Cognitive Complexity Applied to Software Development: An Automated Procedure to Reduce the Comprehension Effort

Dinuka R. Wijendra[1,*] & K. P. Hewagamage[2]

[1]Department of Information Technology, Sri Lanka Institute of Information Technology, SLIIT Malabe Campus, New Kandy Rd, Malabe 10115, Sri Lanka
[2]Department of Information Engineering, University of Colombo School of Computing, UCSC Building Complex, 35 Reid Avenue, Colombo 00700, Sri Lanka
*E-mail: dinuka.w@sliit.lk

Abstract. The cognitive complexity of a software application determines the amount of human effort required to comprehend its internal logic, which results in a subjective measurement. The quantification process of the cognitive complexity as a metric is problematic since the factors representing the computation do not represent the exact human cognition. Therefore, the determination of cognitive complexity requires expansion beyond its quantification. The human comprehension effort related with a software application is associated with each phase of its development process. Correct requirements identification and accurate logical diagram generation prior to code implementation can lead to proper logical identification of software applications. Moreover, human comprehension is essential for software maintenance. Defect identification, correction and handling of code quality issues cannot be maintained without good comprehension. Therefore, cognitive complexity can be effectively applied to demonstrate human understandability inside the respective phases of requirements analysis, design, defect tracking, and code quality optimization. This study involved automation of the above-mentioned phases to reduce the manual human cognitive load and reduce cognitive complexity. It was found that the proposed system could enhance the average accuracy of requirements analysis and class diagram generation by 14.44% and 9.89% average accuracy incrementation through defect tracking and code quality issues compared to manual procedures.

Keywords: *cognitive complexity; cognitive load; comprehension; software development; subjectivity.*

1 Introduction

Numerous studies have been carried out to express software complexity. Software complexity was found to be declared through the human comprehension level [1-3]. Along with that, research works have been conducted to examine the factors that influence the comprehension level of computer programs to express software complexity. Consequently, the concept of cognitive complexity was introduced. The cognitive complexity of a software application defines the

Received August 14th, 2022, Revised October 12th, 2022, Accepted for publication November 17th, 2022.
 ISSN: 2337-5787, DOI: 10.5614/itbj.ict.res.appl.2022.16.3.6

amount of human effort required to comprehend the internal logic of a given software application [1-5]. Since the expression of software complexity elaborates the level of difficulty associated with understandability, it can be claimed to have a direct impact on the human comprehension effort and the cognitive load [6]. Therefore, cognitive complexity can be considered as a direct parameter for evaluating overall software application complexity, as human comprehension and cognitive load are the major determinants behind cognitive complexity.

Human comprehension efforts depend on the individual who performs a task in a given software application. The capability of each individual to handle software applications is different, so that cognitive complexity should be a subjective dimension. However, previous research works were conducted to quantify the cognitive complexity as a form of metric to increase its usability and to make the complexity comparison process easier. Hence, a cognitive complexity metric has been introduced according to quantifiable source code aspects through a set of equations. The architectural aspect of the source code, namely the number of operators, operands, input output parameters, attributes, basic control structures (BCS), method callings [1,2,7-9], spatial capacity [10,11] and consideration of quantifiable object-oriented concepts [12-15] can be observed in these computations. Further, the subjectivity of human comprehension has been expressed through the cognitive weightage concept, which is a numerical value to indicate the comprehension effort related to a corresponding data category [7,12,16]. Each method is based on different quantifiable aspects so that cognitive complexity is denoted by different sets of equations resulting in different quantifications, which are hard to generalize into a single measurement. Nevertheless, the non-standardized nature of cognitive weightages emphasizes the problem of their validity as they do not represent the entire user population. In most contexts, they are merely assumptions or limited to a specific user group that does not represent the understandability of actual users [2,10,17]. Moreover, the limitation of cognitive complexity calculation only based on source code aspects can be stated as another major drawback, as cognitive complexity should be evaluated through personal and software-based aspects as well. In other words, the consideration of personal and software factors cannot be observed in these computations, which makes cognitive complexity measurement unstable.

The performance of these computations is achieved by evaluating the guidelines mentioned in standard complexity metrics frameworks by ensuring their usage in real applications. Accordingly, some computations have been evaluated through Weyuker properties [18,19] and the guidelines in Briands' framework [20], but the problems related with subjectivity, non-standarization and limited factors have remained unchanged. Surprisingly, none of the proposed metrics have been tested against their accuracy levels as a quantitative performance measurement,

which is the major rationale behind the cognitive complexity being proposed for standardization in the near future.

As a solution, we strongly suggest the significance of managing cognitive complexity not only with a system's source code but also with the whole software development process. Accordingly, cognitive complexity is not confined to a set of quantifiable measurements and is represented through the computational background to demonstrate its meta factors. The computational process is implemented through pre and post software development processes. We verified that the comprehension effort associated with a given source code can be reduced by referring to its requirements analysis and design stages. Thereby, for the pre-software development process, automation of the requirements analysis and the design stages was implemented. Furthermore, we analyzed the possibility of applying the cognitive complexity assessment method to the maintenance process, since user understandability is essential for system maintenance [21]. Consequently, automation of defect tracing and code quality optimization inside the maintenance process was implemented as a post-software development process to assist the human cognitive load. Thus, the proposed system is capable of retrieving the requirements using POS tagging, generating class diagrams using the PlantUML library, tracing coding defects that are not identified through FindBugs tracking and optimizing Java code smells. Moreover, the system was proven to reduce the human comprehension effort by mitigating the cognitive load through incremented accuracies, thereby reducing the software application's cognitive complexity and increasing its usability.

