


Cognitive-Driven Development: Preliminary Results on Software Refactorings

Victor Hugo Santiago C. Pinto^{1,2}^a, Alberto Luiz Oliveira Tavares de Souza¹,
Yuri Matheus Barboza de Oliveira¹ and Danilo Monteiro Ribeiro¹

¹*Zup Innovation, São Paulo, SP, Brazil*

²*Federal University of Pará (UFPA), Belém, PA, Brazil*

victor.santiago@ufpa.br; {alberto.tavares, yuri.oliveira, danilo.ribeiro}@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, there is a risk that the source code can become worse than the previous version. Developers often lose sight of the business problems being solved and forget the importance of managing complexity. As a result, after refactorings many software parts continue to have low readability levels. Cognitive-Driven Development (CDD) is our recent strategy for reducing cognitive overload during development when improving the code design. This paper provides an experimental study carried out in an industrial context to evaluate refactorings through the use of conventional practices guided by a cognitive constraint for complexity, a principle pointed out by CDD. Eighteen experienced participants took part in this experiment. 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 CDD can guide the restructuring process since it is designed to obtain a coherent and balanced separation of concerns.

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 have shown there is 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 have pointed out that if the refactoring is not guided by a clear purpose, the code can often become worse than the original version, or the changes might fail to have a significant impact (Baqais and Alshayeb, 2020; Alomar, 2019), i.e., the nature of the improvements can be questionable. This effect becomes even more problematic when the refactoring is not guided by a quality metric or the improvements are assessed by developers who have a restricted outlook. Deciding when and what changes

are required, as well as understanding the particular reason for restructuring some code, is still a challenging task for many of them.

Over the years, the continuous expansion and growing complexity of software has led to a good deal of discussions among the software engineering community (Zuse, 2019; Clarke et al., 2016; Weyuker, 1988; Shepperd, 1988). Most researchers are continually seeking better and novel methods for handling the complexity involved in the design and maintenance of software systems. Approaches have been adopted to support code design based on architectural styles and code quality metrics. Nevertheless, there is a lack of practical and clear strategies for changing the way that we develop software for reduced testing and maintenance efforts efficiently.

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 (Duran et al., 2018). Cognitive complexity is a departure from the standard practice of using strictly numeric values to assess software maintainability. It

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

starts with the precedents set by cyclomatic complexity (CYC) (McCabe, 1976), but uses human judgment to assess how the code’s structures should be interpreted. 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 the use of basic control structured and corresponding weights were suggested. Although the cognitive complexity measurements can assist in assessing the understanding of the source code, there is a lack of studies in the literature that works exploring how this strategy can perspective could be applied to reducing complexity in the period ranging from the early stages of development until in the future, where the process incurs costs related to maintenance and testing activities.

This paper extends our recent research study 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, or more specifically, of the recognition of the limited human capacity for dealing with the the expansion of software complexity. The need for new empirically-based studies concerning the generation of high-quality code by means of CDD principles led to an experiment being conducted involving refactoring scenarios. Our main premise was that the definition of cognitive complexity constraints during refactorings could guide the developer to use more types of refactorings and generate a more readable code than conventional refactoring practices without this kind of constraint.

Eighteen software engineers participated in the experiment. Classes from an open-source application called Student Success Portal¹ (SSP) were chosen for the refactorings. We analyzed the developers’ productivity by taking account of their time spent in the restructuring process using conventional practices, both with and without a complexity constraint. Both the original and refactored classes were analyzed and these included 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 since it can achieve better levels of cohesion, readability and separation of concerns.

The remainder of the paper is structured as follows: Section 2 discusses the key characteristics of the CDD; Section 3 outlines the structure of our ex-

perimental study and results. Section 4 describes related work. Finally, Section 5 summarizes the conclusions and sets out future perspectives.

2 BACKGROUND

The continuous expansion of the software scale is one of the main challenges for industry and a current software engineering problem (Zuse, 2019; Pawade et al., 2016). According to the classic statement made by Robert M. Pirsig: *“There’s so much talk about the system. And so little understanding”* (Pirsig, 1999). As there is a greater degree of complexity in software, from the standpoint of people, there is a risk that understanding can be compromised (Kabbur et al., 2020; Briggs and Nunamaker, 2020), since they are unable to follow it in a proportional way.

