100% satisfaction ratio through Weyuker properties, the stability of the proposed *CgC* metric can be highlighted. As it has been observed in proposed cognitive weight, the subjectivity associated with *CgC* metric has also been examined by calculating cognitive complexity for ten different GitLab projects by the same user group. The complexity for same set of source codes were calculated by using existing software complexity metrics and by using a latest cognitive complexity metric. Based on the calculated complexity values, a considerable variation has been obtained with average *CgC* value which indicates the possibility of demonstrating the variation of the understandability of each user. However, the objective values obtained from current software complexity metrics and cognition metric emphasizes the inability to demonstrate that variation. Herein, the proposed *CgC* metric can be concluded as a meaningful metric which can be used to indicate the user comprehension effort with respect to both personal profile and source code factors. Furthermore, more emphasis given to the personal profile through this metric has shown a significant achievement of demonstrating the subjectivity related with user understandability, which have not been achieved from other complexity metrics.

5.7 Scope of the Study

The cognitive complexity discusses about the effort utilized to understand a given source code. The comprehension of a source code is generally performed by two parties. The users engaged in software development and maintenance processes are required to understand the logic of a given source code in order to handle it as expected. On the other hand, the computer should understand its logic by converting the logical flow of source code in a computer understandable format in order to execute it and produce the output. In this research work, the cognitive complexity associated with users involved in software development and maintenance has been discussed. Even though the logic of a source code is implemented correctly or incorrectly, a considerable amount of effort is required for the logical comprehension. Therefore, this research work is emphasized on the cognitive complexity of a user with respect to a source code with correct logical implementation. It has been found that the major factors to determine cognitive complexity is personal profile and source code factors. Consequently, the sub factors associated with each main factor have been described to facilitate the cognitive complexity. However, the combination of all these main factors and sub factors do not represent the entire cognitive complexity effected with a source code as the factors considered for cognitive complexity determination under this research work is limited. Moreover, it should be noted that this work has emphasized more on personal profile than source code factors, as majority of previous research works have emphasized source code factor only. Hence, the design implemented to demonstrate the procedures of reducing cognitive complexity and the proposed cognitive complexity metric along with new cognitive weightage introduction have focused more on personal profile. This can be further described as the procedures of reducing cognitive complexity have been found on the basis of handling the cognitive load of a users' working memory, which comes through personal profile. Moreover, the majority of parameters considered to predict the new cognitive weight (*Cw*) are from personal profile. Hence, the proposed cognitive complexity metric (*CgC*) can also be considered as an indicator of cognitive complexity which emphasizes more on personal profile. Furthermore, the procedures developed to lessen cognitive complexity through software development and maintenance processes and the factors considered for *Cw* prediction can also be stated as limited with respect to personal profile. To evaluate the system components, proposed *Cw* and *CgC* metric, a group of students were selected based on the marks obtained from a computer-based aptitude test. Therefore, it has been assumed that all selected students consist

with same comprehension level in order to maintain a stability of the measurement taken for evaluation. It should be noted that, the BCS questionnaire has also been conducted along with the same assumption. The procedures of reducing cognitive complexity have been evaluated by considering mean time taken to apply a modification to a source code by understanding it. Even though that duration indicates time taken to comprehend and perform modifications, it has been assumed that total time is equivalent to the comprehension time, in which modification time has been ignored. Also, a week gap has been maintained among same type of modifications assuming that the users' awareness of the same source code will not be existed after one week. Finally, weightage assignment from 1-3 has been defined as high comprehension level region, and 4-5 has been defined as high comprehension level region, since a lesser cognitive weight implies high understandability, and a higher cognitive weight implies low understandability.

5.8 Contribution of the Study

In this study, cognitive complexity has been expressed in terms of personal profile and source code factors. This can be considered as a vital achievement and a contribution to the field of software engineering, as most of the research works on cognitive complexity has been limited only to source code factors. Along with the inclusion of personal profile to express cognitive complexity and with more emphasis given to personal profile over source code factors, the subjectivity associated with users' comprehension effort has been demonstrated, which other research works could not be capable of handling. Consequently, the proposed cognitive weight (*Cw*) and cognitive complexity metric (*CgC*) are capable of illustrating the subjectivity associated with user cognition, although they compute a quantitative value. However, proposed cognitive weight is an indicator of personal profile, and the metric is an indicator of cognitive complexity, which are not equivalent to the exact cognitive complexity. Therefore, with this research work, the applicability of cognitive complexity concept inside software development and maintenance processes has been introduced without restraining it to a quantifiable measurement through a metric. In other words, the possibility of using cognitive complexity concept is studied beyond indicating it through a metric. As such, the procedures of reducing the cognitive complexity of a software through its source code have been introduced to achieve high understandability and maintainability, thereby to ensure a less complex source code. Hence, the computational design

of this research work contributes to analyze the procedures of obtaining a source code with high understandability and maintainability.

References

[1] A. K. Jakhar and K. Rajnish, “A New Cognitive Approach to Measure the Complexity of Software,” *Int. J. Softw. Eng. Its Appl.*, vol. 8, pp. 185–198, Jul. 2014.

[2] G. A. Campbell, “Cognitive complexity: an overview and evaluation,” in *Proceedings of the 2018 International Conference on Technical Debt - TechDebt ’18*, Gothenburg, Sweden: ACM Press, 2018, pp. 57–58. doi: 10.1145/3194164.3194186.

[3] J. Feigenspan, S. Apel, J. Liebig, and C. Kastner, “Exploring Software Measures to Assess Program Comprehension,” in *2011 International Symposium on Empirical Software Engineering and Measurement*, Banff, AB, Canada: IEEE, Sep. 2011, pp. 127–136. doi: 10.1109/ESEM.2011.21.

[4] M. M. Barón, M. Wyrich, and S. Wagner, “An Empirical Validation of Cognitive Complexity as a Measure of Source Code Understandability,” *Proc. 14th ACM IEEE Int. Symp. Empir. Softw. Eng. Meas. ESEM*, pp. 1–12, Oct. 2020, doi: 10.1145/3382494.3410636.

[5] L. Lavazza, A. Z. Abualkishik, G. Liu, and S. Morasca, “An empirical evaluation of the ‘Cognitive Complexity’ measure as a predictor of code understandability,” *J. Syst. Softw.*, vol. 197, p. 111561, Mar. 2023, doi: 10.1016/j.jss.2022.111561.

[6] A. Trockman, K. Cates, M. Mozina, T. Nguyen, C. Kästner, and B. Vasilescu, “‘Automatically assessing code understandability’ reanalyzed: combined metrics matter,” in *Proceedings of the 15th International Conference on Mining Software Repositories*, Gothenburg Sweden: ACM, May 2018, pp. 314–318. doi: 10.1145/3196398.3196441.

[7] M. Wyrich, L. Merz, and D. Graziotin, “Anchoring code understandability evaluations through task descriptions,” in *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*, Virtual Event: ACM, May 2022, pp. 133–140. doi: 10.1145/3524610.3527904.

[8] L. Ardito, R. Coppola, L. Barbato, and D. Verga, “A Tool-Based Perspective on Software Code Maintainability Metrics: A Systematic Literature Review,” *Sci. Program.*, vol. 2020, pp. 1–26, Aug. 2020, doi: 10.1155/2020/8840389.

[9] K. K. Aggarwal, Y. Singh, and J. K. Chhabra, “An integrated measure of software maintainability,” in *Annual Reliability and Maintainability Symposium. 2002 Proceedings (Cat. No.02CH37318)*, Seattle, WA, USA: IEEE, 2002, pp. 235–241. doi: 10.1109/RAMS.2002.981648.

[10] S. Scalabrino, G. Bavota, C. Vendome, M. Linares-Vasquez, D. Poshyvanyk, and R. Oliveto, “Automatically Assessing Code Understandability,” *IEEE Trans. Softw. Eng.*, vol. 47, no. 3, pp. 595–613, Mar. 2021, doi: 10.1109/TSE.2019.2901468.

[11] S. Yu and S. Zhou, “A survey on metric of software complexity,” in *2010 2nd IEEE International Conference on Information Management and Engineering*, Chengdu, China: IEEE, 2010, pp. 352–356. doi: 10.1109/ICIME.2010.5477581.

[12] R. Saborido, J. Ferrer, F. Chicano, and E. Alba, “Automatizing Software Cognitive Complexity Reduction,” *IEEE Access*, vol. 10, pp. 11642–11656, 2022, doi: 10.1109/ACCESS.2022.3144743.

[13] G. A. Campbell, “A new way of measuring understandability,” vol. 1, no. 5, p. 21, Apr. 2021.

[14] N. Kasto and J. Whalley, “Measuring the difficulty of code comprehension tasks using software metrics,” *Fifteenth Australas. Comput. Educ. Conf.*, vol. 136, p. 7, 2013.

[15] A. Madi, O. K. Zein, and S. Kadry, “On the Improvement of Cyclomatic Complexity Metric,” *Int. J. Softw. Eng. Its Appl.*, vol. 7, no. 2, pp. 67–82, 2013.

[16] G. Jay, J. E. Hale, R. K. Smith, D. Hale, N. A. Kraft, and C. Ward, “Cyclomatic Complexity and Lines of Code: Empirical Evidence of a Stable Linear Relationship,” *J. Softw. Eng. Appl.*, vol. 02, no. 03, pp. 137–143, 2009, doi: 10.4236/jsea.2009.23020.

[17] L. Kaur and A. Mishra, “Cognitive complexity as a quantifier of version to version Java-based source code change: An empirical probe,” *Inf. Softw. Technol.*, vol. 106, pp. 31–48, Feb. 2019, doi: 10.1016/j.infsof.2018.09.002.

[18] O. Esther, O. Stephen, O. Elijah, A. Rafiu, T. Dimple, and Y. Olajide, “Development of an Improved Cognitive Complexity Metrics for Object- Oriented Codes,” *Br. J. Math. Comput. Sci.*, vol. 18, no. 2, pp. 1–11, Jan. 2016, doi: 10.9734/BJMCS/2016/28515.

[19] J. K. Chhabra, “Code Cognitive Complexity: A New Measure,” *Proc. World Congr. Eng. 2011*, vol. 3, p. 5, 2011.

[20] S. Misra, M. Koyuncu, M. Crasso, C. Mateos, and A. Zunino, “A Suite of Cognitive Complexity Metrics,” in *Computational Science and Its Applications – ICCSA 2012*, B. Murgante, O. Gervasi, S. Misra, N. Nedjah, A. M. A. C. Rocha, D. Taniar, and B. O. Apduhan, Eds., in Lecture Notes in Computer Science, vol. 7336. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 234–247. doi: 10.1007/978-3-642-31128-4_17.