The remaining sections of this paper are structured as follows. Section 2 outlines the methodology used for the proposed system and the functionalities. The analysis of the system components is elaborated in Section 3. In Section 4, the results and the discussion are presented. Finally, the conclusion and future works are mentioned in Section 5.

2 Methodology

The proposed system mainly aims to demonstrate the applicability of cognitive complexity assessment through pre and post software development processes. The reduction of the human cognition effort through system components is another goal of the proposed functionalities. Thereby, some drawbacks of the current cognitive complexity metrics, namely the lack of consideration of qualitative factors, the incapability of illustrating the subjectivity and the usage of non-standardized measurements, are expected to be solved. The overall architecture of the system is shown in Figure 1.

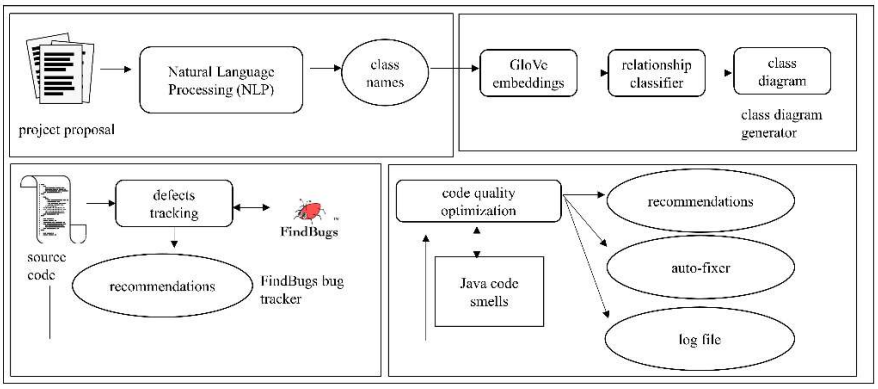

Figure 1 Proposed system architecture.

2.1 System Requirements Automation

A proper requirements analysis of any software implementation would lead to the processing of other phases inside the Software Development Life Cycle (SDLC) more easily than a software application with complex and non-analyzed requirements [22]. Additionally, non-analyzed and error-prone requirements would lead to complex processing, which could create a complex source code. The analysis process of the requirements is basically a manual process, which consumes a substantial amount of human effort. It is therefore essential to examine the significance of analyzing these requirements computationally.

Generally, the requirements are preliminarily documented under the project proposal. The non-existence of a standard format for the project proposal document leads to the problem of defining a proper mechanism of analyzing the project proposal documents computationally and to finalize the requirements specified by the customer. Therefore, the proposal document was converted into a common outline consisting of five sections. The Introduction section outlines a brief explanation of the overall system. The Problem Definition section describes the problem which is expected to be addressed by the system. The Solution section defines the expected characteristics of the software. The expected requirements of the software are listed under the Functionalities section. Finally, the details of the software development process and the team are listed under the Team Profile section. Thus, the system requirements can be gathered by referring to the Functionalities section. However, retrieving the same requirements specified in the Functionalities section would not help the visualization component, as class diagram generation requires the possible class names to be inputted. Therefore, it is aimed at extracting the class names mentioned in the Functionalities section as requirements. Class names are basically identified by

extracting the nouns in the Functionality section due to the high probability of using nouns as the names for the classes [23]. High frequency nouns listed in the requirements were targeted to be accessed as possible class names, which can be used in the diagram generation process. The nouns in the requirements are filtered through Part Of Speech tagging (POS) [24]. Each word mentioned inside the Functionality section is assigned with related tags based on POS. The identified nouns are based on highest frequency, which is passed through a Python script and stored in a text file. The logic of identifying the class names of a given specification with respect to the given frequency level by *classIdentification.py* is given in Algorithm 1.

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

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

2.2 Class Diagram Generation

The analyzed requirements should be converted to different visual representations that demonstrate the internal logic of the software to be implemented. This process is usually a manual process, as the relevant source codes will be implemented by referring to those representations [22]. The conversion of the requirements to respective logical diagrams as a manual process can include failures and error-prone activities. This could be one of the reasons why the generated source codes could turn out more complex and error-prone than expected. Meanwhile, the existing automation tools for logical diagram generation cannot be used for this purpose, as the generated source code has to

be inputted for diagram generation [25]. By considering this situation, the proposed system was outfitted with a component to generate the class diagrams prior to source code generation.

Table 1 Types of relationships based on cosine similarities.

Cosine similarity range	Type of the relationship	Notation
80<= similarity <100 Similarity = 100 implies the same word is compared	Inheritance	"<|--"
70<= similarity <80	Composition	"*--"
60<= similarity <70	Aggregation	"o--"
25<= similarity <60	Association	"--"
Similarity < 25	No relation	Not applicable

The class names recognized by the requirements analyzer is the major input for this task. The relationships and the similarity levels of each class name have to be studied, as the class diagram generation requires the logical association with each class. Global Vectors for Word Representation (GloVe) [26] was used to compute the relationships between each class name. The relationships among each of two class names are then evaluated through cosine similarity [27]. The Torchtext[i] library supported by the PyTorch machine learning framework has been used to achieve GloVe vector representations and cosine similarity [27]. Then, the necessity of deriving the types of the relationships was analyzed. Consequently, we defined five types of relationship categories based on the range of cosine similarities. Accordingly, the component is capable of generating a class diagram by referring to the types of relationships. The class diagram generation was performed through the PlantUML library [25]. Each relationship type is uniquely represented using different notations. Table 1 specifies the types of relationships and the notations used to denote different types of relationships.