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

However, elements with low cohesion and an inefficient separation of concerns are still being produced and residing in final releases (Souza and Pinto, 2020). It is a challenging task to limit the frontiers for concerns and reduce their complexities. The goal of software design is to create chunks or slices that fit into a human mind. However, code tangled and spread in addition to the lack of a clear strategy to mitigate them contribute to that the developers can 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 with regard to learning and instructional design. Problems that require a large number of items to be stored in our short-term memory may lead to an excessive cognitive load. According to Sweller (Sweller, 2010), some material has its own complexity and is inherently 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

¹<https://github.com/Jasig/SSP>

able to hold only seven plus or minus two units of information in short-term memory, an important work known as “*Magical Number 7*”.

Efficient programmers write code that people 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 should guide 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 to assist our understanding; this enables developers to set a limit on cognitive complexity and implementation units can be kept within this constraint, even with the expansion of the system** (Souza and Pinto, 2020).

Although the theory behind the CDD is not limited to specific metrics or numerical values to assess code readability, cognitive complexity metrics (Shao and Wang, 2003) were included to introduce the concept of the Intrinsic Complexity Points (ICPs). In our previous work (Souza and Pinto, 2020), our objective was to take note of some basic control structures (BCS) and resources such as ICPs. It should be stressed that other alternatives 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.

With the aid of 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 concern 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 regarded as ICPs.

The coupling with objects provided by a certain framework can be included in the total number of ICPs, but it should be noted that code designed for *crosscutting requirements* is mainly related to infrastructure resources, i.e., it is reused or extended for specific project purposes. Although this kind of code is not trivial, we suggest not considering them for the following reasons: (i) these codes are less often changed than codes related to business logic, and (ii) it is expected that the practitioners have a wide knowledge of the language and framework resources. In previous experiences with real development scenarios, we observed that these kinds of definitions can be used to improve the level of the team. However, this matter still requires further investigation and

empirically-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 was collected² from gitHub using a domain-specific language and infrastructure for mining software repositories called *BOA* (Dyer et al., 2013). These projects were analyzed to identify representative classes and assess the approximate complexity of those found in real projects from industry. Student Success Plan³ (SSP) was the application chosen on the basis of these criteria. The experiment sought to determine if the refactoring guided by a cognitive complexity constraint suggested by CDD, yielded results in code of a higher quality than that produced when only conventional refactoring practices were employed without a limit. This involved carrying out a static code analysis through object-oriented metrics that were applicable to all versions.

3.1 Planning

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

Analyzing resulting pieces of code from refactorings with conventional methods and CDD’s principles with the aim of comparing their quality through object-oriented metrics, regarding the complexity reduction from the standpoint 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.

RQ₁: Is There a Difference in Terms of Quality Metrics for Refactorings Conducted with a Strategy Based on CDD and Conventional Methods?

To answer this question, all the resulting refactorings were evaluated with the aid of a tool for static analysis to collect values for object-oriented metrics.

²<http://boa.cs.iastate.edu/boa/?q=boa/job/public/91048>

³<https://github.com/Jasig/SSP.git>

RQ₂: Is Productivity Different When a Strategy is Adopted based on CDD and Conventional Practices for Refactorings? When addressing this question, we gathered and assessed the time spent to make the refactorings. The time spent includes all the stages involved in the refactoring activity from the code analysis to identify the potential candidate structures until the compilation process.

It should be noted that the single difference when using CDD here is determining the ICPs and imposing a feasible limit to guide the refactorings. We suggested a strategy for the subjects to define this limit and proceed with the refactorings. First, it was decided to calculate the number of ICPs by looking at the refactoring target. Second, an attempt was made to face the growing number of challenges, such as reducing the number of ICPs by 10%, 30%, and even 50% in a class under refactoring. Finally, the subjects were able of obtaining a refactoring cluster keeping the implementation units at a balanced level of complexity.

The planning phase was divided into six parts which 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 Formulation of the Hypothesis

The RQ₁ was formalized into two hypotheses. **Null hypothesis (H₀)**: There is no difference between the conventional refactoring practices and the use of CDD based refactoring, given the quality metrics adopted in this study.

