Cognitive Load Analyzer: A Support Tool for Cognitive-Driven Development

Jherson Haryson A. Pereira
Federal University of Pará
Belém, PA, Brazil
harysonjherson@gmail.com

Alberto Luiz Oliveira Tavares de Souza
Zup Innovation
São Paulo, SP, Brazil
alberto.tavares@zup.com.br

Victor Hugo Santiago C. Pinto
Federal University of Pará
Belém, PA, Brazil
victor.santiago@ufpa.br

ABSTRACT

Software modularity refers to the decomposition of complex software to be manageable for the purpose of implementation and maintenance. Most methodologies and practices adopted in the industry follow this principle, often with the benefit of flexibility and variety in use. This is also a recognition that human work can be improved by focus on a limited set of data. However, code more complex than needed continues being produced resulting in cognitive overload for developers. Cognitive-Driven Development (CDD) is an inspiration from the Cognitive Load Theory for software development with the goal of reducing the split-attention effect and the problem space through a feasible cognitive complexity constraint. Implementation units can be kept under this limit even with the continuous expansion of software scale. Experimental studies were carried out and their results suggested that CDD is a promising method when guiding the development focusing on understanding and achieving high-quality code with respect to the quality metrics. In this paper, we present a tool called "Cognitive Load Analyzer" to support the CDD, a plugin for IntelliJ IDEA and Java language. This tool can be useful to support the adoption of the CDD aiming to overcome the to slice the code and help developers to reduce code complexity.

CCS CONCEPTS

• **Software and its engineering → Software evolution**.

KEYWORDS

Cognitive load analyzer, Cognitive-driven development, Software complexity

ACM Reference Format:
Jherson Haryson A. Pereira, Alberto Luiz Oliveira Tavares de Souza, and Victor Hugo Santiago C. Pinto. 2021. Cognitive Load Analyzer: A Support Tool for Cognitive-Driven Development. In *Brazilian Symposium on Software Engineering (SBES '21), September 27-October 1, 2021, Joinville, Brazil.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3474624.3476011

SBES '21, September 27-October 1, 2021, Joinville, Brazil

ACM ISBN 978-1-4503-9061-3/21/09…$15.00
https://doi.org/10.1145/3474624.3476011

1 INTRODUCTION

Software modularity emphasizes separating the functionality of a program into independent modules, such that each contains everything necessary to execute only one aspect of the desired feature. The main idea behind this process is to maximize the developer's productivity when creating maximum software value for minimum development cost [20]. Therefore, the developer productivity can be amplified when the software complexity is in some way managed [25], [26], [8].

Continuous changes and the expansion of software complexity were highlighted in the Lehman's laws of software evolution [9–11]. The system must be continually adapted or it becomes progressively less satisfactory. However, when a system evolves, its complexity increases unless work is done to maintain or reduce it. According to Dijkstra [4], the success of what we think is closely related to reducing the amount of reasoning needed. Based on this principle, several strategies such as SOLID Design Principles [12], Clean Architecture [13] and Hexagonal Architecture [3] are usually adopted in the industry to make the software design more flexible and maintainable. Domain-driven design (DDD) practices [6] suggested that the language of the software code should be aligned with the business domain. Although these proposals can contribute to a high-quality code and support domain understanding, a serious problem continues affecting software companies: the growth of software complexity over time and the lack of a clear strategy to distribute this complexity considering the human factor. This problem could be a possible reason for the development of more complex code than needed, resulting in cognitive overload for developers.

Cognitive load refers to the amount of information that working memory resources can hold at one time. Cognitive Load Theory (CLT) [22],[2],[21] is generally discussed as regarding the instructional design and learning. Sweller [22] highlighted that any material is intrinsically difficult to understand and this depends on the number of required elements to be simultaneously processed in our working memory. Therefore, problems that require a large number of items to be stored in short-term memory contribute for an excessive cognitive load. Miller's work [14] has suggested that humans are generally able to hold only seven plus or minus two units of information in short-term memory.