[21] S. Misra, A. Adewumi, L. Fernandez-Sanz, and R. Damasevicius, “A Suite of Object Oriented Cognitive Complexity Metrics,” *IEEE Access*, vol. 6, pp. 8782–8796, 2018, doi: 10.1109/ACCESS.2018.2791344.

[22] M. Crasso, C. Mateos, A. Zunino, S. Misra, and P. Polvor´ın, “ASSESSING COGNITIVE COMPLEXITY IN JAVA-BASED OBJECT-ORIENTED SYSTEMS: METRICS AND TOOL SUPPORT,” *Comput. Inform.*, vol. 35, pp. 497–527, 2016.

[23] N. Setiani, R. Ferdiana, and R. Hartanto, “Test Case Understandability Model,” *IEEE Access*, vol. 8, pp. 169036–169046, 2020, doi: 10.1109/ACCESS.2020.3022876.

[24] A. Mishra and D. Dubey, “A comparative study of different software development life cycle models in different scenarios,” *Int. J. Adv. Res. Comput. Sci. Manag. Stud.*, vol. 1, no. 5, 2013.

[25] Jingqiu Shao and Yingxu Wang, “A new measure of software complexity based on cognitive weights,” *Can. J. Electr. Comput. Eng.*, vol. 28, no. 2, pp. 69–74, Apr. 2003, doi: 10.1109/CJECE.2003.1532511.

[26] D. S. Kushwaha and A. K. Misra, “Robustness analysis of cognitive information complexity measure using Weyuker properties,” *ACM SIGSOFT Softw. Eng. Notes*, vol. 31, no. 1, pp. 1–6, Jan. 2006, doi: 10.1145/1108768.1108775.

[27] S. Misra, “Modified Cognitive Complexity Measure,” in *Computer and Information Sciences – ISCIS 2006*, A. Levi, E. Savaş, H. Yenigün, S. Balcısoy, and Y. Saygın, Eds., in Lecture Notes in Computer Science, vol. 4263. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 1050–1059. doi: 10.1007/11902140_109.

[28] S. Misra, “Cognitive Program Complexity Measure,” in *6th IEEE International Conference on Cognitive Informatics*, IEEE, Aug. 2007, pp. 120–125. doi: 10.1109/COGINF.2007.4341881.

[29] S. Misra, I. Akman, and M. Koyuncu, “An inheritance complexity metric for object-oriented code: A cognitive approach,” *Sadhana*, vol. 36, no. 3, pp. 317–337, Jun. 2011, doi: 10.1007/s12046-011-0028-2.

[30] S. Misra and F. Cafer, “Estimating complexity of programs in Python language,” *Teh. Vjesn.*, vol. 18, no. 1, pp. 23–32, 2011.

[31] D. S. Kushwaha and A. K. Misra, “Improved Cognitive Information Complexity Measure: A Metric that Establishes Program Comprehension Effort,” *ACM SIGSOFT Softw. Eng. Notes*, vol. 31, no. 5, p. 7, 2006, doi: 10.1145/1163514.1163533.

[32] I. Candela, G. Bavota, B. Russo, and R. Oliveto, “Using cohesion and coupling for software remodularization: Is it enough?,” *ACM Trans. Softw. Eng. Methodol. TOSEM*, vol. 25, no. 3, pp. 1–28, 2016.

[33] A. Aloysius and L. Arockiam, “A Survey on Metric of Software Cognitive Complexity for OO design,” vol. 5, no. 10, p. 5, 2011.

[34] A. Aloysius and L. Arockiam, “Coupling Complexity Metric: A Cognitive Approach,” *Int. J. Inf. Technol. Comput. Sci.*, vol. 4, no. 9, pp. 29–35, Aug. 2012, doi: 10.5815/ijitcs.2012.09.04.

[35] U. Chhillar and S. Bhasin, “A New Weighted Composite Complexity Measure for Object-Oriented Systems,” *Int. J. Inf. Commun. Technol. Res.*, vol. 1, no. 3, p. 8, 2011.

[36] Y. Wang and V. Chiew, “Empirical Studies on the Functional Complexity of Software in Large-Scale Software Systems,” p. 20, 2011.

[37] T. J. McCabe, “A complexity measure,” *IEEE Trans. Softw. Eng.*, vol. 2, no. 4, pp. 308–320, 1976, doi: 10.1109/TSE.1976.233837.

[38] S. Dissem, E. Pregerson, A. Bhargava, J. Cordova, and L. Bang, “Path Complexity Correlates with Source Code Comprehension Effort Indicators,” in *2023 IEEE/ACM 31st International Conference on Program Comprehension (ICPC)*, Melbourne, Australia: IEEE, May 2023, pp. 266–274. doi: 10.1109/ICPC58990.2023.00041.

[39] M. E. Hansen, A. Lumsdaine, and R. L. Goldstone, “An experiment on the cognitive complexity of code,” in *Proceedings of the Thirty-Fifth Annual Conference of the Cognitive Science Society, Berlin, Germany*, 2013.

[40] J. Rilling and T. Klemola, “Identifying comprehension bottlenecks using program slicing and cognitive complexity metrics,” in *11th IEEE International Workshop on Program Comprehension, 2003.*, IEEE, 2003, pp. 115–124. doi: 10.1109/WPC.2003.1199195.

[41] B. S. Alqadi and J. I. Maletic, “Slice-Based Cognitive Complexity Metrics for Defect Prediction,” in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, London, ON, Canada: IEEE, Feb. 2020, pp. 411–422. doi: 10.1109/SANER48275.2020.9054836.

[42] Y. Dai and S. Liu, “Applying Cognitive Complexity to Checklist-Based Human-Machine Pair Inspection,” in *2021 IEEE 21st International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, Hainan, China: IEEE, Dec. 2021, pp. 314–318. doi: 10.1109/QRS-C55045.2021.00054.

[43] J. P. Miguel, D. Mauricio, and G. Rodríguez, “A Review of Software Quality Models for the Evaluation of Software Products,” *Int. J. Softw. Eng. Appl.*, vol. 5, no. 6, pp. 31–53, Nov. 2014, doi: 10.5121/ijsea.2014.5603.

[44] S. Scalabrino, G. Bavota, C. Vendome, M. Linares-Vasquez, D. Poshyvanyk, and R. Oliveto, "Automatically assessing code understandability: How far are we?," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Urbana, IL: IEEE, Oct. 2017, pp. 417–427. doi: 10.1109/ASE.2017.8115654.

[45] V. Lenarduzzi, T. Kilamo, and A. Janes, "Does Cyclomatic or Cognitive Complexity Better Represents Code Understandability? An Empirical Investigation on the Developers Perception." arXiv, Mar. 14, 2023. Accessed: Jul. 26, 2023. [Online]. Available: http://arxiv.org/abs/2303.07722

[46] I. Akman, S. Misra, and F. Cafer, "The role of leadership cognitive complexity in software development projects: An empirical assessment for simple thinking," *Hum. Factors Ergon. Manuf. Serv. Ind.*, vol. 21, no. 5, pp. 516–525, Sep. 2011, doi: 10.1002/hfm.20256.

[47] A. Abbad-Andaloussi, "On the relationship between source-code metrics and cognitive load: A systematic tertiary review," *J. Syst. Softw.*, vol. 198, p. 111619, Apr. 2023, doi: 10.1016/j.jss.2023.111619.

[48] F. Stolp, "Assessing Cognitive Load in Software Development with Wearable Sensors," in *2023 IEEE/ACM 45th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, Melbourne, Australia: IEEE, May 2023, pp. 227–229. doi: 10.1109/ICSE-Companion58688.2023.00062.

[49] G. Hao *et al.*, "On the accuracy of code complexity metrics: A neuroscience-based guideline for improvement," *Front. Neurosci.*, vol. 16, p. 1065366, Feb. 2023, doi: 10.3389/fnins.2022.1065366.

[50] V. Gruhn and R. Laue, "Reducing the cognitive complexity of business process models," in *2009 8th IEEE International Conference on Cognitive Informatics*, Kowloon, Hong Kong: IEEE, Jun. 2009, pp. 339–345. doi: 10.1109/COGINF.2009.5250717.

[51] V. Gruhn and R. Laue, "Complexity Metrics for Business Process Models," *Bus. Inf. Syst.*, p. 12, 2006.

[52] B. Hilburn, "COGNITIVE COMPLEXITY IN AIR TRAFFIC CONTROL: A LITERATURE REVIEW," p. 81, 2004.

[53] G. Andrews and G. S. Halford, "A cognitive complexity metric applied to cognitive development," *Cognit. Psychol.*, vol. 45, no. 2, pp. 153–219, Sep. 2002, doi: 10.1016/S0010-0285(02)00002-6.

[54] Q.-F. Yang, S.-C. Chang, G.-J. Hwang, and D. Zou, "Balancing cognitive complexity and gaming level: Effects of a cognitive complexity-based competition game on EFL students' English vocabulary learning performance, anxiety and behaviors," *Comput. Educ.*, vol. 148, p. 103808, Apr. 2020, doi: 10.1016/j.compedu.2020.103808.

[55] A. Presbitero, "Proactivity in career development of employees: The roles of proactive personality and cognitive complexity," *Career Dev. Int.*, vol. 20, no. 5, pp. 525–538, Sep. 2015, doi: 10.1108/CDI-03-2015-0043.

[56] J. E. Silva, P. Ferreira, J. L. Coimbra, and I. Menezes, "Theater and Psychological Development: Assessing Socio-Cognitive Complexity in the Domain of Theater," *Creat. Res. J.*, vol. 29, no. 2, pp. 157–166, Apr. 2017, doi: 10.1080/10400419.2017.1302778.

[57] S. Misra and I. Akman, "Applicability of Weyuker's properties on OO metrics: Some misunderstandings," *Comput. Sci. Inf. Syst.*, vol. 5, no. 1, pp. 17–23, 2008, doi: 10.2298/CSIS0801017M.

[58] L. C. Briand and S. Morasca, “Property Based Software Engineering Measurement,” *IEEE Trans. Softw. Eng.*, vol. 22, no. 1, pp. 68–86, Jan. 1996, doi: 10.1109/32.481535.

[59] S. Misra, “An Object Oriented Complexity Metric Based on Cognitive Weights,” in *6th IEEE International Conference on Cognitive Informatics*, IEEE, Aug. 2007, pp. 134–139. doi: 10.1109/COGINF.2007.4341883.

[60] D. S. Kushwaha and A. K. Misra, “A modified cognitive information complexity measure of software,” *ACM SIGSOFT Softw. Eng. Notes*, vol. 31, no. 1, pp. 1–4, Jan. 2006, doi: 10.1145/1108768.1108776.