Alternative Hypothesis (H₁): There is a difference between conventional refactoring methods and the use of a strategy based on CDD, from the perspective of the quality metrics employed. These hypotheses can be formalized by Equations 1 and 2:

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

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

Similarly, the RQ₂ was also formalized into two hypotheses. **Null hypothesis (H₀)**: There is no significant difference between the productivity of the developers when account is taken of the time spent for refactorings with conventional practices and a strategy based on CDD, since they are equivalent. **Alternative hypothesis (H₁)**: There is a difference between the time spent on refactorings when conventional practices are employed and a strategy based on

CDD. The hypotheses for the RQ₂ can be formalized by Equations 3 and 4:

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

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

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*): this 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), this 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 the LCOM metric. Finally, LOC (*Lines of code*) counts the lines of code, when ignoring empty lines and comments. Note that these metrics were selected because they are considered important for the company.

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

3.1.4 Selection of Subjects

The subjects were selected according to convenience sampling (Wohlin et al., 2012). Eighteen software engineers who took part in the experiment were working on the development of web projects and they had a degree of knowledge of the Java language.

3.1.5 Experimental Design

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

When separating the subjects into balanced groups, we first asked them to fill out a *Categorization Form* with questions about their experience in areas related to the experiment, i.e., it was a self-evaluation. Based on data that was obtained, we divided them into two blocks: one with eight subjects and another with ten subjects. The first group had to apply conventional practices to improve the readability of the provided classes, while the second group attended a planned training session designed for employing the CDD principles for refactorings.

The *Categorization Form* included questions regarding knowledge about: (i) Object-oriented programming, Java, the 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 projects for “training and learning”, i.e., personal projects aimed at improving technical skills; (ii) software metrics known to them and that can eventually be used to improve code cohesion and the separation of responsibilities; (iii) practices for software refactorings; (iv) practices usually adopted to minimally understand a legacy project and start implementing new features; (v) programming practices and code design; Finally, (vi) testing activities and tools.

Figure 1 describes the results of the application of this form in a grouped bar chart, that only takes account of numeric values (number of books, real and personal projects) for each subject. The main reason to use these elements is that the “time experience” is a relative measurement. For instance, it is likely that a programmer with little time for development but who has attended a higher number of projects can perform better than a person with more time experience and attended a low number of projects. The subjects “S1-S7;S17” belong to the group that are defined as adopting conventional practices without a complexity constraint and the subjects “S8-S16;S18”, CDD for refactorings.

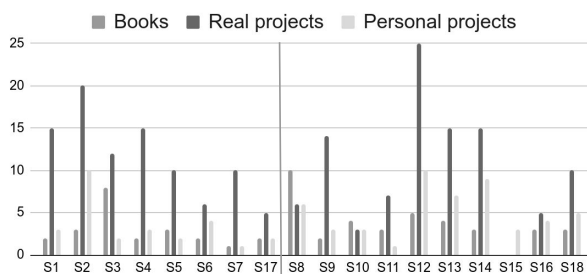


Figure 1: Gathered data with the Categorization Form.

Additional information was obtained to define this separation which can be described as follows, including corresponding percentages of answers.

With regard to **software metrics**, we asked the participants which one they use to generate code thinking about readability, cohesion and separation of concerns. The choices/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 regards the **practices involved in software refactoring**, most subjects underlined 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 concerns more clearly. Another question was targeted at the **practices to minimally understand a legacy project** for including new features or changing any one available. In most cases, the strategies were related to searching for data entry points from the user’s perspective and from there, tracing the execution flows or conducting debugging and analysis of automated tests.

With regard to **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** they were as follows: *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 that described constraints and guidelines to assist them in both the refactoring and the data submission process, as follows:

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

With regard to 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*.

taking into account our cognitive complexity criteria, i.e., a considerable amount of ICPs compared with classes in real-world projects. After the refactorings, the subjects were requested to submit the URLs of their remote repositories, together with the time spent and their perceptions about the experiment, using a web form.

3.2 Operation

Once the experiment had been defined and planned, it was undertaken through the following stages: preparation, operation and validation of the collected data.

3.2.1 Preparation

At this stage, the subjects were committed to carrying out 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 were granted their freedom to withdraw, by signing a *Consent Form*. In addition, other objects were provided:

- *Characterization Form*: A questionnaire in which the subjects assessed their knowledge of the technologies and concepts used in the experiment;
- *Instructions*: A document describing all the stages, including the instructions about the SSP and classes chosen for refactoring;
- *Data Collection Form*: Document to be filled in by the participants to record the start and finishing time of each activity during the experiment.

