

# Cognitive-Driven Development: Preliminary results on Software Refactorings

Victor Hugo Santiago C. Pinto<sup>1</sup><sup>a</sup>, Alberto Luiz Oliveira Tavares de Souza<sup>1</sup>, Yuri Matheus Barboza de Oliveira<sup>1</sup> and Danilo M. Ribeiro<sup>1</sup>

<sup>1</sup>*Zup Innovation - São Paulo, SP, Brazil*

*{victor.pinto, yuri.oliveira, danilo.ribeiro, alberto.souza}@zup.com.br*

**Keywords:** Cognitive-driven development, Software refactoring, Experimental study

**Abstract:** Refactoring is a maintenance activity intended to restructure code to improve different quality attributes without changing its observable behavior. However, if this activity is not guided by a clear purpose such as reducing complexity and the coupling between objects, the source code can become worse than the previous version. Developers often lose sight of the business problems being solved and forget to manage complexity. As a consequence, many software parts after refactorings continue to have low readability levels. Cognitive-Driven Development (CDD) is our recent proposal to reduce cognitive overload during development when improving the code design. This paper presents an experimental study carried out in an industrial context to evaluate refactorings using conventional practices guided by a cognitive complexity constraint, a principle points out by CDD. Eighteen highly experience participated in this experiment. Different software metrics were used 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 suggest that CDD can guide the restructuring process aiming for a coherent and balanced separation of responsibilities.

## 1 INTRODUCTION

Refactoring is the process of changing the internal structure of software to improve its quality without changing its observable behavior (Fowler et al., 1999). Empirical studies suggested a positive correlation between refactoring operations and code quality metrics (Abid et al., 2020; AlOmar et al., 2019; Alshayeb, 2009; Kataoka et al., 2002). Refactoring can have a positive impact on the readability and maintenance of software systems.


However, many studies point out that often if the refactoring is not guided by a clear purpose, the code can become worse than the original version, or the changes cannot generate a relevant impact (Baqais and Alshayeb, 2020; Alomar, 2019), i.e., the improvements can be questionable. This effect can be more problematic when the refactorings are not guided by a quality metric or the improvements are assessed by a restricted perception of some developers. Deciding when, what and understanding why to restructure some code is still a challenge to developers.

Over the years, continuous expansion of the soft-

ware complexity promotes several discussions in the software engineering community (Zuse, 2019; Clarke et al., 2016; Weyuker, 1988; Shepperd, 1988). Most researchers are continually seeking better and novel methods to deal with the complexity involved in develop and maintenance of software systems. Approaches have been proposed to support code design based on architectural styles and code quality metrics. Nevertheless, there is a lack of practical and clear strategies to change the way that we develop software to reduce efficiently maintenance and testing efforts.

Most research involving human cognition in software engineering focuses on evaluating programs and learning instead of how software development could be guided using this perspective (Duran et al., 2018). Cognitive complexity disrupts from the practice of using strictly numeric values to assess software maintainability. It starts with the precedents set by cyclomatic complexity (CYC) (McCabe, 1976), but uses human judgment to assess the understanding of the code's structures. Object-oriented cognitive complexity metrics were proposed by Shao and Wang's work in (Shao and Wang, 2003) and extended by Misra et al. (Misra et al., 2018), where basic control structures and corresponding weights were suggested. Al-

---

<sup>a</sup> <https://orcid.org/0000-0001-8562-6384>

though the cognitive complexity measurements can contribute to assessing the understanding of source code, there is a lack of works exploring how this perspective could be applied to reducing complexities from the early stages of development until to future costs related to maintenance and testing activities.

This paper extends our recent approach termed Cognitive-Driven Development (CDD) (Souza and Pinto, 2020) that is based on cognitive complexity measurements and Cognitive Load Theory (Sweller, 1988; Sweller, 2010). CDD is an inspiration from cognitive psychology, more specifically, of the recognition of a human limitation to deal with the expansion of software complexity. The need for new empirically based studies concerning the generation of high-quality code using CDD’ principles motivated the execution of an experiment involving refactoring scenarios. Our main premise was that the definition of a cognitive complexity constraint during refactorings could guide the developer to use of more types of refactorings and generate more readable code than conventional refactoring practices without such constraint.

Eighteen software engineers participated in the experiment. Classes from an open-source application called Student Success Portal<sup>1</sup> (SSP) were chosen for refactorings. We analyzed the developers’ productivity considering their time spent in the restructuring process using conventional practices with and without a complexity constraint. Both the original and refactored classes were analyzed considering the following object-oriented metrics: 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 results suggested that CDD is a promising and useful method for restructuring code to achieve better levels of cohesion, readability and separation of responsibilities.

The remainder of the paper is organized as follows: Section 2 discusses the key characteristics of the CDD; Section 3 presents the structure of our experimental study and results. Section 4 describes related work. Finally, Section 5 presents the conclusions and future perspectives.

## 2 BACKGROUND

The continuous expansion of the software scale is one of the main challenges for the industry and a current software engineering problem (Zuse, 2019; Pawade et al., 2016). According to the classic affirmation of

<sup>1</sup><https://github.com/Jasig/SSP>

Robert M. Pirsig: “*There’s so much talk about the system. And so little understanding*” (Pirsig, 1999). As complexity increases for the software, understanding can be compromised from the point of view of people (Kabbur et al., 2020; Briggs and Nunamaker, 2020), once that cannot follow it in the same proportion.

SOLID design principles (Martin, 2000), Clean Architecture (Martin, 2018), Hexagonal Architecture (Cockburn, 2005) and other know practices are usually adopted to make the software designs more flexible and maintainable. Domain-driven design (DDD) practices (Evans, 2004) suggested that the source code should be aligned with the business domain. Although not all proposals are related to code and modeling, the common goal is to provide strategies for dealing with the complexity in different development stages.