The class names with high similarity levels imply more common behavior, which can be considered a parent-child relationship [28]. Hence, the relationships with the 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 the inheritance [29]. 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 smaller than the similarity range used for composition [30]. An association relation maintains a weaker connectivity among the classes, so the similarity range has to be smaller than the inheritance, composition and aggregation[31]. Class names smaller than 25 similarity level were considered unnecessary relationships, which do not imply a strong connection among them.

Then, PlantUML statements were written with class names, their relationship types along with the corresponding notations given as input. Then, these statements are compiled to generate the class diagram. The coded diagrams are then converted into a Portable Network Graphic (PNG) image using GraphViz[ii] and stored inside the project directory for reference. The internal logic of generating the class diagram by *generateClassDiagram()* is shown in Algorithm 2. The proposed system was evaluated using different types of system specifications and Figure 2 demonstrates the class diagrams generated for some system specifications.

Algorithm 2: Function *generateClassDiagram()* to generate the class diagram

Input: The list of class names obtained through requirements automation
Output: Class diagram
1: Let *Relationship* be a separate class created to handle pairs of class names with their similarity values
2: Let *reList* is the list created from *Relationship* class
3: call function *assignRelationshipType(reList)* //refer Table 1
4: Let *classA* and *classB* represent the class names in r
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 1
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

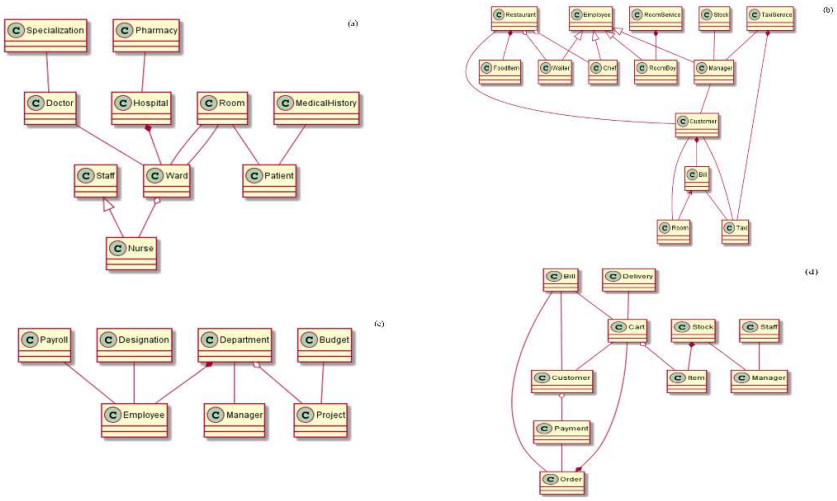

Figure 2 Generated class diagrams for (a) a hospital management system, (b) a hotel management system, (c) a human resources system, (d) an online shopping system.

2.3 Defect Tracking Automation

A defect or a bug is an unexpected event that occurs in a source code and results in a non-functioning state of the system [32]. Therefore, the occurrence of a defect should be resolved to resume the functioning of the source code to obtain the expected outcomes. The usage of compilers and bugs trackers assists in identifying possible coding defects, so that the user can identify and correct them to produce a smooth execution of the given source code. However, available bug trackers do not have the ability to recognize all types of coding defects.

A non-recognized bug would create an unexpected scenario of source code execution, resulting in the user having to manually identify the cause of the unexpected outcome and taking necessary actions to rectify it. This process consumes a considerable amount of comprehension effort. Thereby, a mechanism must be implemented to identify possible defects that existing bug trackers could not identify to reduce the cognitive complexity associated with the source code. The proposed system is implemented with Apache NetBeans Integrated Development Environment (IDE)[iii] 12.5 with Java Maven. The FindBugs bugs tracker is applicable for NetBeans IDE, so that it was installed as a plugin, which is supported to track coding defects [33]. The proposed defect tracking component was applied to detect bugs that cannot be handled by the FindBugs tracker. The types of bugs exposed by the proposed component are shown in Table 2.

Table 2 Types of bugs (code defects) detected by the system.

Identified bug	Bug 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	Incompatible values are stored 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 state error	.next(), .set() and .remove() methods are called in an 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	A null parameter is passed as a parameter in any method
Illegal thread states	Attempt to start or 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 a 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

The Java source code is inputted for the component and stored inside an array list and checks the appearance of any defects listed in Table 2 by assessing the source code content. Detection of a bug is indicated by its line number to be able to locate the bug without difficulty.

Moreover, the recommendation procedure consists of displaying the identified bug category and suggestions for correcting the bug. Because of this, the user has less effort in finding, locating and comprehending a particular defect. Therefore, the suggested component automates the precise procedure for manual bug detection and correction, which reduces the cognitive complexity of the inputted source code.