Cognitive Complexity is a measure of how difficult a unit of code is to intuitively understand. Souza and Pinto [19] proposed a method called Cognitive-Driven Development (CDD), an inspiration from studies of cognitive psychology for software development. The main goal of CDD is to reduce cognitive overload for developers facing the challenge of dealing with the exponential growth of software

complexity over time. The definition of a cognitive complexity limit for the code to be produced is the main CDD principle. With this in mind, even with software complexity expansion, its units can be kept without exceeding this value. This constraint can be defined using different perspectives, such as the project's nature, team experience level and basic control structures.

The purpose of this paper is to introduce a tool called Cognitive Load Analyzer, a support tool for CDD. It should be noticed that the CDD has already been used by some Brazilian software developers and it does not require a specific tool to follow its principles. However, the calculation process of complexity points can be a tedious and error-prone task, especially when there are many lines of code. In addition, developers need to constantly check whether the cognitive complexity threshold has been reached. Currently, both activities are performed in a manual way.

We believe that CDD can be adopted by more software developers if a support tool were made available to automate such processes. Cognitive Load Analyzer is a plugin for IntelliJ IDEA[1] that enables the definition of the code elements to be considered in the cognitive complexity constraint. Notifications about the status of this limit can be shown and this can be useful to guide the development focusing on understanding. In addition, an analysis report screen is provided to show a more detailed view of the current complexity of the code in comparison to the cognitive complexity defined for each implementation unit in the project.

The remainder of the paper is organized as follows: Section 2 discusses related work; Section 3 presents the CDD principles and how a feasible cognitive complexity constraint can be defined; Section 4 describes the Cognitive Load Analyzer, including its architecture, main features and how it can help developers to improve their code reducing the cognitive load. Finally, Section 5 summarizes the conclusions and sets out future perspectives.

2 RELATED WORK

In practice, an experimental study [16] was carried out considering refactoring scenarios of know projects by the Java developer community. The purpose of this work was to evaluate refactorings guided by CDD, specifically following a cognitive complexity constraint against conventional practices. For this, 18 experienced developers attended the experiment and part of them followed the CDD. Different software metrics were employed through static analysis, such as: CBO (Coupling between objects), WMC (Weight Method Class), RFC (Response for a Class), LCOM (Lack of Cohesion of Methods) and LOC (Lines of Code). The result suggests that the CDD can guide the restructuring process and contributes to achieve a coherent and balanced separation of concerns. Meanwhile, the developers had to calculate the complexity points in their code, as well as having to manage if they reached the complexity limit set in a manual way. Cognitive Load Analyzer can be employed since the early stages of software development and experimental studies can be carried out for verifying the effects of a cognitive complexity limit threshold in the medium and long term.

Campbell [1] describes a new metric that breaks from the use of mathematical models to evaluate code in order to remedy Cyclomatic Complexity's shortcomings and produce a measurement that more accurately reflects the relative difficulty of understanding and therefore of maintaining methods, classes, and applications. The cognitive complexity described by Campbell also uses the strategy of deciding what should be taken into account when computing weights based on human judgment. Nonetheless, unlike Campbell's Cognitive Complexity, the CDD is not just a metric, it guides the development and restricts the developers, not allowing them to exceed the limit set by the team and can indicate the need to refactor the code.

3 COGNITIVE-DRIVEN DEVELOPMENT

Cognitive load relates to the amount of information that working memory can hold at one time. Cognitive Load Theory (CLT) was developed by John Sweller [21] [22] to the learning materials, so that they present information at a pace and at a level of complexity that the learner can fully understand. Based on this theory, all materials are intrinsically complex, therefore, it is important when producing materials to reduce the problem space and the split-attention effect because if we have multiple sources of information, our attention is divided between them.

According to Sweller, since human working memory has a limited capacity, instructional methods should avoid overloading it with additional activities that do not directly contribute to learning. Experimental studies conducted by Miller [14] have suggested that humans are generally able to hold only seven plus or minus two units of information in short-term memory.