However, elements with low cohesion and inefficient separation of responsibilities are still produced and residing in final releases (Souza and Pinto, 2020). To limit frontiers for responsibilities and their complexities is a challenge. Also, the lack of a clear strategy to mitigate it can increase the tangle and spread of the code, consequently, the developers may be affected by cognitive overload.

In the cognitive psychology field, cognitive load means the amount of information that working memory resources can hold at once. Cognitive Load Theory (CLT) (Sweller, 2010; Chandler and Sweller, 1991; Sweller, 1988) is generally discussed in relation to learning and instructional design. Problems that require a large number of items to be stored in our short-term memory may contribute to an excessive cognitive load. According to Sweller (Sweller, 2010), some material has its own complexity and is intrinsically difficult to understand. This proposition is related to the number of elements that we are supposed to be able to process simultaneously in our working memory. According to the experimental studies conducted by Miller (Miller, 1956), humans are generally able to hold only seven more or less two units of information in short-term memory, an important work known as “*magical number 7*”.

Efficient programmers write code that human can understand (Fowler et al., 1999). However, there is often so much information in a single implementation unit that developers cannot easily process it. The recognition of this human limitation can guide the software development, since the source code has an intrinsic load. Cognitive-Driven Development (CDD) is based on cognitive complexity measurements and CLT. **The main contribution provided by CDD is that based on a common concept for understand-**

ing, developers can set a limit on cognitive complexity and implementation units can be kept under this constraint, even with the expansion of the system (Souza and Pinto, 2020).

Despite the theory behind the CDD is not limited to specific metrics or numerical values to assess the code readability, cognitive complexity metrics (Shao and Wang, 2003) were considered to introduce the concept of intrinsic complexity points (ICPs). In our previous work (Souza and Pinto, 2020), the proposed idea was to consider some basic control structures (BCS) and resources as ICPs. It is important to highlight that other possibilities may be valid. For instance, we could define metrics that cover different types of projects (web applications, libraries, etc.), programming languages and developer experience levels.

Considering our guidelines, the total of ICPs can be increased for each branch analyzed. For instance, *if-else* has 2 points and *try-catch-finally*, 3. For *contextual coupling*, if a class was created to deal with a specific responsibility inside the project, however it collaborates with the class under analysis, 1 point is also increased. Functions that can accept other *functions as arguments*, so called higher-order functions, can also be considered as ICPs.

The coupling with objects provided by a certain framework can be included in the total of ICPs, but it is important to note that code related to *crosscutting requirements* are mainly related to infrastructure resources, i.e., they are reused or extended for specific project purposes. Although such code is not trivial, we suggest not considering them for the following reasons: (i) they are less changed than code related to business logic and (ii) it is expected that the practitioners widely known the language and frameworks resources. In previous experiences with real development scenarios, we observed that such definitions can be used to improve team's level. However, this still requires further investigations and empirical-based experiments.

### 3 EXPERIMENT

This section describes an empirical study that compares object-oriented metrics for original Java classes and their corresponding refactored versions. A set containing 145 Java web projects with more than 500 revisions were collected from [github](https://github.com/)<sup>2</sup> using a domain-specific language and infrastructure for mining software repositories called *BOA* (Dyer et al.,

2013). Such projects were analyzed to identify representative classes and approximate complexity of those found in real projects from the industry. Student Success Plan<sup>3</sup> (SSP) was the application chosen under such criteria. The experiment aimed to investigate if the refactoring guided by a cognitive complexity constraint suggested by CDD results in code with higher quality than produced using only conventional refactoring practices without a limit. For this, we carried out static analysis through object-oriented metrics considering all versions.

#### 3.1 Planning

The experiment was planned following the guidelines of the model *Goal Question Metric (GQM)* (Van Solingen et al., 2002) to define the goals and evaluation methods. For the experimentation process, the principles proposed by Wohlin et al. (Wohlin et al., 2012) were considered. The characterization of this study can be formally summarized as follows:

*For Analyzing resulting pieces of code from refactorings with conventional methods and CDD's principles aiming to compare their quality through object-oriented metrics, regarding the complexity reduction under the point of view of software engineers in the context of industry.*

The questions addressed, formulated hypotheses, and objectives for the experimental study are described in detail as follows.

**RQ<sub>1</sub>: Is there a difference in terms of quality metrics for refactorings performed with a strategy based on CDD and conventional methods?** To answer this question, all resultant refactorings were evaluated using a tool for static analysis to collect values for object-oriented metrics. **RQ<sub>2</sub>: Is productivity different when using a strategy based on CDD and conventional practices for refactorings?** For this question, we gathered and assessed the time spent to make the refactorings. The time spent includes all steps involved in refactoring activity since the code analysis to identify the potential candidate structures until the compilation process.

Notice that the single difference when using CDD here is to identify the ICPs and use a feasible limit to guide the refactorings. We suggested a strategy for the subjects to define a limit and proceed with refactorings. First, the idea was to calculate the number of ICPs looking at the refactoring target. Second, they could organize growing challenges, such as reducing 10%, 30% and even 50% the number of ICPs in a

<sup>2</sup><https://github.com/>

<sup>3</sup><https://github.com/Jasig/SSP.git>

class under refactoring. Finally, the subjects could restructure the refactoring cluster to keep all implementation units under a balanced complexity level.

The planning phase was divided in six parts that are described in the next subsections.

### 3.1.1 Context selection

The experiment was conducted involving full and senior developers from the same company and it was performed in a controlled way.