2.4 Automation of Code Quality Optimization

Software applications should be maintained to ensure their quality, as they are not handled by the same set of users in its duration of usage [34]. Quality software can be handled easily by different users with less human effort, resulting in low cognitive complexity. The proposed system was implemented to recognize code quality issues aligned with Java code smells [35].

The direct verification of code quality issues without having to upload the source code into existing external quality analyzers such as SonarQube and SonarLint [36] is the main advantage of the proposed component.

The feature of displaying a specific code smell, its location, recommendations to fix the issues were implemented within the quality optimization functionality. Java source code is considered as the input and Java reserved key words, identifiers, operators, numbers, classes, methods, variables and escape characters inside the inputted source code are identified as code elements. Then, their appearance and the structure are verified against the standard code smells. The types of quality issues handled by the component is listed in Table 3.

Once a code smell is recognized, the type of code smell and the recommendation of the procedure to correct it is displayed. A separate log file with the class name is created inside the current project directory so that the user can view the changes made under quality optimization.

More significantly, the feature of auto fixing code smells for unused import detection and redundant modifiers in interfaces was also introduced in this functionality to minimize the manual effort associated with correcting quality issues by referring to recommendations.

Table 3 Types of quality issues (code smells) detected by the system.

Code smell	Description
Unused import detection	Import statements that have not been used inside the source code. Can automatically be fixed by the component.
Redundant modifier usage in interface detection	An interface access modifier is used again for declarations inside the interface. Can automatically be fixed by the component.
Invalid usage of generics in constructor detection	Generics are declared in constructors when the declaration also is expressed as generic. To address this, the diamond operator can be used. Can automatically be fixed 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

3 Analysis of System Components

We have identified that the complexity related to understanding a source code logic can be reduced by referring to its requirements analysis and design phases. It is evident that the time taken to perform a task inside a given source code is reflected by its comprehension effort. Accordingly, we examined the time taken to perform a given set of tasks by referring only to the original source code and by referring to its requirements analysis and design phases without its source code through a set of users. The selection of users was conducted, followed by execution of the experiment performed in [37]. A computer-based questionnaire was conducted among a group of computing-related final-year BSc university students to examine their coding expertise. Students who scored more than 60% were selected as the target user group. Those students were classified into another four groups based on score levels of 60%-70%, 70%-80%, 80%-90% and 90%-100% respectively. Source codes, modeled class diagrams, and specified user requirements of ten different software applications were considered as input for the analysis. The duration for task completion was obtained through a Moodle environment.

Figure 3 represents the analytical outcomes obtained. The average duration spent for the reverse process decreased from 110.12 minutes to 86.28 minutes, equal to 13.83% reduction. This implies that the comprehension effort was reduced through backward navigation from the source code creation process. Along with

this, automation of requirements analysis and class diagram visualization was included to reduce the manual comprehension effort, thus, lower cognitive complexity was expected.

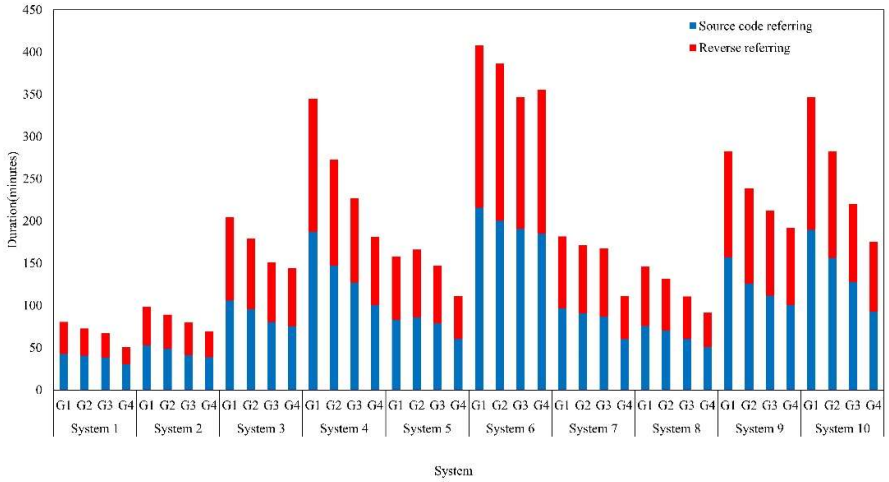

Figure 3 Average duration variation for source code and reverse processing.

Since cognitive complexity as a metric is not standardized, it is impossible to denote the comprehension effort utilized by system components as a metric and to compare it with manual processing. The duration reduction of the system components cannot be granted as a direct parameter of cognitive complexity reduction, as any automation reduces the duration compared to manual processing. Hence, along with the duration, the accuracy levels have to be computed and verified against manual processing. Consequently, a set of equations was formed to compute the accuracy levels in each component. As such, the requirements analysis' accuracy ($accuracy_{RA}$) is denoted by Eq. (1). The number of specifications is denoted by k. Let N_i be the correct number of class names and n_i be the identified number of class names in the i^{th} specification.