Based on such studies from cognitive psychology and object-oriented cognitive complexity metrics field [24][18][23][15], Souza and Pinto [19] have proposed the Cognitive-Driven Development. This method recognizes the human limitation as a way to guide software development, prioritizing the understanding and quality metrics. Most research involving human cognition in software engineering focuses on evaluating programs and learning rather than on understanding how software development could be guided by this perspective [5].

The main idea behind the CDD is not to provide new metrics based on human judgment for the code, the key CDD principle is that a way to deal with the expansion of software scale and its complexity is defining a feasible cognitive complexity limit. This constraint for information units can be applied for software context once the source code has also an intrinsic load. The implementation units can be kept under this limit even with the exponential growth of the software complexity. It should be noticed that this can be natural in software evolution but the human capacity does not follow the same proportion.

A cognitive complexity constraint can be defined by the agreement of the development team members and evaluated later considering the project's nature and level of team expertise. The concept of "*Intrinsic Complexity Points*" (ICPs) was introduced as the elements to be considered in the definition of this limit [19]. According to authors, this constraint must include code branches (*if-else*, *loops*, *when*, *switch/case*, *do-while*, *try-catch-finally* and etc.), *functions as an argument*, *conditionals*, *contextual coupling* - coupling with specific project classes and *inheritance of abstract or concrete class (extends)*. However, developers can include other elements, such as *SQL instructions* and *Annotations*. Specific features of programming

[1] https://www.jetbrains.com/pt-br/idea/

languages and frameworks/libraries are not considered, although they are often not trivial, it is understood that such features are part of common knowledge and under the domain of the developers.

Figure 1 presents a piece of code from a Java class called *GenerateHistoryController*. This class was implemented by one of the authors for a project called *complexity-tracker*[2] that can provide indicatives of how the complexity increases during the software evolution process. The ICPs are calculated as follows: at the top of the figure is shown the number 6, the total of ICPs; 4 points are related to *contextual coupling* (lines 17, 23, 26 and 38) and the remaining points refer to *functions as an argument* (lines 29 and 42).

Currently, this analysis process for identifying ICPs is carried out manually. This is important to verify if the constraint already defined for a project was satisfied for the implementation units. It should be noticed that the main goal for the developers continues the same: understand problems, programming new features and provide a functional solution. Therefore, the CDD can be considered a complementary strategy to the practices already adopted. The main difference when CDD is adopted in software development is that the constraint of cognitive complexity can guide the software development aiming to reduce the cognitive overload for future efforts.

```
...
12 @Controller
13public class GenerateHistoryController {  (6)
14
15    // Contextual Coupling
16    @Autowired
17    private GenerateComplexyHistory generateComplexyHistory;  (1)
18
19    @PostMapping(value = "/generate-history")
20    // Contextual Coupling
21    @ResponseBody
22    public ResponseEntity<?> generate(@Valid
23 (1) GenerateHistoryRequest request, UriComponentsBuilder uriComponent) {
24
25        // Contextual Coupling  (1)
26        InMemoryComplexityHistoryWriter inMemoryWriter =
27        new InMemoryComplexityHistoryWriter();
28
29        new RepoDriller().start(() -> {  (1) // Function as an argument
30            request.toMining(inMemoryWriter).mine();
31        });
32    }
33
34    @PostMapping(value = "/generate-history-class")
35    // Contextual Coupling
36    @ResponseBody
37    public ResponseEntity<?> generatePerClass(@Valid
38 (1) GenerateHistoryPerClassRequest request,
39        UriComponentsBuilder uriComponent) {
40        InMemoryComplexityHistoryWriter inMemoryWriter = new
41        InMemoryComplexityHistoryWriter();
42        new RepoDriller().start(() -> {  (1) // Function as an argument
43            request.toMining(inMemoryWriter).mine();
44        });
45
46        generateComplexyHistory.execute(request,inMemoryWriter.getHistory());
47
48        URI complexityHistoryGroupedReportUri = uriComponent.path(
49                "/reports/pages/complexity-by-class?projectId={projectId}")
50                .buildAndExpand(request.getProjectId()).toUri();
51        return ResponseEntity.created(complexityHistoryGroupedReportUri)
52                .build();
53    }
54 }
```