### 3.1.2 Hypothesis Formulation

The  $RQ_1$  was formalized into two hypotheses. **Null hypothesis ( $H_0$ )**: There is no difference between the conventional refactoring practices and the use of CDD based refactoring, considering the quality metrics adopted in this study.

**Alternative Hypothesis ( $H_1$ )**: There is a difference between conventional refactoring methods and the use of a strategy based on CDD, under the perspective of the quality metrics considered. Such hypotheses can be formalized by equations 1 and 2:

$$H_0 : (\mu_{Conventional}^{metrics} = \mu_{CDD}^{metrics}) \quad (1)$$

$$H_1 : (\mu_{Conventional}^{metrics} \neq \mu_{CDD}^{metrics}) \quad (2)$$

Similarly, the  $RQ_2$  was also formalized in two hypotheses. **Null hypothesis ( $H_0$ )**: There is no significant difference between the productivity of the developers taking into account the time spent for refactorings with conventional practices and a strategy based on CDD, they are equivalent. **Alternative hypothesis ( $H_1$ )**: There is a difference between the time spent on refactorings employing conventional practices and a strategy based on CDD. The hypotheses for the  $RQ_2$  can be formalized by equations 3 and 4:

$$H_0 : (\mu_{Conventional}^{time} = \mu_{CDD}^{time}) \quad (3)$$

$$H_1 : (\mu_{Conventional}^{time} \neq \mu_{CDD}^{time}) \quad (4)$$

### 3.1.3 Variable Selection

The dependent variables are: “**the values from static analysis for object-oriented metrics (CBO, WMC, RFC, LCOM and LOC)**” and “**time spent refactoring the provided classes**”. CBO (*Coupling between objects*): counts the number of dependencies for a certain class, such as field declaration, method return types, variable declarations, etc. For this experiment, dependencies to Java itself were ignored. WMC (*Weight Method Class*), so-called McCabe’s complexity (McCabe, 1976), counts the number of branch instructions in a class. RFC (*Response for a*

*Class*) counts the number of unique method invocations in a class. LCOM (*Lack of Cohesion of Methods*) calculates LCOM metric. Finally, LOC (*Lines of code*) counts the lines of code, ignoring empty lines and comments.

The independent variables is the **SSP**, specifically **the candidate classes to be refactored in the experiment**: The subjects were asked to refactoring two components from the given application. The main differences between such features are their complexity and each subject should apply only one method, i.e., either conventional practices or CDD for refactorings.

### 3.1.4 Selection of Subjects

Subjects were selected according to convenience sampling (Wohlin et al., 2012). Eighteen software engineers that attended the experiment were working in the development of web projects and they had some knowledge level in the Java language.

### 3.1.5 Experiment Design

The experimental principle of subjects grouping in homogeneous blocks (Wohlin et al., 2012) was adopted to increase the accuracy of this experiment. We look for ways to soften out interference from the experience level of the subjects in the treatment outcomes. Two pilot experiments were carried out with a restricted number of subjects. It is important to note that they were not included in the real experiment and the gathered data were useful to select proper classes for refactorings and rearranged the groups for the real experiment.

For separating the subjects into balanced groups, we first asked them to fill out a *Categorization Form* with questions about their experience in themes related to the experiment, a self-evaluation. Based on data, we divided them into two blocks: one with eight subjects and another with ten subjects. The first group should apply conventional practices to improve the readability of the provided classes, while the second group attended a previous training for employing the CDD’ principles for refactorings.

The *Categorization Form* included questions regarding the knowledge about: (i) Object-oriented programming, Java, number of books read about software development (e.g., Java, *Clean Code*, *Clean Architecture*, *Domain-Driven Design*, etc.), number of real (corporate) projects with active participation, number of project for “training and learning”, i.e., personal projects aiming to improve technical skills; (ii) software metrics known to them and that eventually can be used to improve code cohesion and separation of responsibilities; (iii) practices for software

refactorings; (iv) practices usually used to minimally understand a legacy project and start implementing new features; (v) programming practices and code design; Finally, (vi) testing activity and tools;

Figure 1 describes the results of the application of this form in a grouped bar graph, considering only numeric values (number of books, real and personal projects) for each subject. The main reason to use these elements is that the time of experience is a relative measurement. For instance, it is likely that a programmer with low time in the development but who attended a higher number of projects can perform better than a person with more time of experience and attended a low number of projects. The subjects “S1-S7;S17” belong to the group defined to use conventional practices and the subjects “S8-S16;S18”, CDD for refactorings.

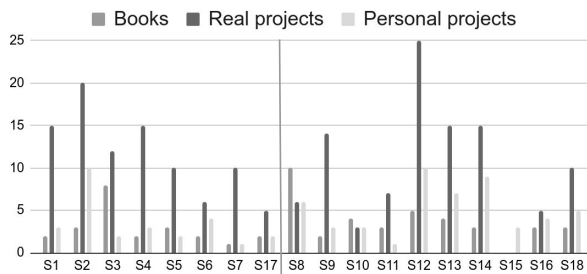


Figure 1: Gathered data with the Categorization Form

Additional information was considered to define this separation which are described as follows, including corresponding percentages of answers.

For **software metrics**, we ask them which ones they use to generate code thinking about readability, cohesion and division of responsibilities, the options/answers (not limited) were as follows: *Fan-in/Fan-out* (5.5%), *Cyclomatic Complexity* (50%), *KLOC* (11%), *Number of root classes* (0 %), *Coupling between objects* (72 %), *Lack of cohesion in methods* (27 %), *Class size* (67 %), *Coupling factor* (16 %) or *Software Maturity Index (SMI)* value (0%).