[61] A. K. Misra, “Evaluating cognitive complexity measure with Weyuker properties,” in *Proceedings of the Third IEEE International Conference on Cognitive Informatics, 2004.*, Victoria, BC, Canada: IEEE, 2004, pp. 103–108. doi: 10.1109/COGINF.2004.1327464.

[62] V. Gupta and J. K. Chhabra, “Object-Oriented Cognitive-Spatial Complexity Measures,” *World Acad. Sci. Eng. Technol.*, vol. 3, no. 3, p. 8, 2009, doi: doi.org/10.5281/zenodo.1072347.

[63] J. K. Chhabra and V. Gupta, “Evaluation of object-oriented spatial complexity measures,” *ACM SIGSOFT Softw. Eng. Notes*, vol. 34, no. 3, pp. 1–5, May 2009, doi: 10.1145/1527202.1527208.

[64] S. Misra and K. I. Akman, “Weighted Class Complexity: A Measure of Complexity for Object Oriented System,” *J. Inf. Sci. Eng.*, vol. 24, pp. 1689–1708, 2008.

[65] R. Damasevicius and V. Stuikys, “Metrics for evaluation of metaprogram complexity,” *Comput. Sci. Inf. Syst.*, vol. 7, no. 4, pp. 769–787, 2010, doi: 10.2298/CSIS090315004D.

[66] N. Cowan, “What are the differences between long-term, short-term, and working memory?,” *Prog. Brain Res.*, vol. 169, pp. 323–338, 2008, doi: 10.1016/S0079-6123(07)00020-9.

[67] M. Klepsch and T. Seufert, “Understanding instructional design effects by differentiated measurement of intrinsic, extraneous, and germane cognitive load,” *Instr. Sci.*, vol. 48, no. 1, pp. 45–77, Feb. 2020, doi: 10.1007/s11251-020-09502-9.

[68] K. E. DeLeeuw and R. E. Mayer, “A comparison of three measures of cognitive load: Evidence for separable measures of intrinsic, extraneous, and germane load.,” *J. Educ. Psychol.*, vol. 100, no. 1, pp. 223–234, Feb. 2008, doi: 10.1037/0022-0663.100.1.223.

[69] F. Détienne, *Software Design — Cognitive Aspects*. in Practitioner Series. London: Springer London, 2002. doi: 10.1007/978-1-4471-0111-6.

[70] I. Heitlager, T. Kuipers, and J. Visser, “A Practical Model for Measuring Maintainability,” in *6th International Conference on the Quality of Information and Communications Technology (QUATIC 2007)*, Lisbon, Portugal: IEEE, Sep. 2007, pp. 30–39. doi: 10.1109/QUATIC.2007.8.

[71] L. Pascarella, M. Bruntink, and A. Bacchelli, “Classifying code comments in Java software systems,” *Empir. Softw. Eng.*, vol. 24, no. 3, pp. 1499–1537, Jun. 2019, doi: 10.1007/s10664-019-09694-w.

[72] D. Gopstein *et al.*, “Understanding misunderstandings in source code,” in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, Paderborn Germany: ACM, Aug. 2017, pp. 129–139. doi: 10.1145/3106237.3106264.

[73] A. Yamashita and S. Counsell, “Code smells as system-level indicators of maintainability: An empirical study,” *J. Syst. Softw.*, vol. 86, no. 10, pp. 2639–2653, Oct. 2013, doi: 10.1016/j.jss.2013.05.007.

[74] F. Palomba, G. Bavota, M. D. Penta, F. Fasano, R. Oliveto, and A. D. Lucia, “On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation,” *Empir. Softw. Eng.*, vol. 23, no. 3, pp. 1188–1221, Jun. 2018, doi: 10.1007/s10664-017-9535-z.

[75] R. Baggen, J. P. Correia, K. Schill, and J. Visser, “Standardized code quality benchmarking for improving software maintainability,” *Softw. Qual. J.*, vol. 20, no. 2, pp. 287–307, Jun. 2012, doi: 10.1007/s11219-011-9144-9.

[76] H. Alsolai and M. Roper, “A systematic literature review of machine learning techniques for software maintainability prediction,” *Inf. Softw. Technol.*, vol. 119, p. 106214, Mar. 2020, doi: 10.1016/j.infsof.2019.106214.

[77] J. Lang and D. Spišák, “Activity Diagram as an Orientation Catalyst within Source Code,” *Acta Polytech. Hung.*, vol. 18, no. 3, pp. 127–146, 2021, doi: 10.12700/APH.18.3.2021.3.7.

[78] R. Bloem *et al.*, “RATSY – A New Requirements Analysis Tool with Synthesis,” in *Computer Aided Verification*, T. Touili, B. Cook, and P. Jackson, Eds., in Lecture Notes in Computer Science, vol. 6174. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 425–429. doi: 10.1007/978-3-642-14295-6_37.

[79] P. K. Ragunath, S. Velmourougan, P. Davachelvan, S. Kayalvizhi, and R. Ravimohan, “Evolving a new model (SDLC Model-2010) for software development life cycle (SDLC),” *Int. J. Comput. Sci. Netw. Secur.*, vol. 10, no. 1, pp. 112–119, 2010.

[80] L. Marquez, H. Rodriguez, J. Carmona, and J. Montolio, “Improving POS Tagging Using Machine-Learning Techniques,” pp. 53–62, 2002.

[81] L. M. Rquez, L. Padro, and H. Rodriguez, “A Machine Learning Approach to POS Tagging,” *Mach. Learn.*, vol. 39, pp. 59–91, 2000, doi: https://doi.org/10.1023/A:1007673816718.

[82] B. Litvak, S. Tyszberowicz, and A. Yehudai, “Behavioral consistency validation of UML diagrams,” in *First International Conference onSoftware Engineering and Formal Methods, 2003. Proceedings.*, IEEE, 2003, pp. 118–125.

[83] K. H. Heung, “A tool for generating UML diagram from source code,” *Proc. 2009 IEEEACM Int. Conf. Autom. Softw. Eng.*, pp. 680–682, Nov. 2009, doi: https://doi.org/10.1109/ASE.2009.48.

[84] D. K. Deeptimahanti and M. A. Babar, “An Automated Tool for Generating UML Models from Natural Language Requirements,” in *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, in ASE ’09. USA: IEEE Computer Society, Nov. 2009, pp. 680–682. doi: 10.1109/ASE.2009.48.

[85] J. Pennington, R. Socher, and C. Manning, “Glove: Global Vectors for Word Representation,” in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, Doha, Qatar: Association for Computational Linguistics, 2014, pp. 1532–1543. doi: 10.3115/v1/D14-1162.

[86] B. Li and L. Han, “Distance Weighted Cosine Similarity Measure for Text Classification,” in *Intelligent Data Engineering and Automated Learning – IDEAL 2013*, H. Yin, K. Tang, Y. Gao, F.

Klawonn, M. Lee, T. Weise, B. Li, and X. Yao, Eds., in Lecture Notes in Computer Science, vol. 8206. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 611–618. doi: 10.1007/978-3-642-41278-3_74.

[87] M. J. Egenhofer and A. U. Frank, "Object-Oriented Modeling in CIS: Inheritance and Propagation," p. 11, 2008.

[88] D. Dey, V. C. Storey, and T. M. Barron, "Improving database design through the analysis of relationships," *ACM Trans. Database Syst.*, vol. 24, no. 4, pp. 453–486, Dec. 1999, doi: 10.1145/331983.331984.

[89] X. Renguo, T. S. Dillon, W. Rahayu, E. Chang, and N. Gorla, "An Indexing Structure for Aggregation Relationship in OODB," in *Database and Expert Systems Applications*, M. Ibrahim, J. Küng, and N. Revell, Eds., in Lecture Notes in Computer Science, vol. 1873. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 21–30. doi: 10.1007/3-540-44469-6_3.

[90] J. Han and Y. Fu, "Discovery of Multiple-Level Association Rules from Large Databases," *Proceddings 21st VLDB Conf.*, p. 12, 1995.

[91] W. Ben Slama Souei, C. El Hog, L. Sliman, R. Ben Djemaa, and I. A. Ben Amor, "Towards a Uniform Description Language for Smart Contract," in *2021 IEEE 30th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*, Bayonne, France: IEEE, Oct. 2021, pp. 57–62. doi: 10.1109/WETICE53228.2021.00022.

[92] N. Rutar, C. B. Almazan, and J. S. Foster, "A Comparison of Bug Finding Tools for Java," in *15th International Symposium on Software Reliability Engineering*, Saint-Malo, Bretagne, France: IEEE, 2004, pp. 245–256. doi: 10.1109/ISSRE.2004.1.

[93] K. Herzig, S. Just, and A. Zeller, "It's not a bug, it's a feature: How misclassification impacts bug prediction," in *2013 35th International Conference on Software Engineering (ICSE)*, San Francisco, CA, USA: IEEE, May 2013, pp. 392–401. doi: 10.1109/ICSE.2013.6606585.

[94] A. Yamashita and L. Moonen, "Do code smells reflect important maintainability aspects?," in *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, Trento, Italy: IEEE, Sep. 2012, pp. 306–315. doi: 10.1109/ICSM.2012.6405287.

[95] T. Zimmermann and A. Casanueva Artis, "Impact of Switching Bug Trackers: A Case Study on a Medium-Sized Open Source Project," in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Cleveland, OH, USA: IEEE, Sep. 2019, pp. 13–23. doi: 10.1109/ICSME.2019.00011.

[96] D. Hovemeyer and W. Pugh, "Finding Bugs is Easy," *Static Anal.*, p. 49, Oct. 2004.

[97] G. Bortis and A. van der Hoek, "PorchLight: A tag-based approach to bug triaging," in *2013 35th International Conference on Software Engineering (ICSE)*, San Francisco, CA, USA: IEEE, May 2013, pp. 342–351. doi: 10.1109/ICSE.2013.6606580.

[98] H. Keuning, B. Heeren, and J. Jeuring, "Code Quality Issues in Student Programs," in *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education*, Bologna Italy: ACM, Jun. 2017, pp. 110–115. doi: 10.1145/3059009.3059061.

[99] X. Xia, L. Bao, D. Lo, Z. Xing, A. E. Hassan, and S. Li, "Measuring Program Comprehension: A Large-Scale Field Study with Professionals," *IEEE Trans. Softw. Eng.*, vol. 44, no. 10, pp. 951–976, Oct. 2018, doi: 10.1109/TSE.2017.2734091.

[100] B. Seref and O. Tanriover, “Software Code Maintainability : A Literature Review,” *Int. J. Softw. Eng. Appl.*, vol. 7, no. 3, pp. 69–87, May 2016, doi: 10.5121/ijsea.2016.7305.