Figure 1: Class GenerateHistoryController

4 COGNITIVE LOAD ANALYZER

Cognitive Load Analyzer is a plugin for IntelliJ IDEA and Java language to support the adoption of Cognitive-Driven Development. The goal of this tool is to guide developers during programming and reduce cognitive overload in future efforts with software evolution. According to Fowler [7], great programmers write code that humans can understand. Therefore, the easier it is to understand some codes, the more effective it will be to apply the changes and verify that its correctness.

Motivation and problem: Although the CDD effects are promising, developers need to constantly check whether the complexity limit has been reached and currently, such activities are carried out in a manual way.

IntelliJ IDEA is among one of the most adopted and efficient Java IDEs ever released to date. This was the main reason for choosing the platform. Another reason was the advanced support for static analysis of Java source codes through AST (Abstract Syntax Tree), where IntelliJ's SDK already supports several identifiers. This is useful to develop new plugins and improving the programmer's experience.

In the industry, a developers' community adopting the CDD is beginning to emerge, making it naturally appear the need to execute the CDD in a more automated way. Bearing this in mind and taking into account the developers' productivity, a plugin for a specific development IDE is proposed with the aim of analyzing written codes in real-time.

Self-management with respect to the limit of cognitive complexity can increase the effort to use the proposed method. As more rules are adopted when using CDD (in terms of ICPs) more effort is required from their users to manage them during the development process. In a critical scenario, the developer can spend more time with the method than solving problems related to programming.

Features and potential users: Developers can use the Cognitive Load Analyzer to define a feasible cognitive complexity constraint for a certain project and the elements intrinsically complex on the code. From this, the tool is able to automate the calculation of occurrences for some ICPs through a static code analysis.

The tool reminds the developer about the limit established by the team. Every time that this limit is exceeded, the tool can indicate the moment to carry out possible code refactorings to make it more simplified from the perspective of the CDD. Unlike other practices, CDD proposes to assess complexity while coding, guiding the developer on a cognitive complexity threshold established by the development team. For each user interaction on live coding, the code is verified to indicate when the complexity limit was exceeded.

Comparison with related tools: The SonarQube[3] platform is widely used by the development community to assess code quality, detect bugs and vulnerabilities. It has a Cognitive Complexity metric inspired by the work of Campbell [1]. In addition, there is a plugin available on the Jetbrains marketplace inspired by the same work, which implements a live calculation of this metric. However, the plugin was built for .NET and works only in Rider IDE. There are many tools and software quality metrics to assess the source code but to the best of our knowledge, there is no tool to guide development under a complexity constraint.

License: The tool is in the testing phase to verify its effectiveness in terms of supporting the CDD, evaluations for the tool's interface are needed. The tool is not yet available due to the need to run such

