Toward a Definition of Software Component Responsibility based on Human Cognition

Alberto Luiz Oliveira Tavares de Souza and Victor Hugo Santiago C. Pinto
Zup Innovation / Academy - São Paulo, Brazil
alberto.tavares,victor.pinto@{zup.com.br}

***Abstract*—Software separation into components according to responsibility is a recognition that human work can be improved by focus on a limited set of data. The growing complexity of software has always been a challenge to industry. Several approaches have been proposed to support code design based on architectural styles and code quality metrics. However, most research involving human cognition in software engineering is focused on the evaluation of programs and learning instead of how the source code could be developed under this perspective. This paper presents an approach called Cognitive-Driven Development (CDD) that is based on cognitive complexity measurements and Cognitive Load Theory. This strategy can reduce the cognitive overload of the developers through the limitation of intrinsic complexity points from source code. Some cognitive complexity metrics has been extended and some recommendations are presented to calculate the intrinsic complexity points and how the limit for complexity points can be adapted under certain quality criteria. Experimental studies are currently being conducted to evaluate the effects of the CDD in development and refactoring scenarios. Preliminary results indicate that the approach is promising and it can reduce the future effort for maintenance and fixing software faults.**

***Index Terms*—Cognitive complexity, Cognitive-driven development, Software design**

I. Introduction

Separation of concerns is one of the key principles of software engineering [8], [14]. Since the analysis, developers need to understand the problem. A strategy for this is to split it into more understandable blocks. For developing the solution, recommended coding practices and an architectural pattern must be followed to achieve an acceptable modularity and cohesion for the implementation units. However, software complexity increases as new features are incorporated [20], [21], [5] impacting its maintainability, one of the most rewardful software quality attributes [1]. Therefore, the separation of component responsibility must consider not only the domain, but also the software cognitive complexity.

In cognitive psychology, cognitive load refers to the amount of information that working memory resources can hold at one time. Cognitive Load Theory (CLT) [18], [2], [17] is generally discussed in relation to learning. Problems that require a large number of items to be stored in short-term memory may contribute to an excessive cognitive load. According to Sweller [18], some material is intrinsically difficult to understand and this is related to the number of elements that must be simultaneously processed in the working memory. Experimental studies performed by Miller [12] have suggested that humans are generally able to hold only seven plus or minus two units of information in short-term memory. Such limit for information units can be applied for software once the source code has an intrinsic load. The developers are frequently affected by cognitive overload when they need to add a feature, fixing a bug, improve the design or optimize resource usage. Thus, there is too much information that they cannot easily process.

SOLID Design Principles [9], Clean Architecture [10], Hexagonal Architecture [3], and other well know practices are usually adopted in the industry to make the software designs more understandable, flexible and maintainable. According to Domain-driven design (DDD) practices [4], the language of the software code should be aligned with the business domain. Although not all proposals are related to code and modeling, the goal is to mitigate the complexity in the software. Since the improvements are expected in the software development life cycle, the code needs low intrinsic complexity, a quality software attribute [1]. Nevertheless, elements with low cohesion and inefficient separation of responsibilities continue being produced and residing in final releases.

McCabe's cyclomatic complexity (CYC) [11] is a metric for the maximum number of linearly independent paths in a program control graph. The main idea is to identify software modules that will be difficult to test or maintain. CYC can be important to limit code complexity and determine the number of test cases required. However, it can be one of the most difficult code metrics to understand what makes it labored to calculate. Defining an intrinsic complexity point in the software is not clear to developers, even less how to establish a complexity limit for each implementation unit.

This paper presents an approach termed Cognitive-Driven Development (CDD). The proposed approach is based on the CLT and software cognitive complexity. For this purpose, a set of metrics to assess cognitive complexity in object-oriented applications was extended, including a different method to use them. Guidelines to define a limit for intrinsic complexity points in the source code are proposed. Preliminary results suggested that CDD can contribute to reducing the required effort for maintenance and fixing faults.

The remainder of the paper is organized as follows: Section II discusses related work; Section III presents the CDD principles, extensions for some cognitive complexity measures and discussions about their support to limit the cognitive overload for programmers; Finally, Section IV suggests future work.

II. Related work

Object oriented cognitive complexity metrics were proposed in the work of Shao and Wang in [15] and modified by Misra

et al. [13]. Theoretical and empirical validation was conducted to evaluate each metric based on Weyuker's properties [19]. Although the CDD complements such metrics, its main objective is to provide a method to conduct the development that maintain the intrinsic complexity limit.

Gonçales et al. [6] provide a taxonomy of cognitive load in software engineering. Based on this research, recent advances are related to the programming tasks, machine learning techniques to identify the programmer's difficulty level and their code-level comprehensibility. CDD can be considered as a complementary design proposal to mitigate the increase in cognitive complexity regardless of the software size.

III. Cognitive-driven development

Software development targeted to the business domain, modularity, code reuse, high cohesion, low coupling, and technology independence are the essence of object-oriented (OO) programming. However, due the continuous expansion of the software complexity [21], the understandability cannot follow in the same proportion.

CDD is an inspiration from cognitive psychology for software development, considering a reasonable limit for intrinsic complexity points [18]. Table I presents some basic control structures and resources, including their corresponding number for intrinsic complexity points. They are based on metrics to assess cognitive complexity in object-oriented applications [15]. "Coupling" and "Crosscutting requirements" are new categories and the column "Intrinsic complexity points" was chosen instead of weights to clarify our proposed method. Elements depicted here are not limited, developers are free to include additional elements that they consider interesting.