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

3.3 Data Analysis

This section examines our findings. The analysis is divided into two areas: (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 the statistical methods were applied. There is a risk that incorrect data sets can be obtained as the result of some error or the presence of outliers, which are data values much higher or much lower than the remaining data.

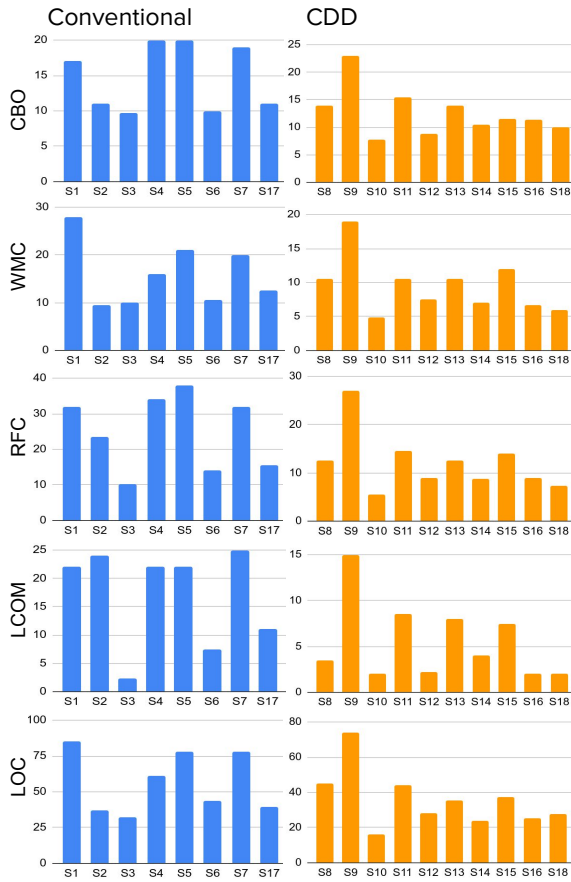
The metrics adopted in this study have different scales and when taking note of the “refactoring clusters” we decided to be conservative and analyze all the gathered data, in an individual way per metric. Refactoring clusters can be understood as a) the set of changes produced during refactoring; b)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

When clarifying descriptive statistics and making comparisons, it is important to keep certain values in mind. The following values were obtained for the original version of the Class *JournalEntryServiceImpl*: *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 include these kinds of data for *JournalEntryServiceImpl* and *EarlyAlertServiceImpl* refactoring clusters, respectively. The reason why there is a difference between the number of subjects that attended both refactorings is that not all the subjects finished the activities during the time of four hours and thirty minutes. With regard to the averages, the clusters that were generated following the cognitive complexity constraint (just called CDD) tend to keep more balanced and lower levels for the metrics than refactorings without this 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 regards the same metrics when conventional practices were applied without a complexity constraint, it should be noted that the limits had higher values in the intervals than was the case with the aforementioned data.

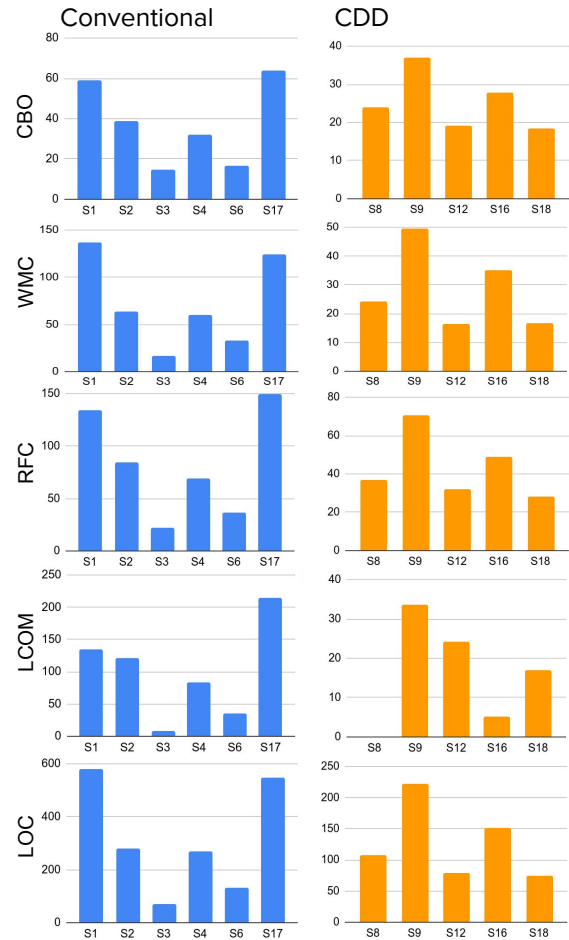
Figure 4 displays box plots for each metric per samples. The box plots are “A-E” for *JournalEntryServiceImpl* and “F-J” *EarlyAlertServiceImpl*, *CBO*, *WMC*, *RFC*, *LCOM* and *LOC*, respectively. Note that each interior of a box plot refers to a specific sample,