[2]https://github.com/asouza/complexity-tracker

[3]https://www.sonarqube.org

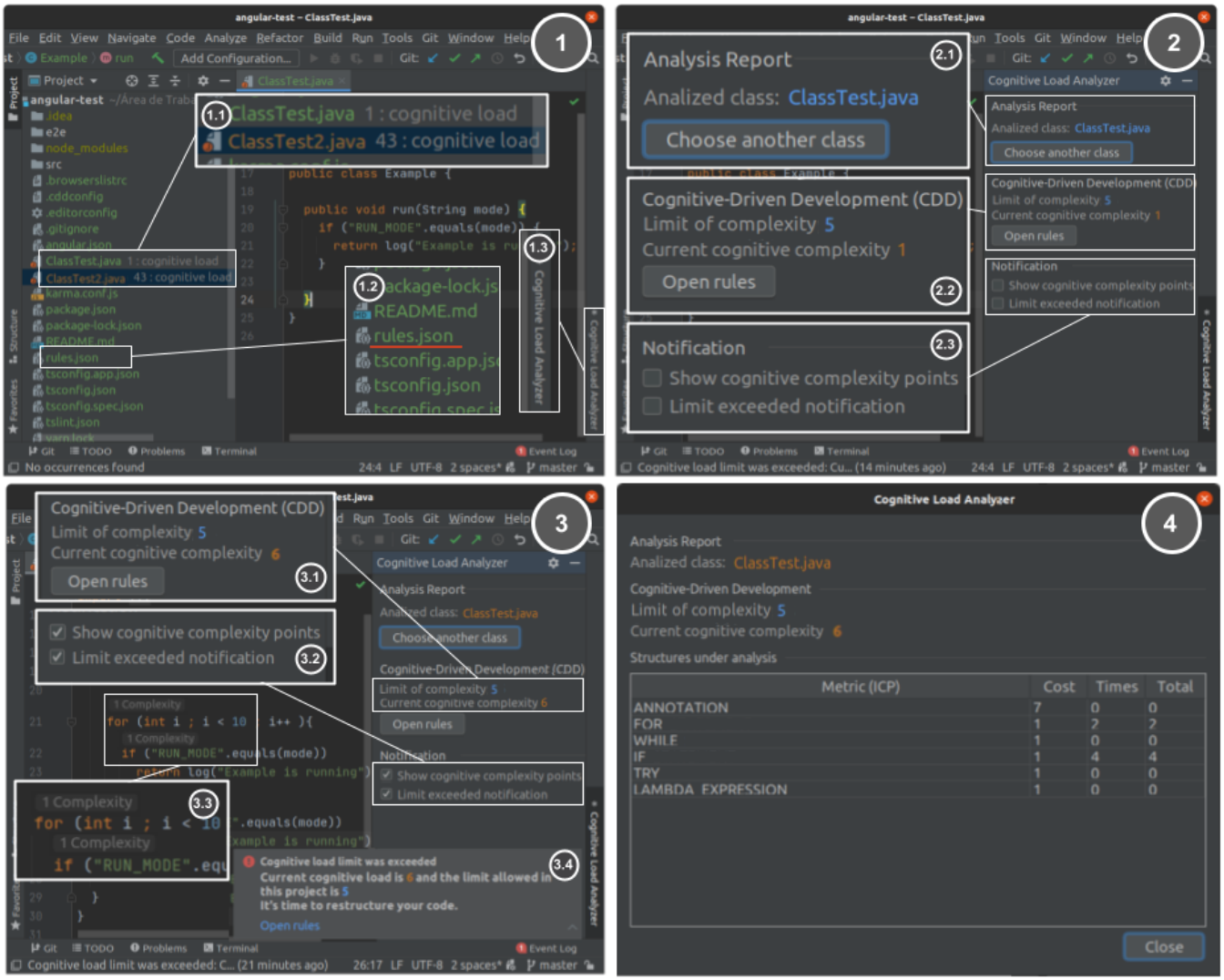

Figure 2: Cognitive Load Analyzer interface

tests involving users. However, after performing the initial tests of the tool, we will make it available under an open-source license. Currently, the most quoted license for the release of the tool is the MIT license.

4.1 Interface and an Illustrative example

The tool's interface was designed to be minimalist, without overloading the developer's vision, precisely so that users can focus mainly on coding.

Figure 2 (**Part 1**) shows the total complexity calculation for each Java file available in the project (1.1). This information is close to the name of Java class within the project view, avoiding the need to open the files to verify their complexity. The visual feedback shows the file name in orange to highlight the classes with the complexity limit exceeded. The tool options appear in the project as soon as a configuration file called "*rules.json*" (1.2) is identified at the root of the project. From this, the tool extracts information about ICPs, costs and limits defined for the project.