$$accuracy_{(RA)} = \sum_{i=1}^{k}(\frac{n_i}{N_i}) * 100 \quad (1)$$

Similarly, the accuracy of the visualization of the class diagram ($accuracy_{VC}$) was computed as follows:

$$accuracy_{(VC)} = \sum_{i=1}^{k}(\frac{m_i}{n_i C_2}) * 100 \quad (2)$$

Note that the correct number of class name pairs was identified through ${}^{ni}C_2$. The correct number of relationship types identified through each pair of classes in the

i^{th} specification is denoted by m_i. The Eq. (2) can be further simplified and used as shown in Eq. (3):

$$accuracy(vc) = \sum_{i=1}^{k}(\frac{m_i*(n_i-2)!*2!}{n_i!}) * 100 \quad (3)$$

The efficiency of the defect tracing component was also retrieved by maintaining the accuracy level. As such, the accuracy of defect tracing ($accuracy_{DT}$) was calculated with Equation (4). The number of software systems tested is denoted by k. Let DT_i be the actual coding defects inside the i^{th} source code and dt_i be the number of defects that have been solved correctly.

$$accuracy_{(DT)} = \sum_{i=1}^{k}(\frac{dt_i}{DT_i}) * 100 \quad (4)$$

Similarly, the accuracy of code quality optimization ($accuracy_{CQ}$) was computed with Equation (5), where CQ_i is the actual number of code smells inside the i^{th} source code and cq_i is the number of correctly solved code smells.

$$accuracy_{(CQ)} = \sum_{i=1}^{k}(\frac{cq_i}{CQ_i}) * 100 \quad (5)$$

In order to obtain the accuracies of both manual and automated processing, thirty different software systems were considered. The written specifications and implemented source codes for each system were inputted for the accuracy computations of the pre and post software development processes, respectively. The same four user groups were used as target user group to stabilize the comprehension effort. Firstly, thirty specifications were given to each user group, requesting them to analyze the requirements as class names and model the class diagrams manually. The same procedure was followed through the proposed system by inputting the system specifications and calculate the accuracies using Eqs. (1), (2) and (3). Thus, the procedure of comparing the accuracies generated for the pre-software development process was accomplished. Then, the source codes generated for the thirty systems were given to the test subjects, requesting them to manually analyze and fix the coding defects and quality smells. Simultaneously, the same process was done automatically through the proposed system, obtaining the output codes to analyze the accuracies with Eqs. (4) and (5), which can be used to represent the performance of the post-software development process.

4 Results and Discussion

4.1 Pre-Software Development Process

The accuracy levels obtained for requirements analysis by both manual processing and the proposed system can be seen in Figure 4. It is noteworthy that

the average accuracy levels of the four different user groups for each software specification were computed and shown.

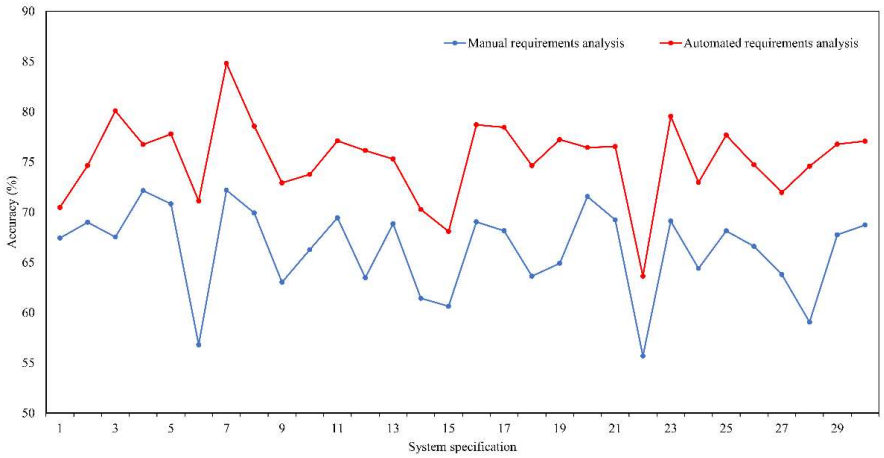

Figure 4 Average accuracy for the requirements analysis phase.

It can be clearly observed that the average accuracy levels of the requirements analysis phase were comparatively higher than with manual processing. This indicates the reduction of the cognitive load that each individual should have through the analysis process, which in turns reduces the comprehension effort. On average, the manual processing accuracy was 66.29%, while 75.29% accuracy was achieved by the system component. The accuracy increment of the system component was computed as 13.58% with respect to manual processing. Thus, we can state that the automated requirements analysis helped to reduce the cognitive complexity while achieving accurate outcomes.

The average accuracy levels obtained through class diagram visualization can be seen in Figure 5. Manual processing for the class diagram visualization attained 66.39% accuracy, while 76.55% accuracy was obtained by the system component. Hence, an increase of 15.3% accuracy was achieved by the proposed system compared to manual processing. Along with both components, it can be stated that the average accuracy increase of the pre-software development phases provided by the software was 14.44%. Based on these accuracy increases, it can be stated that the usage of the proposed system components can optimize the requirements analysis and class diagram visualization in pre-software development. This can be considered as an essential achievement in generating accurate source code as accurate source code is an outcome of properly executed requirements analysis and design phases. Moreover, an accurate source code should decrease the cognitive complexity affected by it, as the amount of human

comprehension effort to understand its logic should be comparatively low with complex source codes. Hence, along with the automation procedure and accuracy increase, the amount of human comprehension effort and cognitive load involved can be stated as having decreased by using the proposed components to ensure lower cognitive complexity.