[101] C. Chen, R. Alfayez, K. Srisopha, B. Boehm, and L. Shi, “Why Is It Important to Measure Maintainability and What Are the Best Ways to Do It?,” in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, Buenos Aires, Argentina: IEEE, May 2017, pp. 377–378. doi: 10.1109/ICSE-C.2017.75.

[102] M. Kim, T. Zimmermann, and N. Nagappan, “A field study of refactoring challenges and benefits,” in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering - FSE ’12*, Cary, North Carolina: ACM Press, 2012, p. 1. doi: 10.1145/2393596.2393655.

[103] S. B. Green and N. J. Salkind, “Using SPSS for Windows and Macintosh : analyzing and understanding data,” *CERN Document Server*, 2012. https://cds.cern.ch/record/1564874 (accessed Mar. 22, 2021).

[104] V. Damasiotis, P. Fitsilis, P. Considine, and J. O’Kane, “Analysis of software project complexity factors,” in *Proceedings of the 2017 International Conference on Management Engineering, Software Engineering and Service Sciences - ICMSS ’17*, Wuhan, China: ACM Press, 2017, pp. 54–58. doi: 10.1145/3034950.3034989.

[105] R. D. Banker, S. M. Datar, and D. Zweig, “Software complexity and maintainability,” in *Proceedings of the tenth international conference on Information Systems - ICIS ’89*, Boston, Massachusetts, United States: ACM Press, 1989, pp. 247–255. doi: 10.1145/75034.75056.

[106] J. García-Muñoz, M. García-Valls, and J. Escribano-Barreno, “Improved Metrics Handling in SonarQube for Software Quality Monitoring,” 2016, pp. 463–470. doi: 10.1007/978-3-319-40162-1_50.

[107] N. Shi and R. Olsson, “Reverse Engineering of Design Patterns from Java Source Code,” in *21st IEEE/ACM International Conference on Automated Software Engineering (ASE’06)*, Tokyo: IEEE, 2006, pp. 123–134. doi: 10.1109/ASE.2006.57.

[108] T. Ishio, H. Date, T. Miyake, and K. Inoue, “Mining Coding Patterns to Detect Crosscutting Concerns in Java Programs,” in *2008 15th Working Conference on Reverse Engineering*, Antwerp, Belgium: IEEE, Oct. 2008, pp. 123–132. doi: 10.1109/WCRE.2008.28.

[109] P. G. Armour, “Beware of counting LOC,” *Commun. ACM*, vol. 47, no. 3, pp. 21–24, Mar. 2004, doi: 10.1145/971617.971635.

[110] D. Kelly, J. Arguello, A. Edwards, and W. Wu, “Development and Evaluation of Search Tasks for IIR Experiments using a Cognitive Complexity Framework,” in *Proceedings of the 2015 International Conference on The Theory of Information Retrieval*, Northampton Massachusetts USA: ACM, Sep. 2015, pp. 101–110. doi: 10.1145/2808194.2809465.

[111] A. J. Bishara and J. B. Hittner, “Testing the significance of a correlation with nonnormal data: Comparison of Pearson, Spearman, transformation, and resampling approaches.,” *Psychol. Methods*, vol. 17, no. 3, pp. 399–417, Sep. 2012, doi: 10.1037/a0028087.

[112] S. Gharsellaoui, M. Mansouri, S. S. Refaat, H. Abu-Rub, and H. Messaoud, “Multivariate Features Extraction and Effective Decision Making Using Machine Learning Approaches,” *Energies*, vol. 13, no. 3, p. 609, Jan. 2020, doi: 10.3390/en13030609.

[113] A. Ashari, I. Paryudi, and A. Min, “Performance Comparison between Naïve Bayes, Decision Tree and k-Nearest Neighbor in Searching Alternative Design in an Energy Simulation Tool,” *Int. J. Adv. Comput. Sci. Appl.*, vol. 4, no. 11, 2013, doi: 10.14569/IJACSA.2013.041105.

[114] Ali Haghpanah Jahromi and M. Taheri, “A non-parametric mixture of Gaussian naive Bayes classifiers based on local independent features,” presented at the 2017 Artificial Intelligence and Signal Processing Conference (AISP), IEEE, pp. 209–212.

[115] G. A. F. Seber and A. J. Lee, *Linear Regression Analysis*. John Wiley & Sons, 2012.

[116] E. Morozoff, “Using a Line of Code Metric to Understand Software Rework,” *IEEE Softw.*, vol. 27, no. 1, pp. 72–77, Jan. 2010, doi: 10.1109/MS.2009.160.

[117] D. Steidl, B. Hummel, and E. Juergens, “Quality analysis of source code comments,” in *2013 21st International Conference on Program Comprehension (ICPC)*, San Francisco, CA, USA: IEEE, May 2013, pp. 83–92. doi: 10.1109/ICPC.2013.6613836.

[118] G. Rakić, M. Tóth, and Z. Budimac, “Toward recursion aware complexity metrics,” *Inf. Softw. Technol.*, vol. 118, p. 106203, Feb. 2020, doi: 10.1016/j.infsof.2019.106203.

[119] M. M. Suleman Sarwar, S. Shahzad, and I. Ahmad, “Cyclomatic complexity: The nesting problem,” in *Eighth International Conference on Digital Information Management (ICDIM 2013)*, Islamabad, Pakistan: IEEE, Sep. 2013, pp. 274–279. doi: 10.1109/ICDIM.2013.6693981.

[120] C. T. Bailey and W. L. Dingee, “A software study using Halstead metrics,” in *Proceedings of the 1981 ACM workshop/symposium on Measurement and evaluation of software quality*, 1981, pp. 189–197.

[121] P. Robinson, “Cognitive Complexity and Task Sequencing: Studies in a Componential Framework for Second Language Task Design,” *IRAL - Int. Rev. Appl. Linguist. Lang. Teach.*, vol. 43, no. 1, pp. 1–32, Jan. 2005, doi: 10.1515/iral.2005.43.1.1.

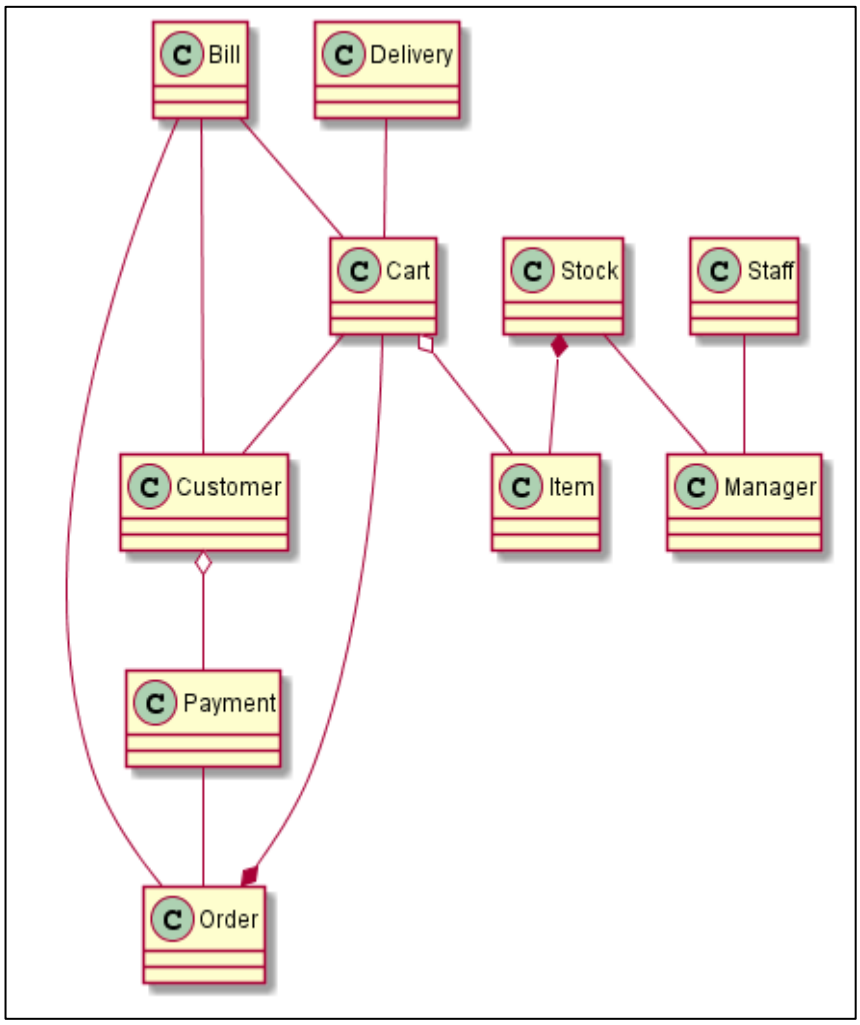
[122] A. Piolat, T. Olive, and R. T. Kellogg, “Cognitive effort during note taking,” *Appl. Cogn. Psychol.*, vol. 19, no. 3, pp. 291–312, Apr. 2005, doi: 10.1002/acp.1086.

[123] K. P. Srinivasan and T. Devi, “Software Metrics Validation Methodologies in Software Engineering,” *Int. J. Softw. Eng. Appl.*, vol. 5, no. 6, pp. 87–102, Nov. 2014, doi: 10.5121/ijsea.2014.5606.