As regarding the **practices for software refactoring**, the most subjects reported the importance of following principles from *Clean Code*, *Clean Architecture*, *SOLID*, *Design patterns*, *Complexity analysis*, identifying classes or methods overloaded with various tasks, *DDD* for defining domains and separating better the responsibilities. Another question was targeted to the **practices to minimally understand a legacy project** to include new features or changed any one available. The strategies in most cases were related to searching for data entry points from the user’s perspective and from there, trace the execution flows or perform debugging and analysis of automated tests.

Considering **programming practices**, the options/answers (not limited) were as follows: *Clean Architecture* (55 %), *SOLID* (83 %), *DDD* (55 %), *TDD* (33 %), *Conventional practices for code cohesion* (33%) and other practices (5 %). Finally, for **testing techniques and tools**: *functional testing techniques* (72 %), *structural testing techniques* (39 %), *defect-based testing (mutation testing)* (0 %) and *ad hoc testing (non-systematic testing)* (56 %). For tools, *JUnit* (89 %), *Mockito* (78 %), *Cucumber* (11 %), *Selenium* (5.5 %), *Jest* (15 %), *Mocha* (0 %), *REST Assured* (28 %) and *pitest* (0 %).

### 3.1.6 Instrumentation

A document was provided to the subjects describing constraints and guidelines to assist them both refactoring as the data submission process, as follows:

- The package structure should be kept and they could not create new packages;
- Automated tests must be kept working completely without changes;
- Public methods, protected or package-private could not be removed from the original classes.

Regarding the guidelines, our suggestion was to clone the SSP repository from gitHub and import it into IDEs. The classes chosen for refactorings were *JournalEntryServiceImpl* and *EarlyAlertServiceImpl* considering cognitive complexity criteria, i.e., a considerable amount of ICPs when comparing classes in real projects. After the refactorings, the subjects were informed to submit the URLs of their remote repositories, time spent and perceptions about the experiment using a web form.

## 3.2 Operation

Once the experiment had been defined and planned, it was performed according to the following steps: preparation, operation, and validation of the collected data.

### 3.2.1 Preparation

At this stage, the subjects got committed with the experiment and they were made aware its purpose. They accepted the confidentiality terms regarding the provided data, which would be only used for research purposes, and their freedom to withdraw, by signing a *Consent Form*. In addition, other objects were provided:

- *Characterization Form*: Questionnaire in which the subjects assessed their knowledge about on the technologies and concepts used in the experiment;

- *Instructions*: Document describing all steps, including instructions about the SSP and classes chosen for refactoring;
- *Data Collection Form*: Document to be filled by the participants to record the start and finishing time of each activity during the experiment.

The platform adopted had Java as implementation language and Eclipse or IntelliJ IDEA as development environments. The group that would use CDD received 1 hour training in web meeting format. A class from a real project was selected to exemplify the identifying process of ICPs. CDD fundamentals were explained highlighting the importance to define a cognitive complexity constraint to guide the refactorings (Souza and Pinto, 2020). Complementary materials were provided and a web chat was created for doubts before the experiment, during one week.

### 3.3 Data Analysis

This section presents our findings. The analysis is divided into two points: (i) descriptive statistics and (ii) hypotheses testing.

#### 3.3.1 Descriptive Statistics

The quality of the input data (Wohlin et al., 2012) was verified before applying statistical methods. Incorrect data sets can be obtained due to some error or the presence of outliers, which are data values much higher or much lower when compared with the remaining data.

Metrics adopted in this study have a different scales and when considering the “refactoring clusters” we decided to be conservative and analyzing all gathered data, in a individual way per metric. Refactoring clusters can be understood as the set of changes produced during refactoring, as new classes or interfaces, including implementation units provided by SSP project that were modified.

To clarify descriptive statistics and comparisons, it is important to keep in mind some values. For original version of the Class *JournalEntryServiceImpl* the following values were collected: *CBO*=23; *WMC*=20; *RFC*=27; *LCOM*=15 and *LOC*=82. Similarly, *EarlyAlertServiceImpl*, *CBO*=70; *WMC*=136; *RFC*=161; *LCOM*=136 and *LOC*=560, respectively.

Averages were calculated for each refactoring cluster per subject. Figures 2 and 3 present such data for *JournalEntryServiceImpl* and *EarlyAlertServiceImpl* refactoring clusters, respectively. The difference between the number of subjects that attended both refactorings is because not all subjects finished the activities during the time of four hours and thirty

minutes. With regarding the averages, clusters generated following the cognitive complexity constraint (called only CDD) tend to keep more balanced and lower levels for the metrics than refactorings without the restriction.

Although it is not possible to have a conclusive result when analyzing these values, we believe that the adoption of a quality criterion based on understanding can generate code with better modularity levels. For instance, in Figure 2 the gathered data when CDD was followed the *RFC* was kept between 10 and 20, in most cases; *LCOM* 5 and 10; and *LOC*, 20 and 40 on average. As regarding the same metrics when conventional practices were applied without a complexity constraint we can note that the limits with higher values in the intervals than the aforementioned data.