Part 2 shows the use of the "*toolWindow*", which is usually hidden, but can be accessed from the right side bar (1.3). The *toolWindow* has three sections: Analysis Report, Cognitive-Driven Development and Notification. In the first section, it is possible to find the name of the analyzed file, together with the option to open another file. Based on Cognitive-Driven Development, information about the allowed complexity limit and the current complexity of the file is shown, as well as the option to view the list of rules (list of ICPs, also shown in Part 4). Finally, in the Notification section we can observe the following options: "Show cognitive complexity points" and "Limit exceeded notifications", as explained in the following.

Part 3 presents an example for "Show cognitive complexity points" notification in a piece of code where an "IF statement" was

defined as an ICP and the limit of complexity for Java classes in the project is 5. It should be noticed that on the line above each "IF/FOR statement" is placed feedback (*//1 Complexity*) indicating the counting points and their corresponding costs (3.3). Besides, these complexity indications also turn orange when exceeding the complexity threshold. The element identified by 3.2 in the **Part 3** demonstrates the use of the "Limit exceeded notification". In the *toolWindow* (3.1) it is possible to identify that the limit of complexity is 5 and the current complexity is 6, so the limit has been reached. When the limit is reached, a *Balloon Notification* appears on IntelliJ IDEA notifying that it can be necessary to restructure the code (3.4). In addition, the link provided in such notification show the rules in a more detailed view. In the *toolWindow* is depicted that when the complexity limit is exceeded, the name of the displayed file also changes color.

Part 4 presents a more detailed view for the ICPs and the status for the complexity constraint in the evidenced project. The ICPs can be viewed using the "Open rules" option shown in *Balloon Notification* when the limit is exceeded, as well as can be accessed at any time through the *toolWindow*. When accessing the functionality, a new window will appear showing the summary of the accounting for each ICP contained in the configuration file, together with their respective costs as well as the number of repetitions in the analyzed file.

4.2 Architecture and main features

The CDD is based on identifying points of code complexity such as code branches, function as an argument, annotations or another element that the development team judged as important in the complexity limit, as aforementioned.

Currently, contextual coupling functionality is not fully functional and embedded in the current version. Figure 3 shows an illustrative list of some elements mapped in the proposed tool. The CDD suggests the value 1 for each point of complexity, so-called ICP, useful to facilitate the manual analysis.

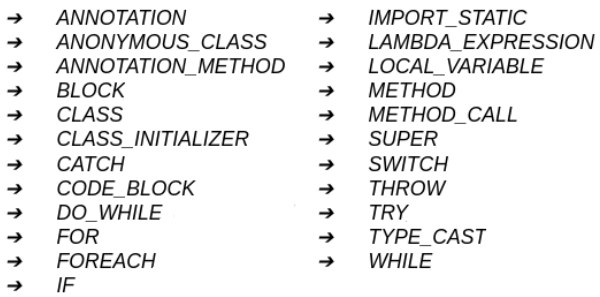

Figure 3: Supported Statements

Cognitive Load Analyzer enables the selection of such elements by the team and the definition of their costs through a structure based on the JSON format (Javascript Object Notation). Figure 4 presents data format for the corresponding setting file. When starting the tool, a file with the name "*rules.json*" is automatically searched for in the project root. This setting file allows the insertion of a numerical value for each ICP that makes sense to the development team. This file must contain an attribute called "*limitOfComplexity*", containing the feasible limit defined by the developers; and an attribute called "rules", which contains the list of the mapping of elements/identifiers ("name") and their corresponding weight (identified as "cost"). From this configuration file, the tool maps all the project's Java files, i.e., it is able to analyze each implementation unit and its changes.

```
{
 "limitOfComplexity": 14,
 "rules": [
   {
     "name": "IF",
     "cost": 1
   },
   {
     "name": "ANNOTATION",
     "cost": 2
   },
   {
     "name": "FOR",
     "cost": 1
   }
 ]
}

```

Figure 4: Data format for rules.json

5 CONCLUSION

Human factors in software engineering impose several challenges. The maintenance can consume more resources than all the effort spent in the creation of new software [17]. As the importance of software increases, practitioners and researchers propose strategies and methodologies that make maintenance of software products more effective. Despite many proposals adopted in the industry are results of experiences of developers and not all have rigorous scientific evaluations, they converge on the same goal: improving software quality to reduce costs related to evolution and testing activities.