Figure 2: *JournalEntryServiceImpl* refactoring clusters.

“Conventional”, where the subjects strictly used the conventional practices for refactoring or “CDD”, using the same practices but guided by a cognitive complexity constraint.

In addition, Table 1 shows the types of refactoring detected using *Refactoring Miner* (Tsantalis et al., 2018), an 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 that took account of all the changes brought about 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 the remaining developers.

Figure 5 shows two box plots based on the time spent by all the subjects (grouped by Time (min)). The boxes inside each graph refer to specific samples, Conventional and CDD. In case of *JournalEntryServiceImpl*: (A) the average time spent using CDD was higher in the sample where the subjects were worried about the cognitive complexity constraint, which is likely to be the main reason for this difference. As regards the *EarlyAlertServiceImpl* (B), some subjects were guided by a constraint for separating the respon-

Figure 3: *EarlyAlertServiceImpl* refactoring clusters.

sibilities into more implementation units, but the remaining subjects that also adopted this constraint only kept their focus on the target class, indicated in the experiment, which meant, they spent less time completing their activities. As a result, 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 departed from linearity. This involved conducting the Shapiro-Wilk normality test to check if the samples had a normal (ND) or non-normal distribution (NND).

Table 2 shows the results of testing for normality for all the samples, i.e., for *JournalEntryServiceImpl* and *EarlyAlertServiceImpl* refactorings clusters, respectively. The results refer to the analysis for each metric, including the corresponding samples. For in-

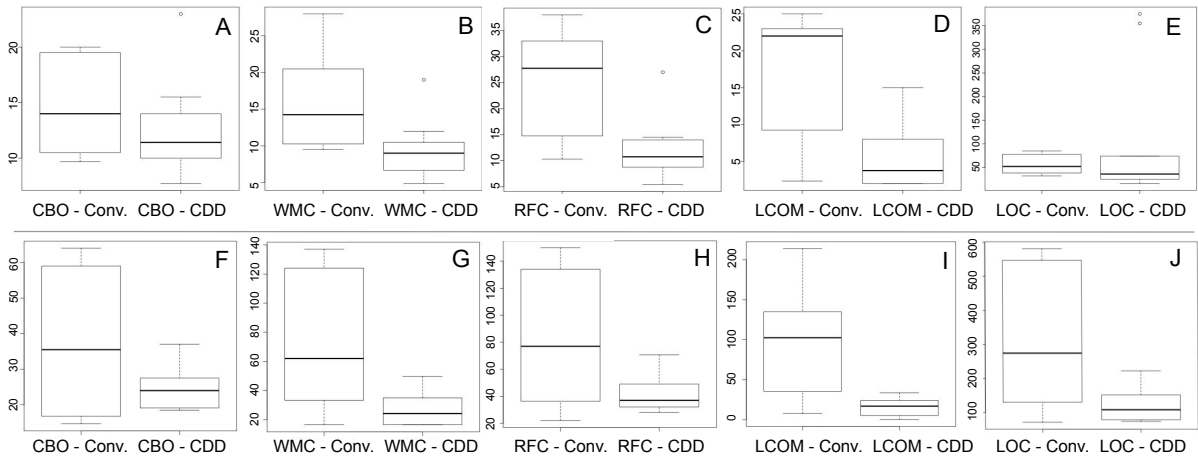
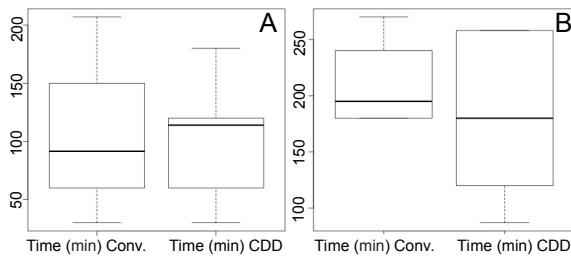
Figure 4: Box plots for *JournalEntryServiceImpl* (A-E) and *EarlyAlertServiceImpl* (F-J) refactoring clusters.