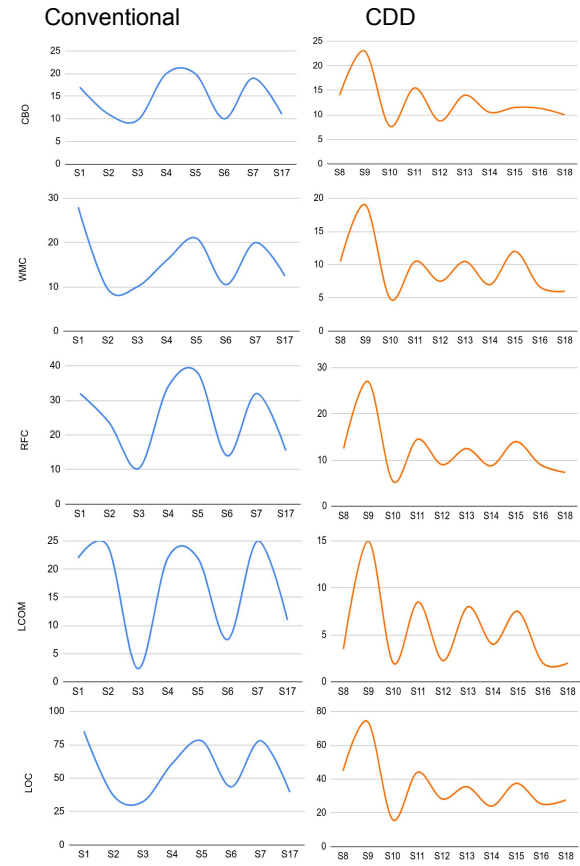


Figure 2: *JournalEntryServiceImpl* refactoring clusters

Figure 4 presents box plots for each metric per samples. The box plots “A-E” for *JournalEntryServiceImpl* and “F-J” *EarlyAlertServiceImpl*, *CBO*, *WMC*, *RFC*, *LCOM* and *LOC*, respectively. Note that each box inside of a box plot refers to a specific sample, “Conventional”, where the subjects strictly used the conventional practices for refactoring or “CDD”, using the same practices but guided by a cognitive com-

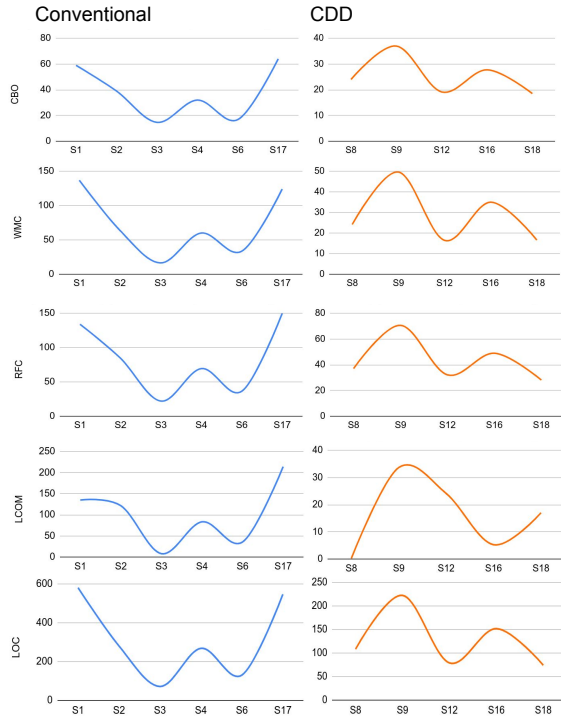


Figure 3: *EarlyAlertServiceImpl* refactoring clusters

plexity constraint.

In addition, Table 1 presents the types of refactoring detected using *Refactoring Miner* (Tsantalis et al., 2018), a API that can detect types of refactorings applied in the history of a Java project. As a result, 24 different types of refactorings were identified considering all changes produced in this study. This graph is useful to observe that the subjects (S8-S16;S18) that followed a complexity constraint were implicitly guided to explore more types of refactorings than remaining developers.

Figure 5 shows two box plots based on time spent by all subjects (grouped by Time (min)). The boxes inside of each graph refer to specific samples, Conventional and CDD. For *JournalEntryServiceImpl* (A) the average of time spent using CDD was higher in the sample where the subjects were worried about the cognitive complexity constraint, it is likely the main reason for this difference. As regarding the *EarlyAlertServiceImpl* (B), some subjects were guided by a constraint for separating the responsibilities in more implementation units, but the remaining subjects that also adopted this constraint kept their focus only on the target class, indicated in the experiment, therefore, they spent less time to conclude their activities. As a consequence, the interquartile range represented by box plot (B) was higher than the first box plot (A).

### 3.3.2 Hypotheses Testing

**Hypothesis Testing - Metrics:** Since some statistical tests only apply if the population follows a normal distribution, before choosing a statistical test we examined whether our gathered data departs from linearity. Therefore, we used the Shapiro-Wilk normality test to verify if the samples had normal (ND) or non-normal distribution (NND).

Table 2 present the results for the normality testing for all samples, i.e., for *JournalEntryServiceImpl* and *EarlyAlertServiceImpl* refactorings clusters, respectively. Results refer to the analysis for each metric, including the corresponding samples. For instance, taking into account the metric *CBO* from the sample “CDD”, we do not reject the hypothesis that the data are from a normally distributed population, however the sample “Conventional” follows a non-normal distribution. Variance testing were performed for *WMC* and *LOC* considering *JournalEntryServiceImpl* refactoring clusters and *CBO* for the samples related to the *EarlyAlertServiceImpl*. Therefore, for such metrics the *p-values* were 0.1841, 0.4392 and 0.6918 considering  $\alpha = 0.05$ , respectively.

Unpaired Two-Samples T-test or (unpaired t-test) can be used to compare the means of two unrelated groups of samples. Such statistical testing was executed for *WMC* and *LOC* considering *JournalEntryServiceImpl* refactoring clusters and the *p-values* were 0.02109 and 0.02993. Therefore, with degrees of freedom being  $df = 16$ , it is possible to reject the null hypothesis for such metrics. However, there is no considerable difference between the means for *CBO* of the *EarlyAlertServiceImpl* refactoring clusters.