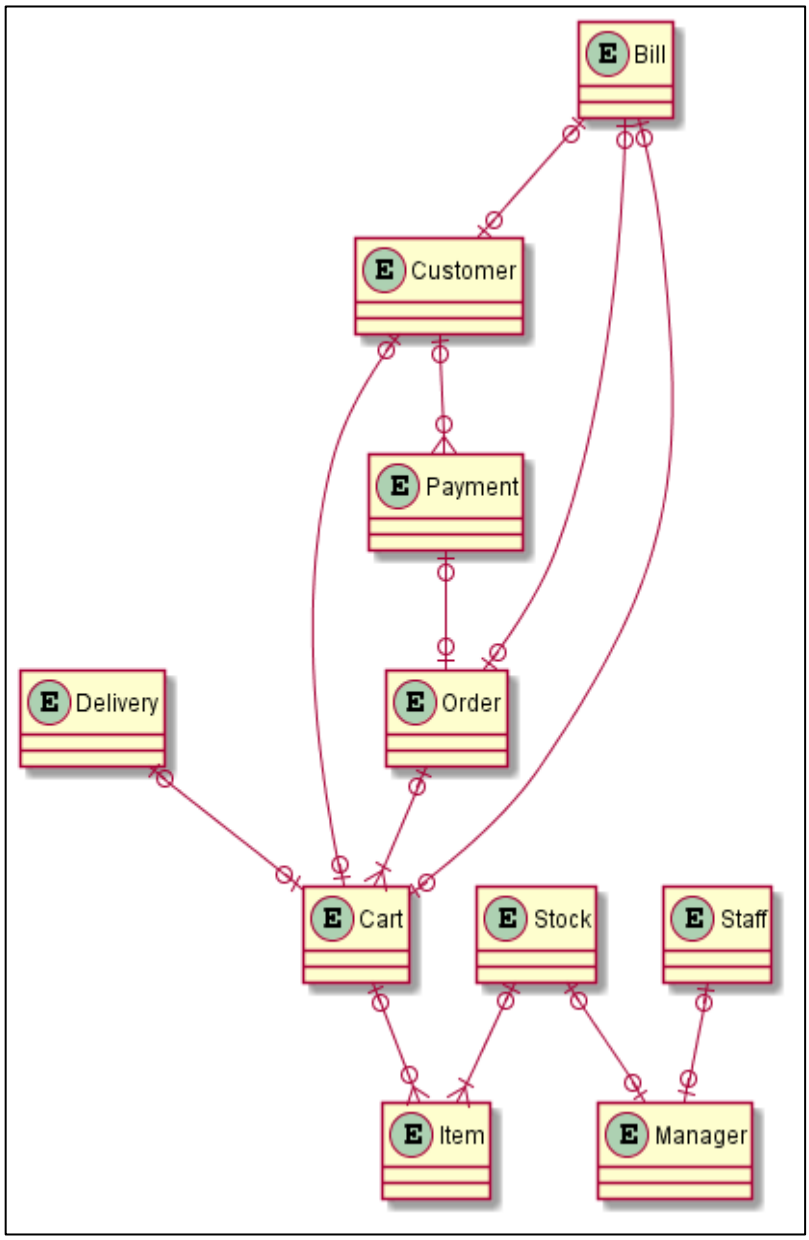
Appendices

Appendix A – Logical Visualizations without the Source Code

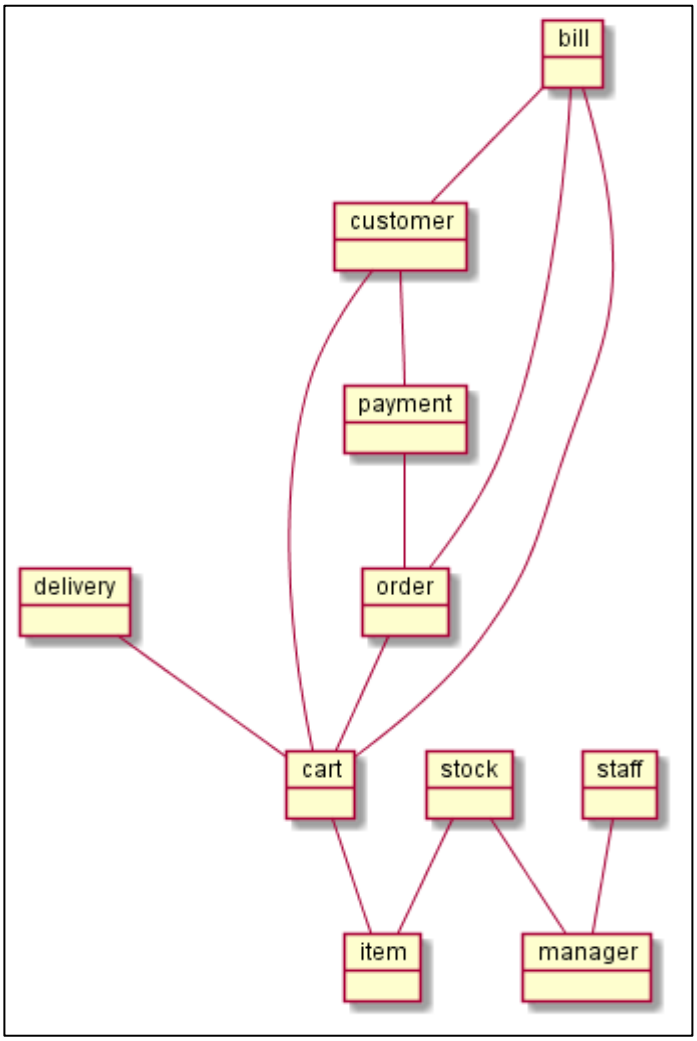
Sample Class Diagram



Sample ER Diagram

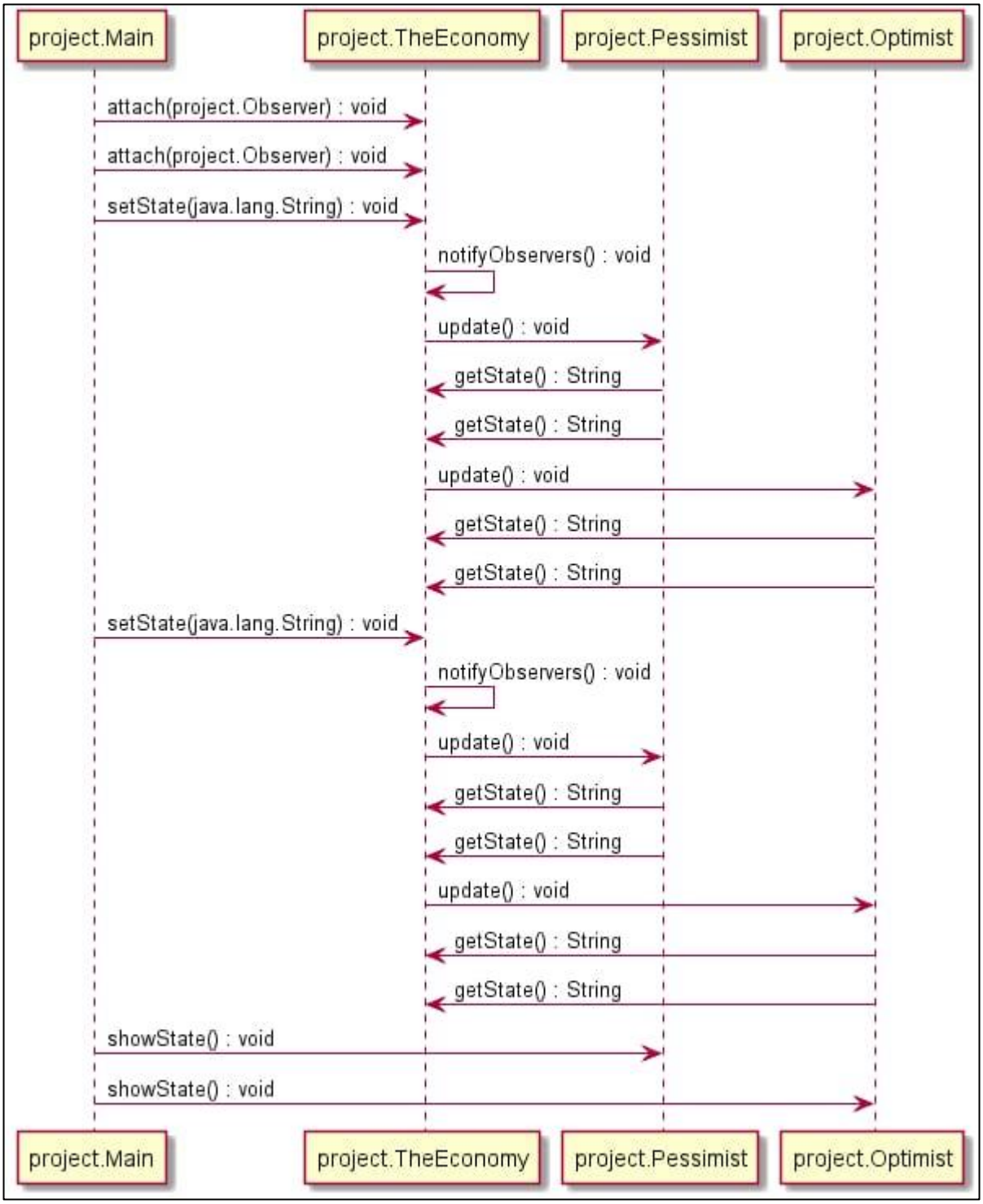


Sample Object Diagram

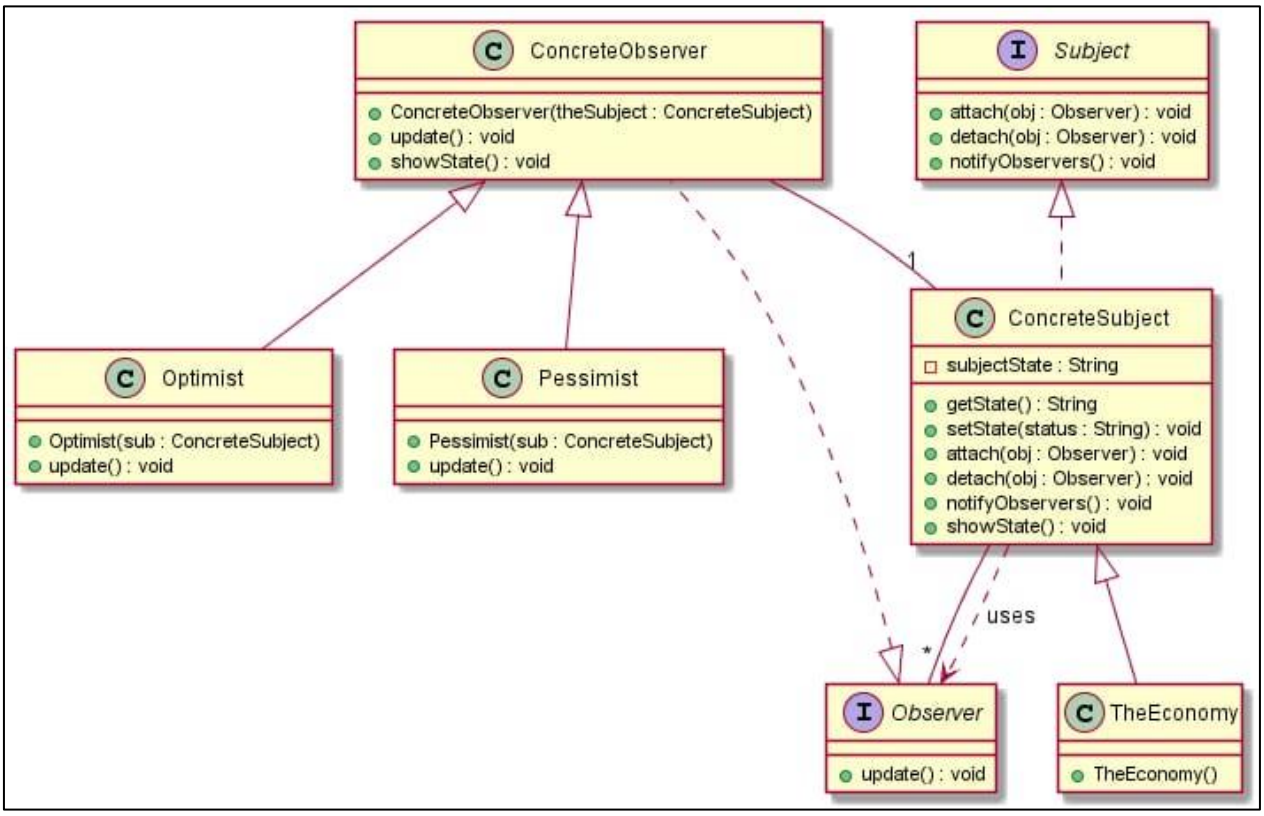


Appendix B – Logical Visualizations using the Source Code

Sample Sequence Diagram



Sample Class Diagram



Appendix C – SPSS Statistical Outcomes Derived from BCS Questionnaire

Statistics for *if-else* and *switch-case* statements based on Time

Paired Samples Statistics

		Mean	N	Std. Deviation	Std. Error Mean
Pair 1	If_Total_Time	6.8377	500	5.64287	.25236
	Switch_Total_Time	3.4622	500	3.35923	.15023