TABLE I
Measurements for intrinsic complexity points

Category	**Basic control structures and resources**	**Intrinsic complexity points**
Branch	*if-else* *Case*	*2* *1*
Exception handling	*try-catch-finally*	*3*
Coupling	Contextual coupling* Functions as an argument*	*1* *1*
Crosscutting requirements	Code related to infrastructure logic, frameworks and library	*0*

For each branch analyzed, one complexity point is increased. For instance, *if-else* has 2 points and *try-catch-finally*, 3. Regarding the coupling, if a class was created to deal with a specific responsibility within the same project but it collaborates with some feature from class under analysis, 1 point is increased. Functions that can accept other functions as arguments, so called higher-order functions, require a certain level of understanding and therefore, they can be considered as an intrinsic complexity point. Code related to Crosscutting requirements is related to infrastructure resources and frameworks. Thus, we suggest not considering such elements in the calculate of the number of intrinsic complexity points.

```
24 @RestController
25 public class NewPaymentController {    8
26
27     @Autowired
28     private CCSoIsValidForOnlineCardValidator 1
29      ccSoIsValidForOnlineCardValidator;
30     @Autowired
31     private ValidPaymentForRestaurantUserValidator 1
32      validPaymentForRestaurantUserValidator;
33
34     @PersistenceContext
35     private EntityManager manager;
36     @Autowired
37     private AllPaymentProcessors allProcessors; 1
38     @Autowired
39     private UserRepository userRepository; 1
40
41     @InitBinder
42     public void init(WebDataBinder binder) {
43         binder.addValidators(ccSoIsValidForOnlineCardValidator,
44                validPaymentForRestaurantUserValidator);
45     }
46
47     @PostMapping(value = "/payments")                1
48     public CompletableFuture<?> execute(@Valid NewPaymentForm form) {
49            PaymentAttempt paymentAttempt =
50                 form.toModel(manager,userRepository);
51        CompletableFuture<TransactionPayment> res =     1
52              allProcessors.pay(paymentAttempt);
53
54         return res.thenApply(transactionPayment -> {     1
55             if(transactionPayment.isOk()) {
56                 return ResponseEntity.ok().build();  1
57             }
58             return ResponseEntity.status(403).build();
59         });
60     }
61 }
```

Fig. 1. Class NewPaymentController

Figure 1 presents a piece of code from a Java class called *NewPaymentController*. At the top of the figure is shown the number 8 corresponding to the total of intrinsic complexity points. The first 6 points are related to contextual coupling (lines 28, 31, 37, 39, 48 and 51) and the remaining points refer to passing a function as an argument and the *if* statement (lines 54 and 55).

In previous experiences training software developers, we have noticed that when they are instructed to not exceed an intrinsic complexity limit during programming, the resulting code generally had high cohesion and modularity levels. CDD is highly flexible for different development scenarios. Regarding the definition of a limit for cognitive complexity, we have some recommendations based on the experiences aforementioned. In the case of web applications and mixed teams in terms of experience, the maximum number of intrinsic complexity points can be more restrictive: five plus or minus two points, where seven would be the limit. For frameworks and libraries, a limit can be hold between ten and twelve points for each implementation unit, considering teams more specialized in a given language.

IV. Future perspectives

As evolution of our work, we intend to explore the following aspects: (i) developing a plugin for *IntelliJ IDEA* [7] to support CDD during programming and for *SonarQube* [16] to estimate the intrinsic complexity points for Java applications; (ii) performing a study to assess the CDD effects for different software quality attributes and (iii) conducting an experimental study to evaluate the level of code understanding for classes with a limited number of intrinsic complexity points.

REFERENCES

[1] International Standard ISO: ISO/IEC 25010–2011. Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models. International Organization for Standardization ISO, 2011.

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

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

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

[5] S. D. Fraser, F. P. Brooks, M. Fowler, R. Lopez, A. Namioka, L. Northrop, D. L. Parnas, and D. Thomas. "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*, OOPSLA '07, page 1026–1030, New York, NY, USA, 2007. Association for Computing Machinery.

[6] L. Gonçales, K. Farias, B. da Silva, and J. Fessler. Measuring the cognitive load of software developers: a systematic mapping study. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pages 42–52. IEEE, 2019.

[7] IntelliJ IDEA. Main page. https://www.jetbrains.com/idea/, 2020. [Online; accessed 5 August 2020].

[8] B. Liskov and S. Zilles. Programming with abstract data types. *ACM Sigplan Notices*, 9(4):50–59, 1974.

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

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

[11] T. J. McCabe. A complexity measure. *IEEE Transactions on software Engineering*, (4):308–320, 1976.

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

[13] S. Misra, A. Adewumi, L. Fernandez-Sanz, and R. Damasevicius. A suite of object oriented cognitive complexity metrics. *IEEE Access*, 6:8782–8796, 2018.

[14] D. L. Parnas. On the criteria to be used in decomposing systems into modules. In *Pioneers and Their Contributions to Software Engineering*, pages 479–498. Springer, 1972.

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

[16] SonarSource. Documentation. https://docs.sonarqube.org/latest/, 2020. [Online; accessed 5 August 2020].

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

[18] J. Sweller. Cognitive load theory: Recent theoretical advances. 2010.

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

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

[21] H. Zuse. *Software complexity: measures and methods*, volume 4. Walter de Gruyter GmbH & Co KG, 2019.