Souza and Pinto [19] looked for an answer to the difficulty that programmers have in dealing with the growing software complexity. Based on cognitive complexity measurements and Cognitive Load Theory, they proposed a method called Cognitive-Driven Development. The main principle provided by CDD is to help the developers to define an agreement for code understanding and what are the inherently complex code elements. From this idea, a cognitive complexity constraint can be defined to guide software development. The software can evolve in a healthy way avoiding cognitive overload for developers.

Experimental studies involving CDD were carried out taking into account refactoring scenarios [16]. Another study was focused on investigating the CDD effects in the early stages of software development and if the code produced following CDD principles

demonstrates more effective comprehension levels considering maintenance scenarios. The results suggest that CDD can guide the developers to achieve better quality levels for the software with lower dispersion for the values of quality metrics.

Based on the promising results of the CDD, a support tool would be valuable to broaden its adoption. This paper presents the Cognitive Load Analyzer, a plugin for IntelliJ IDEA and Java, able to show the intrinsic complexity of the code through the static analysis and verify if the limit of complexity is still being kept, leaving the developer focused on solving problems. When starting a project, the development team defines the code elements considering the available statements and covered by the tool and the feasible cognitive complexity limit for each Java class. If during programming the complexity limit is reached for some code, a notification is displayed.

As suggested by Souza and Pinto [19], every intrinsic complexity point has a value of 1 to facilitate manual calculation. As the tool automatically performs this task, the team can distribute the weights according to their needs, exploring the ideal scenario that best suits their project. In addition, the developers do not need to concern with the calculation of the ICPs and rules pointed out by CDD instead, the tool is in charge of carrying out the validations and warnings about possible refactorings. Therefore, the plug-in makes using the CDD less invasive during development, giving developers more time to focus on coding.

As lessons learned, the main challenge to develop the tool was the lack of certain materials in the documentation available by IntelliJ for the use of its SDK and plugins development. The documentation is still being built by the IntelliJ team. This makes it difficult to build new plug-ins that require features that have not yet been documented. Although IntelliJ provides several repositories with code samples, these are still quite limited due to the low number of examples of the resources available. Thus, our exploratory work was essential to understand some features of the SDK. As part of this strategy, a reverse engineering approach was adopted, observing the SKD's source code and exploring the possible implementations of the interfaces. In addition, forums and repositories from other developers were also explored.

As evolution of our work, we intend to carry out studies to evaluate the tool's interface and develop a mechanism to facilitate the integration of elements not yet available for static analysis, such as SQL statements and Exception handlers. Interested developers would be able to include new elements they consider feasible in their projects. We believe that the task the CDD is trying to solve is very important, breakthroughs in it could affect many developers.

REFERENCES

[1] G Ann Campbell. 2021. Cognitive Complexity: A new way of measuring understandability. *SonarSource SA* (2021), 1–21.

[2] Paul Chandler and John Sweller. 1991. Cognitive load theory and the format of instruction. *Cognition and instruction* 8, 4 (1991), 293–332.

[3] A. Cockburn. 2005. Hexagonal architecture. https://alistair.cockburn.us/hexagonal-architecture/. [Online; accessed 2 August 2020].

[4] Edsger W. Dijkstra. 1976. The effective arrangement of logical systems. In *Mathematical Foundations of Computer Science 1976*, Antoni Mazurkiewicz (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 39–51.

[5] Rodrigo Duran, Juha Sorva, and Sofia Leite. 2018. Towards an analysis of program complexity from a cognitive perspective. In *Proceedings of the 2018 ACM Conference on International Computing Education Research*. 21–30.

[6] Eric Evans. 2004. *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional.