Mann-Whitney U Test is a nonparametric test that can be used when one of the samples does not follow a normal distribution. We applied such testing for *CBO*, *RFC* and *LCOM* considering *JournalEntryServiceImpl* refactoring clusters. As results, *p-values* were 0.4763, 0.007561 and 0.009632, respectively. Therefore, it is impossible to reject the null hypothesis for *CBO*, nevertheless for *RFC* and *LCOM* the null hypothesis can be rejected. As regarding the *WMC*, *RFC*, *LCOM* and *LOC* for *EarlyAlertServiceImpl*, the *p-values* were as follows: 0.2615, 0.6304, 0.1488 and 0.3358, therefore, it is not possible to reject the null hypothesis for such samples.

Summarizing the results, when comparing samples for *JournalEntryServiceImpl* refactoring clusters, specifically the gathered data for metrics following only conventional refactoring practices (Conventional) against the same practices but guided by a cognitive complexity constraint (CDD), we can conclude that there was no statistical difference for



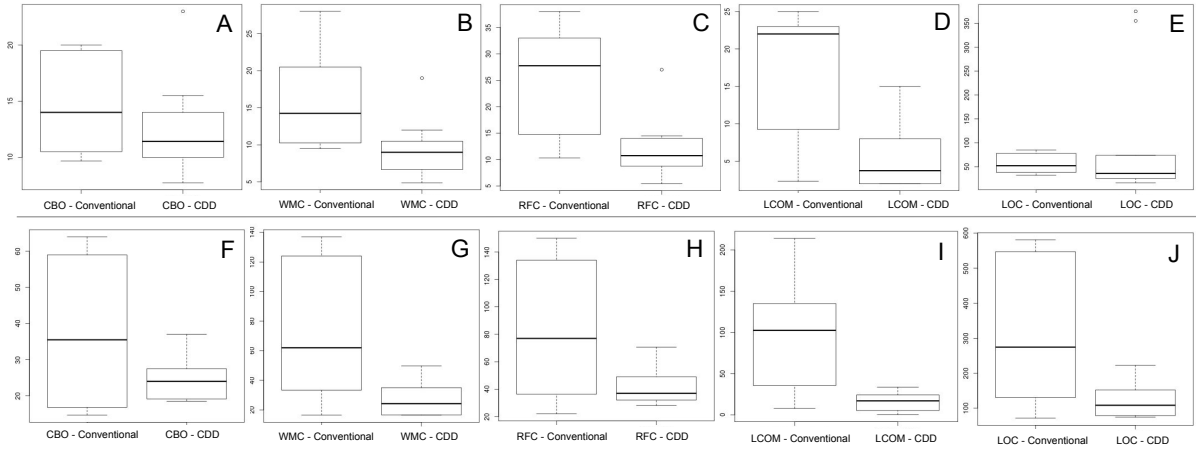


Figure 4: Box plots for *JournalEntryServiceImpl* (A-E) and *EarlyAlertServiceImpl* (F-J) refactoring clusters

Table 1: Types of refactorings detected by Refactoring Miner considering all changes

		Extract Method	Inline Method	Rename Method	Move Method	Move Attribute	Rename Class	Extract and Move Method	Extract Class	Extract Variable	Parameterize Variable	Rename Parameter	Rename Attribute	Replace Variable with Attribute	Change Variable Type	Change Parameter Type	Change Return Type	Change Attribute Type	Move and Rename Method	Add Method Annotation	Remove Method Annotation	Remove Attribute Annotation	Remove Parameter Annotation	Add Parameter	Remove Parameter
Conv.	S1	2													3						20				
	S2	4			4				1	1															
	S3				4	19			3				1						3		20				
	S4	3																							
	S5																								
	S6	2			2				1																
	S7																								
	S17	5								1	2						1					20	1		
CDD	S8				7	13		2	4																1
	S9	10			3				1										3					2	
	S10					2		1															1		
	S11				6	10			4				1												
	S12		1	1	2	7	1		4		1				3										
	S13	3	6		1				1										1	1					
	S14					1	1		1				1					1							
	S15										1														
	S16				3	6			5										3						
	S18				3	11		2	6		2	1			2	1			3					2	

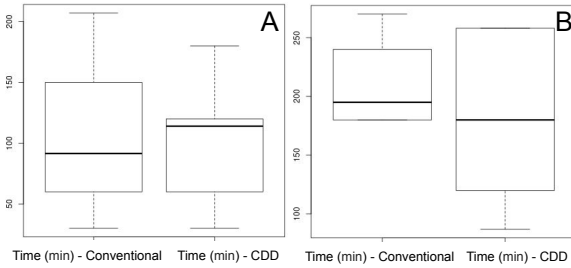


Figure 5: Box plots for the time spent by the subjects when refactoring *JournalEntryServiceImpl* (A) and *EarlyAlertServiceImpl* (B)

*CBO*, but there was for the remaining metrics *WMC*, *LCOM*, *LOC* and *RFC*, i.e., the use of a cognitive complexity constraint was useful for the reduction of

the code complexity. However, for *EarlyAlertServiceImpl* refactoring clusters, we concluded that there was no statistically significant difference between the observed samples. This can still be a great result if there is a reduction of intrinsic complexity points for the code generated following a complexity constraint.