Statistics for *if-else* and *switch-case* statements based on Marks

Paired Samples Statistics

		Mean	N	Std. Deviation	Std. Error Mean
Pair 1	If_Mark	4.63	500	.809	.036
	Switch_Mark	4.71	500	.807	.036

Statistics for *for* and *while* loops based on Time

Paired Samples Statistics

		Mean	N	Std. Deviation	Std. Error Mean
Pair 1	For_Total_Time	6.3616	500	4.82534	.21580
	While_Total_Time	4.0950	500	2.94839	.13186

Statistics for *for* and *while* loops based on Marks

Paired Samples Statistics

		Mean	N	Std. Deviation	Std. Error Mean
Pair 1	For_Marks	4.2960	500	1.03078	.04610
	While_Marks	4.3640	500	1.03617	.04634

Statistics for *nested for* and *nested while* loops based on Time

Paired Samples Statistics

		Mean	N	Std. Deviation	Std. Error Mean
Pair 1	NestedFor_Total_Time	8.1471	500	6.47033	.28936
	NestedWhile_Total_Time	5.8110	500	5.09571	.22789

Statistics for *nested for* and *nested while* loops based on Marks

Paired Samples Statistics

		Mean	N	Std. Deviation	Std. Error Mean
Pair 1	NestedFor_Marks	3.6920	500	1.39032	.06218
	NestedWhile_Marks	3.9260	500	1.37561	.06152

Appendix D - BCS Questionnaire

Questionnaire Given to Test *if-else* Statements under Conditional Statements Category

Question 1 Not answered Marked out of 1.00

What is the output of the following program?

```
1  public class Test {
2    public static void main(String args[]) {
3        int x = 40;
4        if( x == 10 ) {
5           System.out.print("Value of X is 10");
6        }else if( x == 20 ) {
7           System.out.print("Value of X is 20");
8        }else if( x == 30 ) {
9           System.out.print("Value of X is 30");
10       }else {
11          System.out.print("This is else statement");
12       }
13    }
14 }
```

a. Value of X is 30

b. Value of X is 20

c. This is else statement

d. None of the mentioned

e. Value of X is 10

Question 2

Not answered Marked out of 1.00

What is the output of the following program?

```
1  public class IfExample2 {
2    public static void main(String[] args) {
3
4        int number=20;
5        if(number ==10)
6            System.out.println("10");
7        else if(number ==20)
8            System.out.println("20");
9        else if(number==30)
10           System.out.println("30");
11       else
12           System.out.println("Not in 10, 20 or 30");
13       }
14 }
15
```

a. 0

b. None of the mentioned

c. 20

d. 10

e. 30

Question 3

Not answered Marked out of 1.00

What is the output of the following program?

```
public class IfDemo {
    public static void main(String[] args) {
        int month =8;
        if(month ==1)
            monthString = "January";
        else if(month ==2)
            monthString = "February";
        else if(month ==3)
            monthString = "March";
        else if(month ==4)
            monthString = "April";
        else if(month ==5)
            monthString = "May";
        else if(month ==6)
            monthString = "June";
        else if(month ==7)
            monthString = "July";
        else if(month ==8)
            monthString = "August";
        else if(month ==9)
            monthString = "September";
        else if(month ==10)
            monthString = "October";
        else if(month ==11)
            monthString = "November";
        else if(month ==12)
            monthString = "December";
        else
            monthString = "Invalid month";

        System.out.println(monthString);
}
}
```

a. August

b. December

c. Invalid month

d. November

e. July

Question 4

Not answered Marked out of 1.00

What is the output of the following program?

```
1  public class Test {
2     public static void main(String args[]) {
3          char grade = 'D';
4          int success;
5          if(grade=='A'){
6                System.out.println("Excellent grade");
7                success=1;
8          }
9          else if (grade=='B'){
10               System.out.println("Very good grade");
11               success=1;
12         }
13         else if (grade=='C'){
14               System.out.println("Good grade");
15               success=1;
16         }
17         else if (grade=='D'|| grade=='E'|| grade=='F'){
18               System.out.println("Low grade");
19               success=0;
20         }
21         else {
22               System.out.println("Invalid grade");
23               success=-1;
24         }
25         passTheCourse(success);
26   }
27      public static void passTheCourse(int success) {
28          if(success == -1)
29              System.out.println("No result");
30          else if (success == 0)
31              System.out.println("Final result: Fail");
32          else if (success ==1)
33              System.out.println("Final result: Success");
34          else
35              System.out.println("Unknown result");
36
37      }
38  }
39
```

a. Low grade

   Final result: fail

b. None of the mentioned

c. Low grade

   Unknown result

d. Invalid grade

e. Low grade

   No result

Question 5

Not answered Marked out of 1.00

What is the output of the following program?

```
1  public class Test {
2     public static void main(String args[]) {
3        char grade = 'C';
4        if(grade== 'A'){
5              System.out.println("Excellent!");
6        else if(grade == 'B' || grade == 'C')
7              System.out.println("Well done");
8        else if (grade == 'D')
9              System.out.println("You passed");
10       else if (grade == 'F')
11             System.out.println("Better try again");
12       else
13             System.out.println("Invalid grade");
14         }
15       System.out.println("Your grade is " + grade);
16    }
17 }
18
```

a. Excellent!

Your grade is C

b. You passed

Your grade is C

c. Invalid grade

Your grade is C

d. None of the mentioned

e. Well done

Your grade is C

Question 1 Not answered Marked out of 1.00

What is the output of the following program?

```
public class SwitchDemo {
    public static void main(String[] args) {
        int month = 8;
        String monthString;
        switch (month) {
            case 1:  monthString = "January";
                     break;
            case 2:  monthString = "February";
                     break;
            case 3:  monthString = "March";
                     break;
            case 4:  monthString = "April";
                     break;
            case 5:  monthString = "May";
                     break;
            case 6:  monthString = "June";
                     break;
            case 7:  monthString = "July";
                     break;
            case 8:  monthString = "August";
                     break;
            case 9:  monthString = "September";
                     break;
            case 10: monthString = "October";
                     break;
            case 11: monthString = "November";
                     break;
            case 12: monthString = "December";
                     break;
            default: monthString = "Invalid month";
                     break;
        }
        System.out.println(monthString);
    }
}

```

a. July

b. November

c. Invalid month

d. August

e. December

Question 2

Not answered Marked out of 1.00

What is the output of the following program?

```
public class Test {
   public static void main(String args[]) {
      int x = 40;
      switch(x){
        case 10:
          System.out.print("Value of X is 10");
          break;
        case 20:
          System.out.print("Value of X is 20");
          break;
        case 30:
          System.out.print("Value of X is 30");
          break;
        default:
          System.out.print("This is else statement");
          break;
      }
   }
}
```

a. Value of X is 30

b. Value of X is 10

c. Value of X is 0

d. This is else statement

e. Value of X is 20

Question 3

Not answered Marked out of 1.00

What is the output of the following program?

```
public class Test {
   public static void main(String args[]) {
      char grade = 'D';
      int success;
      switch(grade)
      {
          case 'A':
              System.out.println("Excellent grade");
              success= 1;
              break;
          case 'B':
              System.out.println("Very good grade");
              success= 1;
              break;
          case 'C':
              System.out.println("Good grade");
              success= 1;
              break;
          case 'D':
          case 'E':
          case 'F':
              System.out.println("Low grade");
              success= 0;
              break;
          default:
              System.out.println("Invalid grade");
              success= -1;
              break;
      }passTheCourse(success);
  }
      public static void passTheCourse(int success) {
          switch (success) {
              case -1:
                  System.out.println("No result");
                  break;
              case 0:
                  System.out.println("Final result: Fail");
                  break;
              case 1:
                  System.out.println("Final result: Success");
                  break;
              default:
                  System.out.println("Unknown result");
                  break;
          }
}    }
```

a. Low grade

   Final result: fail

b. Invalid grade

c. Low grade

   No result

d. Low grade

Unknown result

e. None of the mentioned

Question 4

Not answered Marked out of 1.00

What is the output of the following program?

```
public class Test {
   public static void main(String args[]) {
      char grade = 'C';
      switch(grade) {
         case 'A' :
            System.out.println("Excellent!");
            break;
         case 'B' :
         case 'C' :
            System.out.println("Well done");
            break;
         case 'D' :
            System.out.println("You passed");
         case 'F' :
            System.out.println("Better try again");
            break;
         default :
            System.out.println("Invalid grade");
      }
      System.out.println("Your grade is " + grade);
   }
}
```

a. Excellent!