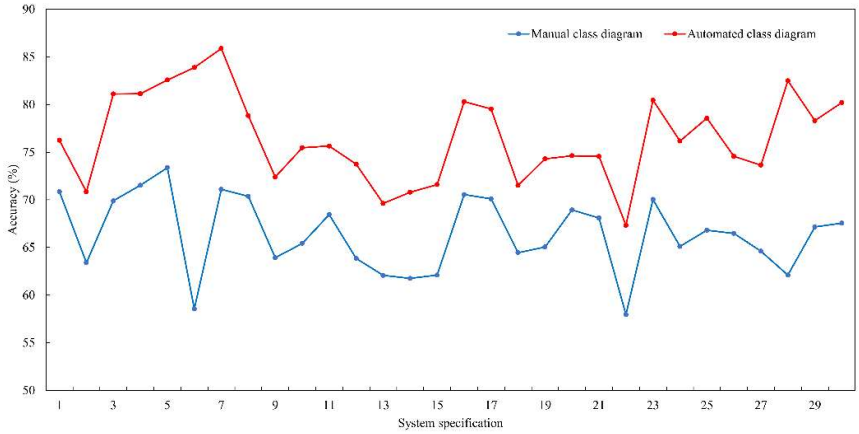

Figure 5 Average accuracy for class diagram generation phase.

4.2 Post-Software Development Process

The analytical outcomes for the thirty source code implementations with average accuracies of the four different user groups for defect tracing and quality optimization are shown in Figures 6 and 7 respectively.

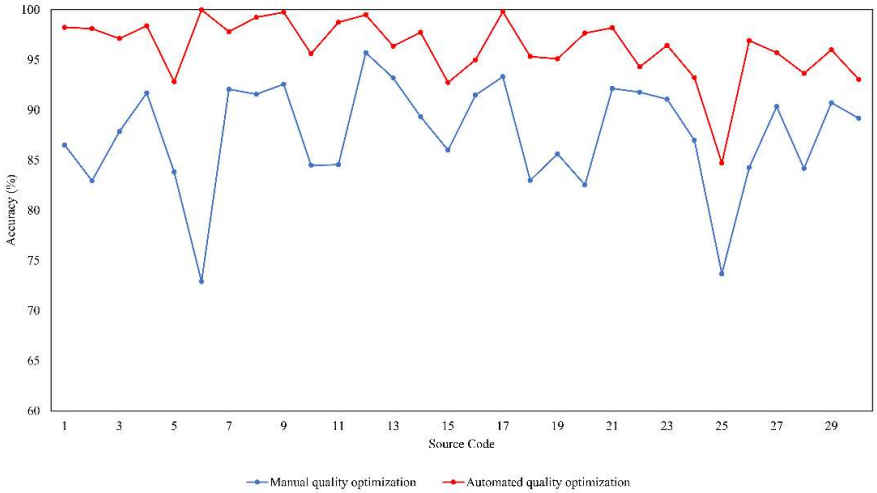

Figure 6 Average accuracy of defect tracking.

Based on both figures, it can be clearly observed to have a higher accuracy through the automated procedures than manual processing. Figure 6 shows an average accuracy of 84.54% for manual defect tracing and an accuracy of 92.83% for automated defect tracing. Further, the average accuracy increase of automated defect racing over manual processing was 9.8%.

In Figure 7, 87.52% and 96.25% accuracy increases can be observed for manual and automated code quality optimization, respectively. Therefore, the accuracy increase of code quality optimization can be stated as 9.97% over manual processing. Consequently, the average accuracy increase obtained by the system in the post-software development phases can be stated as 9.89%.

An accurate outcome through automated functionalities can reduce the comprehension effort more compared to erroneous output since a considerable human effort has to be utilized for handling such defects. Hence, the verification of the system components to reduce the cognitive complexity in the post-software development can be emphasized.

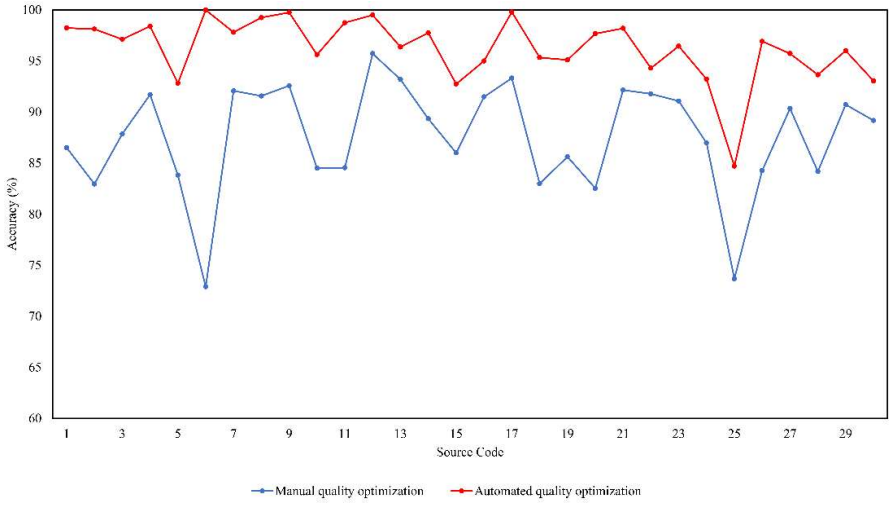

Figure 7 Average accuracy for code quality optimization.