**Hypothesis Testing - Time:** Similarly, we applied statistical tests to evaluate the time spent on refactoring activities. According to normality tests, we cannot reject the hypothesis that all time samples have a normal distribution,  $p\text{-value} = 0.2384$  and  $0.4711$  for *JournalEntryServiceImpl*;  $0.3306$  and  $0.1099$  for *EarlyAlertServiceImpl* refactoring clusters with and without the suggested complexity constraint, respectively. Considering the variance testing, were obtained  $p\text{-value} = 0.3601$  for *JournalEntrySer-*



Table 2: Shapiro-Wilk normality tests for *JournalEntryServiceImpl* and *EarlyAlertServiceImpl* refactoring clusters

JournalEntryServiceImpl			
Metric	Samples		Results
CBO	Conventional	<i>p-value = 0.03153</i>	NND
	CDD	<i>p-value = 0.1042</i>	
WMC	Conventional	<i>p-value = 0.256</i>	ND
	CDD	<i>p-value = 0.09061</i>	
RFC	Conventional	<i>p-value = 0.2886</i>	NND
	CDD	<i>p-value = 0.02513</i>	
LCOM	Conventional	<i>p-value = 0.04899</i>	NND
	CDD	<i>p-value = 0.02381</i>	
LOC	Conventional	<i>p-value = 0.1484</i>	ND
	CDD	<i>p-value = 0.1191</i>	
EarlyAlertServiceImpl			
CBO	Conventional	<i>p-value = 0.4334</i>	ND
	CDD	<i>p-value = 0.07191</i>	
WMC	Conventional	<i>p-value = 0.4435</i>	NND
	CDD	<i>p-value = 0.0196</i>	
RFC	Conventional	<i>p-value = 0.6283</i>	NND
	CDD	<i>p-value = 0.02769</i>	
LCOM	Conventional	<i>p-value = 0.9084</i>	NND
	CDD	<i>p-value = 0.001319</i>	
LOC	Conventional	<i>p-value = 0.3581</i>	NND
	CDD	<i>p-value = 0.01908</i>	

*viceImpl* and 0.1451 for *EarlyAlertServiceImpl*. Finally, Two sample t testing returned  $p\text{-value} = 0.7761$  and 0.4338. Therefore, it is not possible to reject the null hypothesis for the time spent between the samples, i.e., productivity in terms of time was not affected by employing a complexity constraint, since there is no statistical difference.

### 3.4 Threats to Validity

**Internal Validity.** *Experience Level of Subjects:* One can argue that the heterogeneous knowledge of the subjects could have affected the collected data. To decrease this threat, the participants were organized into two-balanced blocks considering their experience level. During the training, the subjects that should applying the cognitive complexity constraint, attended a training on how to use this limit to guide the refactoring process. Thus, they applied conventional practices to restructure the code as the other group but with such difference;

*Productivity under evaluation:* Results could be affected because the subjects often tend to think they are being evaluated during the experiment. In order to mitigate this, we explained to the subjects that no one was being evaluated and their participation would be

considered anonymous;

**Validity by Construction.** *Hypothesis expectations:* the subjects already knew the researchers, in which reflects one of our hypotheses. This issue could affect the collected data and cause the experiment to be less impartial. In order to keep impartiality, we enforced that the participants had to keep a steady pace during the whole study.

**External Validity.** *Interaction between configuration and treatment:* it is possible that the exercises performed in the experiment are not accurate for every Java web application. To mitigate this threat, an open-source application was selected based on the real-world criterion, i.e., considering the complexity of applications that the researchers have contact in the company.

**Conclusion Validity.** *Measure reliability:* it refers to the metrics used to measure the refactoring effort. To mitigate this threat we have used only the time spent, which was captured in forms filled by the subjects;

*Low statistic power:* the ability of a statistical test is in reveals reliable data. Unpaired Two-Samples T-test and Mann-Whitney U Test were adopted to statistically analyze the metrics for all refactoring clusters and the time spent during the restructuring process.

## 4 RELATED WORK

Duran et al. (Duran et al., 2018) have proposed a framework of Cognitive Complexity of Computer Programs (CCCP) that describes programs in terms of the demands they place on human cognition. CCCP is based on a model that recognizes aspects when we are mentally manipulating a program. The contribution of this work concentrates on the cognitive complexity present in program designs instead of how the source code could be developed or restructured under this perspective, as suggested by CDD.

Gonales et al. (Gonales et al., 2019) presented a classification of cognitive load in software engineering. 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 for tackling the increase in cognitive complexity regardless of the software size. Cognitive Complexity as an expressive metric to measure code understandability was evaluated by Muoz Bar3n et al.’s work in (Muoz Bar3n et al., 2020). For this, a systematic literature review was conducted and based on results Cognitive Complexity positively correlates with comprehension time and subjective ratings of understandability. This reinforces the impor-

tance of an intrinsic complexity constraint for the implementation units.

## 5 CONCLUSIONS

The human factor imposes several challenges in software development. For instance, the developers' varying level of experience or team restructuring during the software process can impact software estimates and quality. As writing and maintaining code are human processes, the priority is not only to solve business problems, but also to write code that other people can understand.

However, complex objects without a clear definition of class roles are still produced and reside in final releases, even when developers widely familiar with certain programming language and development stacks resources. Cognitive Load Theory is a framework for investigating the effects of human cognition on task performance and learning (Sweller, 1988; Sweller, 2010). Cognition is constrained by a bottleneck created by working memory, in which we humans can hold only a handful of elements at a time for active processing; to the best of our knowledge, the cognitive complexity constraint has not been applied previously to guide software development. Thus, we proposed a method called Cognitive-driven development (CDD) in which a pre-defined cognitive complexity for application code can be used to limit the number of intrinsic complexity points and tackling the growing software complexity, reducing the cognitive overload.

The main focus of this work was to assess the impacts of adopting of a complexity constraint on refactorings and on the productivity of developers. Despite we do not have conclusive results, refactorings using conventional practices guided by a complexity limit were better evaluated when comparing them with the refactoring clusters generated without such constraint. The main findings of our experiment showed that, in terms of productivity, there was no statistically significant difference between samples of time spent on refactorings with and without complexity constraint. A package containing the tools, materials and more details about the experiment steps is available at <http://bit.ly/3mElnp5>.