Your grade is C

b. You passed

Your grade is C

c. Well done

Your grade is C

d. Invalid grade

Your grade is C

e. None of the mentioned

Question 5

Not answered Marked out of 1.00

What is the output of the following program?

```
public class SwitchExample2 {
  public static void main(String[] args) {

      int number=20;
      switch(number){
        case 10: System.out.println("10");
          break;
        case 20: System.out.println("20");
          break;
        case 30: System.out.println("30");
          break;
        default:System.out.println("Not in 10, 20 or 30");
      }
    }
}
```

a. 10

b. 0

c. 20

d. 30

e. None of the mentioned

Question 1 Not answered Marked out of 1.00

What is the output of the following program?

```
class ForLoopExample {
    public static void main(String args[]){
        for(int i=10; i>1; i--){
            System.out.println("The value of i is: "+i);
        }
    }
}
```

a. The value of i is: 1
The value of i is: 2
The value of i is: 3
The value of i is: 4
The value of i is: 5
The value of i is: 6
The value of i is: 7
The value of i is: 8
The value of i is: 9

b. None of the mentioned

c. The value of i is: 10
The value of i is: 10
The value of i is: 10
The value of i is: 10
The value of i is: 10
The value of i is: 10
The value of i is: 10
The value of i is: 10
The value of i is: 10

d. The value of i is: 10
The value of i is: 9
The value of i is: 8

The value of i is: 7

The value of i is: 6

The value of i is: 5

The value of i is: 4

The value of i is: 3

The value of i is: 2

e. The value of i is: 10

The value of i is: 9

The value of i is: 8

The value of i is: 7

The value of i is: 5

The value of i is: 4

The value of i is: 3

Question 2 Not answered Marked out of 1.00

What is the output of the following program?

```
class ForLoopExample2 {
    public static void main(String args[]){
        for(int i=1; i>=1; i++){
            System.out.println("The value of i is: "+i);
        }
    }
}
```

a. None of the mentioned

b. 1
1
1
1
1
1
1
1
1
1

c. 1

d. 1
1
1
1
1

e. It will go on an infinite loop

Question 3

Not answered Marked out of 1.00

What is the output of the following program?

```
class ForLoopExample2 {
    public static void main(String args[]){
    for(int count=100;count<100;count++)
        System.out.println("The value of count is: "+count);

    }
}
```

a. 100

b. 100

100

100

100

c. No iteration happens

d. It will go on an infinite loop

e. None of the mentioned

Question 4

Not answered Marked out of 1.00

What is the output of the following program?

```
class ForLoopExample1 {
    public static void main(String args[]){
        int []arr=new int[4];
        arr[0]=2;
        arr[1]=11;
        arr[2]=45;
        arr[3]=9;
        for(int i=0;i<4;i++)
            System.out.println(arr[i]);
    }
}
```

a. 2
3
11
9

b. 2
11
45
9

c. 9
45
11
2

d. 0
1
2
3

e. None of the mentioned

Question 5

Not answered Marked out of 1.00

What is the output of the following program?

```
class ForDemo {
    public static void main(String[] args){
        for(int i=1; i<11; i++){
            System.out.println("Count is: " + i);
        }
    }
}
```

a. Count is: 1
Count is: 2
Count is: 3
Count is: 4
Count is: 5
Count is: 6
Count is: 7
Count is: 8
Count is: 9
Count is: 10

b. Count is: 1
Count is: 2
Count is: 3
Count is: 4
Count is: 6
Count is: 7
Count is: 8
Count is: 9
Count is: 10

c. Count is: 1
Count is: 1
Count is: 1
Count is: 1
Count is: 1

Count is: 1

Count is: 1

Count is: 1

Count is: 1

Count is: 1

d.

None of the mentioned

e. Count is: 10

Count is: 9

Count is: 8

Count is: 7

Count is: 6

Count is: 5

Count is: 4

Count is: 3

Count is: 2

Count is: 1

Questionnaire Given to Test *while* Loop under Looping Category

Question 1 Not answered Marked out of 1.00

What is the output of the following program?

```
class WhileDemo {
    public static void main(String[] args){
    int i =0;
    while(i<11)
    {
        System.out.println("Count is: " + i);
        i++;
    }

    }
}
```

a. Count is: 10
Count is: 9
Count is: 8
Count is: 7
Count is: 6
Count is: 5
Count is: 4
Count is: 3
Count is: 2
Count is: 1

b. None of the mentioned

c. Count is: 1
Count is: 3
Count is: 3
Count is: 5
Count is: 5
Count is: 7
Count is: 7
Count is: 9
Count is: 9
Count is: 11

d. Count is: 1

Count is: 2

Count is: 3

Count is: 4

Count is: 5

Count is: 6

Count is: 7

Count is: 8

Count is: 9

Count is: 10

e. Count is: 1

Count is: 1

Count is: 1

Count is: 1

Count is: 1

Count is: 1

Count is: 1

Count is: 1

Count is: 1

Count is: 1

Question 2

Not answered Marked out of 1.00

What is the output of the following program?

```
class WhileLoopExample1 {
    public static void main(String args[]){
            int []arr=new int[4];
        arr[0]=2;
        arr[1]=11;
         arr[2]=45;
          arr[3]=9;
        int i =0;
        while(i<4)
        {
            System.out.println(arr[i]);
            i++;
        }
        }
}
```

a. None of the mentioned

b. 9
45
11
2

c. 0
1
2
3

d. 2
11
45
9

e. 2
3
11
9

Question 3

Not answered Marked out of 1.00

What is the output of the following program?

```
class WhileLoopExample2 {
 public static void main(String args[]){
 int i=1;
 while(i>=1)
 {
 System.out.println("The value of i is: "+i);
 i++;
      }


 }
}
```

a. It will go on an infinite loop

b. 1

1

1

1

1

1

1

1

1

1

c. 1

d. None of the mentioned

e. 1

1

1

1

1

Question 4

Not answered Marked out of 1.00

What is the output of the following program?

```
class WhileLoopExample {
 public static void main(String args[]){
 int i=10;
 while(i>1)
 {
     System.out.println("The value of i is: "+i);
     i--;
 }

 }
}
```

a. None of the mentioned

b. The value of i is: 10

The value of i is: 9

The value of i is: 8

The value of i is: 7

The value of i is: 6

The value of i is: 5

The value of i is: 4

The value of i is: 3

c. The value of i is: 1

The value of i is: 2

The value of i is: 3

The value of i is: 4

The value of i is: 5

The value of i is: 6

The value of i is: 7

The value of i is: 8

The value of i is: 9

d. The value of i is: 10

The value of i is: 10

The value of i is: 10

The value of i is: 10

The value of i is: 10

The value of i is: 10

The value of i is: 10

The value of i is: 10

The value of i is: 10

e. The value of i is: 10

The value of i is: 9

The value of i is: 8

The value of i is: 7

The value of i is: 6

The value of i is: 5

The value of i is: 4

The value of i is: 3

The value of i is: 2

Question 5 Not answered Marked out of 1.00

What is the output of the following program?

```
class WhileLoopExample2 {
   public static void main(String args[]){

   int count = 100;
   while (count < 100) {
   System.out.println("The value of count is: "+count);
    count = count + 1;
   }

   }
}
```

a. 100

100

100

100

100

b. No iteration happens

c. 100

d. None of the mentioned

e. It will go on an infinite loop

Question 1 Not answered Marked out of 1.00

What is the output of the following program?

```
public class Program {
    public static void main(String[] args) {


    for(int i=1; i<=6; i++){
         for(int j=1; j<=i; j++){
            System.out.print("*");
            }
        System.out.println();
    }
}
}

```

a.
```
 *
 **
 ***
 ****
 *****
******
```

b.
```
******
 *****
  ****
   ***
    **
     *
```

c. None of the mentioned

d.
```
******
*****
****
***
**
*
```

e. *

**

***

****

*****

******

Question 2

Not answered Marked out of 1.00

What is the output of the following program?

```
1  public class Program {
2      public static void main(String[] args) {
3
4      for (int i = 0; i < 3; i++) {
5          for (int y = 0; y < 3; y++) {
6          System.out.println(i + "," + y);
7          }
8      }
9  }
10
```

a. 0,0
1,1
2,2
1,1
2,2

b. 0,0
0,1
0,2
1,0
1,1
1,2

c. 0,0
0,1
0,2
1,0
1,1
1,2
2,0
2,1

d. None of the mentioned

e. 0,1

Question 3

Not answered Marked out of 1.00

What is the output of the following program

```
1  public class Program {
2      public static void main(String[] args) {
3
4      for (int i = 1; i < 3; i++) {
5          for (int j = 0; j < 2; j++) {
6          System.out.println("six");
7          }
8      }
9      }
10 }
```

a. six
six

b. six
six
six
six
six
six

c. six
six
six

d. None of the mentioned

e. six
six
six
six

Question 4

Not answered Marked out of 1.00

What is the output of the following program?

```
public class Program {
    public static void main(String[] args) {

    for (int i = 1; i <= 5; i++) {
        for (int j = 0; j <= (5-i); j++) {
            System.out.print(" ");
        }
        for (int k = 1; k <= i; k++){
            System.out.print(i);

        }
        System.out.println();

    }
 }
}

```

a.
```
    1
   22
  333
 4444
55555
```

b.
```
1
22
333
4444
55555
```

c.
```
55555
 4444
  333
   22
    1
```

d.
```
55555
4444
333
22
1
```

e. None of the mentioned

Question 5

Not answered Marked out of 1.00

What is the output of the following program?

```
public class Program {
    public static void main(String[] args) {
    int total = 0;
    for(int i=0; i<=4;i++)
    {
        for(int j=1;j<=i;j++)
        {
            total=total+i;
        }
    }
    System.out.println(total);
   }
}
```

a. 28

b. 30

c. 15

d. None of the mentioned

e. 10

Questionnaire Given to Test *nested-while* Loop under Nested Looping Category

Question 1 Not answered Marked out of 1.00

What is the output of the following program?

```
public class Program {
    public static void main(String[] args) {
    int total =0,i=0,j=1;
    while(i<=4)
    {
        while(j<=i)
        {
            total=total+i;
            j++;
        }
        i++;
        j=1;
    }
    System.out.println(total);
    }
}
```

a. 30

b. 15

c. None of the mentioned

d. 28

e. 10

Question 2 Not answered Marked out of 1.00

What is the output of the following program?

```
public class Program {
    public static void main(String[] args) {

    int i=1,j;
    while(i<=6)
    {
        j=1;
        while(j<=i)
        {
            System.out.print("*");
            j++;
        }
        System.out.println();
        i++;
    }
    }
}
```

a.
```
*
**
***
****
*****
******
```

b.
```
******
 *****
  ****
   ***
    **
     *
```

c.
```
     *
    **
   ***
  ****
 *****
******
```

d. ******

*****

****

***

**

*

e. None of the mentioned

Question 3

Not answered Marked out of 1.00

What is the output of the following program?

```
public class Program {
    public static void main(String[] args) {

    int i=0,y=0;
    while(i<3)
    {
        while(y<3)
        {
            System.out.println(i + "," + y);
            y++;
        }
        y=0;
        i++;
    }
    }
}
```

a. 0,1

b. 0,0
0,1
0,2
1,0
1,1
1,2
2,0
2,1

c. 0,0
0,1
0,2
1,0
1,1
1,2

d. 0,0
1,1
2,2
1,1
2,2

e. None of the mentioned

Question 4

Not answered Marked out of 1.00

What is the output of the following program?

```
public class Program {
    public static void main(String[] args) {
    int i=1,j=0;
    while(i<3)
    {
        while(j<2)
        {
            System.out.println("six");
            j++;
        }
    j=0;
    i++;
    }
    }
}
```

a. six

six

b. six

six

six

six

six

six

c. six

six

six

six

d. six

six

six

e. None of the mentioned

Question 5

Not answered Marked out of 1.00

What is the output of the following program?

```
public class Program {
    public static void main(String[] args) {
    int i=1,j=0,k=1;
    while(i<=5)
    {
        while(j<=(5-i))
        {
            System.out.print(" ");
            j++;
        }
        j=0;
        while(k<=i)
        {
            System.out.print(i);
            k++;
        }
        k=1;
        i++;
        System.out.println();
    }

 }
}
```

a. 55555
4444
333
22
1

b. 1
22
333
4444
55555

c. 1
22
333
4444
55555

d. None of the mentioned

e. 55555

4444

333

22

1

Appendix E - Aptitude Test Conducted to Select Users

Question 1

Not yet answered

Marked out of 1.00

All exception types are subclasses of the built-in class

Select one:

- a. None of the mention
- b. RuntimeException
- c. Throwable
- d. Error
- e. Exception

Question 2

Not yet answered

Marked out of 1.00

```
1.    class output {
2.        public static void main(String args[])
3.        {
4.             String chars[] = {"a", "b", "c", "a", "c"};
5.             for (int i = 0; i < chars.length; ++i)
6.                 for (int j = i + 1; j < chars.length; ++j)
7.                     if(chars[i].compareTo(chars[j]) == 0)
8.                         System.out.print(chars[j]);
9.        }
10.   }
```

What is the output of this program?