5 Conclusion and Future Works

The main purpose of this research was to express the cognitive complexity of software through the phases in the software development process. Most related works on cognitive complexity were based on limited and quantifiable source code aspects regardless of subjectivity. We identified the significance of cognitive complexity without confining it to a quantitative measurement.

Consequently, we have introduced a system which includes the automated features of the requirement analysis, class diagram designing, bug identification, and code quality optimization.

To act as a guidance for the human comprehension effort and to reduce the cognitive complexity of software are the major objectives behind the proposed system. The system was tested for thirty different software specifications by four different user groups with a hundred users in each user group. The cognitive complexity reduction was verified using the percentages of accuracy for each component of the system. On average, it can be stated that the proposed system was capable of achieving 14.44% and 9.89% improved accuracy levels for pre-software and post-software development processes, respectively. Therefore, we can state that the suggested system components can be used to increase human cognition by mitigating the cognitive complexity associated with a given software application. Nevertheless, the number of coding defects and quality smells identified by the proposed system is limited. Moreover, the system is capable of handling source code written only in Java. Therefore, in future work, we expect to expand the number of defects identified by the system, and to introduce a mechanism to automate refactoring techniques with a recommendation system to enhance code quality optimization. Finally, we plan to evaluate the system through current complexity metrics to accentuate the complexity reduction.

References

[1] Campbell, G.A., *Cognitive Complexity: An Overview and Evaluation*, 2018 International Conference on Technical Debt, pp. 57-58, May 2018. DOI: 10.1145/3194164.3194186.

[2] Misra, S., *A Complexity Measure Based on Cognitive Weights*, International Journal of Theoretical and Applied Computer Sciences, **1**(1), pp. 1-10, 2006.

[3] Barón, M.M., Wyrich, M. & Wagner, S., *An Empirical Validation of Cognitive Complexity as a Measure of Source Code Understandability*, 14th ACM IEEE Int. Symp. Empir. Softw. Eng. Meas. ESEM, pp. 1-12, Oct. 2020. DOI: 10.1145/3382494.3410636.

[4] Winter, M., Pryss, R., Probst, T., Bass, J. & Reichert, M., *Measuring the Cognitive Complexity in the Comprehension of Modular Process Models*, IEEE Trans. Cogn. Dev. Syst., pp. 1-18, 2020. DOI: 10.1109/TCDS.2020.3032730.

[5] Xia, X., Bao, L., Lo, D., Xing, Z., Hassan, A.E. & Li, S., *Measuring Program Comprehension: A Large-Scale Field Study with Professionals*, IEEE Trans. Softw. Eng., **44**(10), pp. 951-976, Oct. 2018. DOI: 10.1109/TSE.2017.2734091.

[6] Saborido, R., Ferrer, J., Chicano, F. & Alba, E., *Automatizing Software Cognitive Complexity Reduction*, IEEE Access, **10**, pp. 11642-11656, 2022. DOI: 10.1109/ACCESS.2022.3144743.

[7] Campbell, G.A., *A New Way of Measuring Understandability*, SonarSource S A, 2016/2021.

[8] Kushwaha, D.S. & Misra, A.K., *Improved Cognitive Information Complexity Measure: A Metric that Establishes Program Comprehension Effort*, ACM SIGSOFT Software Engineering Notes, **31**(5), pp. 1-7, 2006. DOI: 10.1145/1163514.1163533

[9] Wang, Y., *On the Cognitive Complexity of Software and Its Quantification and Formal Measurement*, Int. J. Softw. Sci. Comput. Intell., **1**(2), pp. 31-53, Apr. 2009. DOI: 10.4018/jssci.2009040103.

[10] Chhabra, J.K., *Code Cognitive Complexity: A New Measure*, The World Congress on Engineering, July 2011.

[11] Gold, N.E., Mohan, A.M. & Layzell, P.J., *Spatial Complexity Metrics: an Investigation of Utility*, IEEE Trans. Softw. Eng., **31**(3), pp. 203-212, Mar. 2005. DOI: 10.1109/TSE.2005.39.

[12] Chhillar, U. & Bhasin, S., *A New Weighted Composite Complexity Measure for Object-Oriented Systems*, International Journal of Information and Communication Technology Research, **1**(3), pp. 101-108, July 2011.

[13] Misra, S., Adewumi, A., Fernandez-Sanz, L. & Damasevicius, R., *A Suite of Object Oriented Cognitive Complexity Metrics*, IEEE Access, **6**, pp. 8782-8796, 2018, DOI: 10.1109/ACCESS.2018.2791344.

[14] Gupta, V. & Chhabra, J. K., *Object-Oriented Cognitive-Spatial Complexity Measure*s, the World Academy of Science, Engineering and Technology, **3**(3), pp. 972-979, 2009, DOI: 10.5281/zenodo.1072347.

[15] Misra, S. & Akman, I., *Applicability of Weyuker's Properties on OO Metrics: Some Misunderstandings*, Comput. Sci. Inf. Syst., **5**(1), pp. 17-23, 2008. DOI: 10.2298/CSIS0801017M.

[16] Shao, J. & Wang, Y., *A New Measure of Software Complexity Based on Cognitive Weights,* Can. J. Electr. Comput. Eng., **28**(2), pp. 69-74, Apr. 2003. DOI: 10.1109/CJECE.2003.1532511.