As future perspectives, we intend to explore the following aspects: (i) define an automated refactoring strategy using search-based refactoring and cognitive complexity constraints and (ii) carry out new empirical-based studies in an industrial context to evaluate restructured projects with CDD' principles, exploring the number of faults and understanding in

the medium and long term of development.

## REFERENCES

- Abid, C., Kessentini, M., Alizadeh, V., Dhouadi, M., and Kazman, R. (2020). How does refactoring impact security when improving quality? a security-aware refactoring approach. *IEEE Transactions on Software Engineering*.
- Alomar, E. A. (2019). Towards better understanding developer perception of refactoring. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 624–628. IEEE.
- AlOmar, E. A., Mkaouer, M. W., Ouni, A., and Kessentini, M. (2019). On the impact of refactoring on the relationship between quality attributes and design metrics. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–11. IEEE.
- Alshayeb, M. (2009). Empirical investigation of refactoring effect on software quality. *Information and software technology*, 51(9):1319–1326.
- Baqais, A. A. B. and Alshayeb, M. (2020). Automatic software refactoring: a systematic literature review. *Software Quality Journal*, 28(2):459–502.
- Briggs, R. O. and Nunamaker, J. F. (2020). The growing complexity of enterprise software.
- Chandler, P. and Sweller, J. (1991). Cognitive load theory and the format of instruction. *Cognition and instruction*, 8(4):293–332.
- Clarke, P., O'Connor, R. V., and Leavy, B. (2016). A complexity theory viewpoint on the software development process and situational context. In *Proceedings of the International Conference on Software and Systems Process*, pages 86–90.
- Cockburn, A. (2005). Hexagonal architecture. <https://alistair.cockburn.us/hexagonal-architecture/>. [Online; accessed 2 August 2020].
- Duran, R., Sorva, J., and Leite, S. (2018). Towards an analysis of program complexity from a cognitive perspective. In *Proceedings of the 2018 ACM Conference on International Computing Education Research*, pages 21–30.
- Dyer, R., Nguyen, H. A., Rajan, H., and Nguyen, T. N. (2013). Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 422–431. IEEE.
- Evans, E. (2004). *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional.
- Fowler, M., Beck, K., Brant, J., Opdyke, W., and Roberts, D. (1999). Refactoring: improving the design of existing code, ser. In *Addison Wesley object technology series*. Addison-Wesley.

- Gonçales, L., Farias, K., da Silva, B., and Fessler, J. (2019). 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.
- Kabbur, P. K., Mani, V., and Schuelein, J. (2020). Prioritizing trust in a globally distributed software engineering team to overcome complexity and make releases a non-event. In *Proceedings of the 15th International Conference on Global Software Engineering*, pages 66–70.
- Kataoka, Y., Imai, T., Andou, H., and Fukaya, T. (2002). A quantitative evaluation of maintainability enhancement by refactoring. In *International Conference on Software Maintenance, 2002. Proceedings.*, pages 576–585. IEEE.
- Martin, R. C. (2000). Design principles and design patterns. *Object Mentor*, 1(34):597.
- Martin, R. C. (2018). *Clean architecture: a craftsman's guide to software structure and design*. Prentice Hall.
- McCabe, T. J. (1976). A complexity measure. *IEEE Transactions on software Engineering*, (4):308–320.
- Miller, G. A. (1956). The magical number seven, plus or minus two: Some limits on our capacity for processing information. *Psychological review*, 63(2):81.
- Misra, S., Adewumi, A., Fernandez-Sanz, L., and Damasevicius, R. (2018). A suite of object oriented cognitive complexity metrics. *IEEE Access*, 6:8782–8796.
- Muñoz Barón, M., Wyrich, M., and Wagner, S. (2020). An empirical validation of cognitive complexity as a measure of source code understandability. In *Proceedings of the 14th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–12.
- Parwade, D., Dave, D. J., and Kamath, A. (2016). Exploring software complexity metric from procedure oriented to object oriented. In *2016 6th International Conference-Cloud System and Big Data Engineering (Confluence)*, pages 630–634. IEEE.
- Pirsig, R. M. (1999). *Zen and the art of motorcycle maintenance: An inquiry into values*. Random House.
- Shao, J. and Wang, Y. (2003). A new measure of software complexity based on cognitive weights. *Canadian Journal of Electrical and Computer Engineering*, 28(2):69–74.
- Shepperd, M. (1988). A critique of cyclomatic complexity as a software metric. *Software Engineering Journal*, 3(2):30–36.
- Souza, A. L. O. T. d. and Pinto, V. H. S. C. (2020). Toward a definition of cognitive-driven development. In *Proceedings of 36th IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 776–778.
- Sweller, J. (1988). Cognitive load during problem solving: Effects on learning. *Cognitive science*, 12(2):257–285.
- Sweller, J. (2010). Cognitive load theory: Recent theoretical advances.
- Tsantalis, N., Mansouri, M., Eshkevari, L. M., Mazinanian, D., and Dig, D. (2018). Accurate and efficient refactoring detection in commit history. In *Proceedings of the 40th International Conference on Software Engineering, ICSE '18*, pages 483–494, New York, NY, USA. ACM.
- Van Solingen, R., Basili, V., Caldiera, G., and Rombach, H. D. (2002). Goal question metric (gqm) approach. *Encyclopedia of software engineering*.
- Weyuker, E. J. (1988). Evaluating software complexity measures. *IEEE transactions on Software Engineering*, 14(9):1357–1365.
- Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., and Wesslén, A. (2012). *Experimentation in Software Engineering*. Springer Berlin Heidelberg.
- Zuse, H. (2019). *Software complexity: measures and methods*, volume 4. Walter de Gruyter GmbH & Co KG.