[7] Martin Fowler. 2018. *Refactoring: improving the design of existing code*. Addison-Wesley Professional.

[8] Steven D. Fraser, Frederick P. Brooks, Martin Fowler, Ricardo Lopez, Aki Namioka, Linda Northrop, David Lorge Parnas, and David Thomas. 2007. "No Silver Bullet" Reloaded: Retrospective on "Essence and Accidents of Software Engineering". In *Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion* (Montreal, Quebec, Canada) *(OOPSLA '07)*. Association for Computing Machinery, New York, NY, USA, 1026–1030. https://doi.org/10.1145/1297846.1297973

[9] Meir M Lehman. 1979. On understanding laws, evolution, and conservation in the large-program life cycle. *Journal of Systems and Software* 1 (1979), 213–221.

[10] Meir M Lehman. 1980. Programs, life cycles, and laws of software evolution. *Proc. IEEE* 68, 9 (1980), 1060–1076.

[11] Meir M Lehman, Juan F Ramil, Paul D Wernick, Dewayne E Perry, and Wladyslaw M Turski. 1997. Metrics and laws of software evolution-the nineties view. In *Proceedings Fourth International Software Metrics Symposium*. IEEE, 20–32.

[12] Robert C Martin. 2000. Design principles and design patterns. *Object Mentor* 1, 34 (2000), 597.

[13] Robert C Martin. 2018. *Clean architecture: a craftsman's guide to software structure and design*. Prentice Hall.

[14] George A Miller. 1956. The magical number seven, plus or minus two: Some limits on our capacity for processing information. *Psychological review* 63, 2 (1956), 81.

[15] Sanjay Misra, Adewole Adewumi, Luis Fernandez-Sanz, and Robertas Damasevicius. 2018. A suite of object oriented cognitive complexity metrics. *IEEE Access* 6 (2018), 8782–8796.

[16] Victor Pinto., Alberto Tavares de Souza., Yuri Barboza de Oliveira., and Danilo Ribeiro. 2021. Cognitive-Driven Development: Preliminary Results on Software Refactorings. In *Proceedings of the 16th International Conference on Evaluation of Novel Approaches to Software Engineering - Volume 1: ENASE,*. INSTICC, SciTePress, 92–102. https://doi.org/10.5220/0010408100920102

[17] R. Pressman. 2014. *Software Engineering: A Practitioner's Approach*. McGraw-Hill Education.

[18] Jingqiu Shao and Yingxu Wang. 2003. A new measure of software complexity based on cognitive weights. *Canadian Journal of Electrical and Computer Engineering* 28, 2 (2003), 69–74.

[19] Alberto Luiz Oliveira Tavares de Souza and Victor Hugo Santiago C. Pinto. 2020. Toward a Definition of Cognitive-Driven Development. In *Proceedings of 36th IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 776–778.

[20] Kevin J Sullivan, William G Griswold, Yuanfang Cai, and Ben Hallen. 2001. The structure and value of modularity in software design. *ACM SIGSOFT Software Engineering Notes* 26, 5 (2001), 99–108.

[21] John Sweller. 1988. Cognitive load during problem solving: Effects on learning. *Cognitive science* 12, 2 (1988), 257–285.

[22] John Sweller. 2010. Cognitive load theory: Recent theoretical advances. (2010).

[23] Yingxu Wang. 2006. Cognitive complexity of software and its measurement. In *2006 5th IEEE International Conference on Cognitive Informatics*, Vol. 1. IEEE, 226–235.

[24] Elaine J Weyuker. 1988. Evaluating software complexity measures. *IEEE transactions on Software Engineering* 14, 9 (1988), 1357–1365.

[25] Tong Yi and Chun Fang. 2020. A complexity metric for object-oriented software. *International Journal of Computers and Applications* 42, 6 (2020), 544–549.

[26] Horst Zuse. 2019. *Software complexity: measures and methods*. Vol. 4. Walter de Gruyter GmbH & Co KG.