[17] Misra, S., Koyuncu, M., Crasso, M., Mateos, C. & Zunino, A., *A Suite of Cognitive Complexity Metrics, Computational Science and Its Applications*, **7336**, pp. 234-247, 2012. DOI: 10.1007/978-3-642-31128-4_17.

[18] Misra, A.K., *Evaluating Cognitive Complexity Measure with Weyuker Properties*, Third IEEE International Conference on Cognitive Informatics, pp. 103-108, 2004, DOI: 10.1109/COGINF.2004.1327464.

[19] Kushwaha, D.S. & Misra, A.K., *Robustness Analysis of Cognitive Information Complexity Measure Using Weyuker Properties*, ACM SIGSOFT Softw. Eng. Notes, **31**(1), pp. 1-6, Jan. 2006, DOI: 10.1145/1108768.1108775.

[20] Briand, L. C. & Morasca, S., *Property Based Software Engineering Measurement*, IEEE Trans. Softw. Eng., **22**(1), pp. 68-86, Jan 1996, DOI: 10.1109/32.481535.

[21] Scalabrino, S., Bavota, G., Vendome, Linares-Vasquez, M., Poshyvanyk, D. & Coliveto, R., *Automatically Assessing Code Understandability*, IEEE Trans. Softw. Eng., **47**(3), pp. 595-613, Mar. 2021, DOI: 10.1109/TSE.2019.2901468.

[22] Mishra, A. & Dubey, D., *A Comparative Study Of Different Software Development Life Cycle Models in Different Scenarios*, Int. J. Adv. Res. Comput. Sci. Manag. Stud., **1**(5), pp. 64-69, Oct. 2013,

[23] Rquez, L.M., Padro, L. & Rodriguez, H., *A Machine Learning Approach to POS Tagging*, Machine Learning, **39**, pp. 59-91, 2000, DOI: 10.1023/A:1007673816718

[24] Marquez, L., Rodriguez, H., Carmona, J. & Montolio, J*., Improving Pos Tagging Using Machine-Learning* Techniques, pp. 52-63. https://aclanthology.org/W99-0608.pdf, (8 March 2020)

[25] Deeptimahanti, D.K. & Babar, M.A., *An Automated Tool for Generating UML Models from Natural Language Requirements*, IEEE/ACM International Conference on Automated Software Engineering, pp. 680–682, Nov. 2009. DOI: 10.1109/ASE.2009.48.

[26] Pennington, J., Socher, R. & Manning, C., *Glove: Global Vectors for Word Representation*, Conference on Empirical Methods in Natural Language Processing (EMNLP), pp. 1532–1543, 2014. DOI: 10.3115/v1/D14-1162.

[27] Li, B. & Han, L., *Distance Weighted Cosine Similarity Measure for Text Classification*, Intelligent Data Engineering and Automated Learning – IDEAL, pp. 611–618, 2013. DOI: 10.1007/978-3-642-41278-3_74.

[28] Egenhofer, M.J. & Frank, A.U., *Object-Oriented Modeling in CIS: Inheritance and Propagation,* pp. 588-598, 2008.

[29] Dey, D., Storey, V.C. & Barron, T.M*., Improving Database Design through the Analysis of Relationships*, ACM Trans. Database Syst., 24(4), pp. 453–486, Dec. 1999. DOI: 10.1145/331983.331984.

[30] Renguo, X., Dillon, T. S., Rahayu, W., Chang, E. & Gorla, N., *An Indexing Structure For Aggregation Relationship In OODB*, Database and Expert Systems Applications, **1873**, pp. 21–30, 2000, DOI: 10.1007/3-540-44469-6_3

[31] Han, J. & Fu, Y., *Discovery of Multiple-Level Association Rules from Large Databases,* 21st VLDB Conference, 1995.

[32] Rutar, N., Almazan, C.B. & Foster, J.S., *A Comparison of Bug Finding Tools for Java*, 15th International Symposium on Software Reliability Engineering, pp. 245-256, 2004. DOI: 10.1109/ISSRE.2004.1.

[33] Hovemeyer, D. & Pugh, W., *Finding Bugs is Easy*, Static Anal., 2004.

[34] Aggarwal, K.K., Singh, Y. & Chhabra, J.K., *An Integrated Measure of Software Maintainability*, Annual Reliability and Maintainability

Symposium. 2002 Proceedings (Cat. No.02CH37318), pp. 235–241, 2002. DOI: 10.1109/RAMS.2002.981648.

[35] Java Code Smell | Java Code Analyzer, https://rules.sonarsource.com/java/type/Code%20Smell/ (Dec. 02, 2020).

[36] García-Muñoz, J., García-Valls, M. & Escribano-Barreno, J., *Improved Metrics Handling in SonarQube for Software Quality Monitoring*, Advances in Intelligent Systems and Computing book series (AISC), **474**, pp. 463-470, 2016. DOI: 10.1007/978-3-319-40162-1_50.

[37] Aloysius, A. & Arockiam, L., *Coupling Complexity Metric: A Cognitive Approach*, Int. J. Inf. Technol. Comput. Sci., **4**(9), pp. 29-35, Aug. 2012. DOI: 10.5815/ijitcs.2012.09.04.

---

[i] https://pytorch.org/text/stable/index.html, December 2021

[ii] https://graphviz.org/, October 2021

[iii] https://netbeans.apache.org/, January 2020