Select one:

- a. ac
- b. ca
- c. ab
- d. bc
- e. cb

Question 3

Not yet answered

Marked out of 1.00

What is the output of this program?

```
import java.util.*;

class Output {
    public static void main(String args[])
    {
        ArrayList obj = new ArrayList();
        obj.add("A");
        obj.add(0, "B");
        System.out.println(obj.size());
    }
}
```

Select one:

- a. None of the mentioned
- b. 2
- c. Any Garbage Value
- d. 0
- e. 1

Question 4

Not yet answered

Marked out of 1.00

Which of the following statements is TRUE

Select one:

- a.
  A class can have multiple constructors
- b.
  The Java compiler always adds a default constructor to a user defined class.
- c.
  Each instantiated object will have its own copy of a class variable.
- d.
  When an object is passed to a method, a copy of each of the object's data members are created and passed to the method.

Question 5

Not yet answered

Marked out of 1.00

```
int num = 35;
switch (num) {
    case 20: num = num + 1;
    case 25: num = num + 2;
    case 30: num = num + 3;
    case 35: num = num + 4;
    case 40: num = num + 5;
```

What is the value **num** after executing the above code?

Select one:

- a. None of them
- b. 38
- c. 44
- d. 39

Question 6

Not yet answered

Marked out of 1.00

What is the string contained in s after following lines of code?

```
StringBuffer s = new StringBuffer("Hello");
s.deleteCharAt(0);
```

Select one:

- a. Hell
- b. llo
- c. ello
- d. null
- e. Hel

Question 7

Not yet answered

Marked out of 1.00

Which of the following is a valid declaration of an object of class Box?

Select one:

- a. obj = new Box();
- b. new Box obj;
- c. Box obj = new Box;
- d. Box obj = new Box();

Question 8

Not yet answered

Marked out of 1.00

Given the following:

```
int[][] items =  {{0, 1, 3, 4},{4, 3, 99, 0, 7 },{3, 2}};
```

Which of the following replaces 99 with 77?

Select one:

a.
```
items[1][2] = 77;
```

b.
```
items[2][1] = 77;
```

c.
```
items[99] = 77;
```

d.
```
items[2][3] = 77;
```

Question 9

Not yet answered

Marked out of 1.00

Which of these access specifiers can be used for a class so that it's members can be accessed by a different class in the different package?

Select one:

a. protected

b. private

c. friendly

d. public

Question 10

Not yet answered

Marked out of 1.00

What is the output of this program?

```
class Test {
      public static void main(String args[])
      {
         int x;
         x = 10;
         {
          int y = x;
          System.out.print(x +" ");
                        y++;
                       x = y;
         }
         System.out.println(x);
      }
  }
```

Select one:

- a. 10 10
- b. 10 11
- c. 11
- d. 10
- e. 11 10

Question 11

Not yet answered

Marked out of 1.00

```
class output {
  public static void main(String args[])
  {
      StringBuffer b1 = new StringBuffer("Hello World");
      b1.insert(6, "Good");
      System.out.println(b1);
  }
}
```

What is the output of this program?

Select one:

- a. Hello GoodWorld
- b. Hello World
- c. Hello Good
- d. Hello World Good

Question 12

Not yet answered

Marked out of 1.00

Select the in correct variable declarations from given below.

Select one or more:

- a. Distance@home
- b. Studentmark
- c. 3rdheight
- d. Kilometers per hour
- e. $name

Question 13

Not yet answered

Marked out of 1.00

Which one of the following options provides the output of this when executed?

```
class StrEqual {
  public static void main(String[] args) {

    String str1 = "Java is portable";

    String str2 = "Java is portable";

    String str3 = new String("Java is portable");

    if (str1 == str2) {
      System.out.println("str1 and str2 equal");
    } else {
      System.out.println("str1 and str2 not equal");
    }
    if (str1 == str3) {
      System.out.println("str1 and str3 equal");
    } else {
      System.out.println("str1 and str3 not equal");
    }

  }}
```

Select one:

a.
str1 and str2 equal
str1 and str3 not equal

b.
str1 and str2 equal
str1 and str3 equal

c.
str1 and str2 not equal
str1 and str3 equal

d.
str1 and str2 equal
str1 and str3 not equal

Question 14

Not yet answered

Marked out of 1.00

Which function is used to perform some action when the object is to be destroyed?

Select one:

a. finalize()

b. none of the mentioned

c. main()

d. new

e. delete()

Question 15

Not yet answered

Marked out of 1.00

Given the following:

```
double[][] arr = { {1.2, 9.0}, {9.2, 0.5, 0.0},{7.3, 7.9,1.2, 3.9} } ;
```

What is the value of arr.length ?

Select one:

- a. 2
- b. 4
- c. 3
- d. 9

Question 16

Not yet answered

Marked out of 1.00

Which exception could be handled by the catch block for above?

```
public class Test{
    public static void main(String args[]){
        try {
            int a = Integer.parseInt("four");
        }
    }
}
```

Select one:

- a. IllegalStateException
- b. ArrayIndexOutOfBoundsException
- c. ClassCastException
- d. TypeMismatchException
- e. Program doesn't run because of a Compilation Error

Question 17

Not yet answered

Marked out of 1.00

Which of the following variable declaration would NOT compile in a java program?

Select one:

- a. int var_1;
- b. int var1;
- c. int VAR;
- d. int 1_var;
- e. int var;

Question 18

Not yet answered

Marked out of 1.00

If a class inheriting an abstract class and does not define all of its function then it will be known as?

Select one:

- a. Static class
- b. None of the mentioned
- c. A simple class
- d. Abstract
- e. Concrete class

Question 19

Not yet answered

Marked out of 1.00

Determine output of the following program code?

```
public class Test{
      public static void main(String args[]){
            int i;
            try {
                  i = calculate();
                  System.out.println(i);
            } catch(Exception e){
                  System.out.println("Error occured");
            }
      }

      static int calculate(){
            return (7/2);
      }
}
```

Select one:

- Program compiles and runs but no output generated
- Compilation Error
- 3.5
- 3
- Runtime Error

Question 20

Not yet answered

Marked out of 1.00

Which of these keywords is used to make a constant in java?

Select one:

- a. abstract and final
- b. struct
- c. implements
- d. static and final
- e. static

Question 21

Not yet answered

Marked out of 1.00

Which of these statement is incorrect?

Select one:

- a. it is possible to create a nested switch statements.
- b. switch statement can only test for equality, whereas if statement can evaluate any type of boolean expression.
- c. switch statement is more efficient than a set of nested ifs
- d. None of the given answers.
- e. two case constants in the same switch can have identical values.

Question 22

Not yet answered

Marked out of 1.00

Which of these is necessary condition for automatic type conversion in Java?

Select one:

- a. The destination type is smaller than source type
- b. The destination type can be larger or smaller than source type
- c. None of the mentioned
- d. All of the mentioned
- e. The destination type is larger than source type

Question 23

Not yet answered

Marked out of 1.00

Which of these statement is incorrect?

Select one:

- a. main() method must be made public
- b. All object of a class are allocated memory for all the attributes defined in the class
- c. All object of a class are allocated memory for the methods defined in the class
- d. If a method is defined public it can be accessed by object of other class
- e. none of the mention

Question 24

Not yet answered

Marked out of 1.00

What is the output of this program?

```
class Test {
     public static void main(String args[]) {
            int x;
            x = 7;
            {
               int y = 3;
               ++x;
               System.out.print(x + " " + y);
            }
            System.out.println(x + " " + y);
     }
}
```

Select one:

- a. 3 7 3
- b. Runtime error
- c. 7 3 7
- d. Compilation error
- e. 7 3 7 3

Question 25

Not yet answered

Marked out of 1.00

Which of these packages contain all the collection classes?

Select one:

- a. java.lang
- b. java.net
- c. java.util
- d. java.awt
- e. java.io

Question 26

Not yet answered

Marked out of 1.00

What will be the output?

```
public class TestException {
   public static void main(String[] args) {
         try {
               int k = 7;
               int a = 0;
               int no = k/a;
         } catch (NullPointerException e1) {
               System.out.print("n");
         } catch (RuntimeException e2) {
               System.out.print("r");
         } finally {
               System.out.print("f");
   }
  }
}
```

Select one:

- fn
- rf
- nf
- f
- nrf

Question 27

Not yet answered

Marked out of 1.00

Platform independent code file created from Source file is understandable by

Select one:

- a. Java compiler
- b. JVM
- c. JDK
- d. SDK
- e. JRE

Question 28

Not yet answered

Marked out of 1.00

What is true about abstract classes

Select one or more:

- a. There are no restrictions in the return type of abstract methods
- b. Abstract classes can inherit from another abstract class
- c. Abstract class can have variables which are static constant (final) variables
- d. Abstract methods can be defined in a normal class
- e. Abstract classes and interfaces are identical

Question 29

Not yet answered

Marked out of 1.00

Which of the following are wrapper classes

Select one or more:

- [ ] a. Object
- [ ] b. String
- [ ] c. Number
- [ ] d. Integer
- [ ] e. Double

Question 30

Not yet answered

Marked out of 1.00

Which one of these is NOT a wrapper class in Java

Select one:

- ○ a. Boolean
- ○ b. double
- ○ c. Integer
- ○ d. Float
- ○ e. Character