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										3						
	S4	3											1							20					
	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					1					
	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).

stance, taking into account the *CBO* metric from the “CDD” sample, we do not reject the hypothesis that the data are from a normally distributed population, although the “Conventional” sample follows a non-

normal distribution. Variance testing was performed for *WMC* and *LOC* and took note of the *JournalEntryServiceImpl* refactoring clusters and *CBO* for the samples related to the *EarlyAlertServiceImpl*. In the case of these metrics, the *p-values* were 0.1841, 0.4392 and 0.6918 (based on $\alpha = 0.05$, respectively).

Unpaired Two-Samples T-test (or unpaired t-test) can be used to compare the means of two unrelated groups of samples. This kind of statistical testing was conducted for *WMC* and *LOC* by taking note of the *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 these metrics. However, there is no considerable difference between the mean averages for *CBO* of the *EarlyAlertServiceImpl* refac-

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

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

toring clusters.

The 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 this kind of testing for *CBO*, *RFC* and *LCOM* taking account the *JournalEntryServiceImpl* refactoring clusters. As a result, the $p\text{-values}$ were 0.4763, 0.007561 and 0.009632, respectively. Therefore, it is impossible to reject the null hypothesis for *CBO*, even though in the case of *RFC* and *LCOM* the null hypothesis can be rejected. As regards the *WMC*, *RFC*, *LCOM* and *LOC* for *EarlyAlertServiceImpl*, the $p\text{-values}$ were as follows: 0.2615, 0.6304, 0.1488 and 0.3358, which means, it is not possible to reject the null hypothesis for these samples.

Summarizing the results, when comparing the samples for the *JournalEntryServiceImpl* refactoring clusters, we can conclude that there was no statistical difference for *CBO*, although 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 highly significant 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 the normality tests, we cannot reject the hypothesis that all the 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. With regard to the variance testing, we obtained $p\text{-value} = 0.3601$ for *JournalEntryServiceImpl* 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., the productivity in terms of time was not affected by employing a complexity constraint.

3.4 Threats to Validity

Internal Validity. Level of Experience of Subjects: One can argue that the heterogeneous knowledge of the subjects could have affected the collected data. To overcome this threat, the participants were divided into two-balanced blocks that took account of their level of experience.

During the training, the subjects that had to apply the cognitive complexity constraint attended a training session on how to use this limit to guide the refactoring process. Thus, they adopted conventional practices to restructure the code like the other group but following such limit;

Productivity under Evaluation: the results may have been affected because the subjects often tend to think they are being evaluated during an experiment. We attempted to overcome this problem by explaining to the subjects that no one was being evaluated and their participation would be treated as anonymous;

Validity by Construction. Hypothesis Expectations: the subjects already knew the researchers, a point which is reflected in one of our hypotheses. This issue could have affected the collected data and caused the experiment to be less impartial. Impartiality was kept by insisting that the participants had to keep a steady pace during the whole of the study.

External Validity. Interaction between Configuration and Treatment: it is possible that the exercises carried out 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., the complexity of the applications and the fact that the researchers have contact with real-world projects of the company.

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

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

4 RELATED WORK

Duran et al. (Duran et al., 2018) established a framework for the 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 factors when we are mentally manipulating a program. The contribution made by this work is to concentrate on the cognitive complexity present in program designs rather than on how the source code could be developed or restructured from this perspective, as suggested by CDD.

Gonçalves et al. (Gonçalves et al., 2019) provided a classification of cognitive loads in software engineering. Recent advances have been made in the areas of the programming tasks, machine learning techniques to identify a programmer's level of difficulty and their code-level comprehensibility. CDD can be regarded as a complementary design scheme for tackling the increase in cognitive complexity regardless of the software size.

Cognitive Complexity is a significant metric to assess code understandability. According to the systematic literature review conducted by Muñoz Barón et al. (Muñoz Barón et al., 2020) the Cognitive Complexity positively correlates with comprehension time and the subjective ratings of understandability. This underlines the importance of an intrinsic complexity constraint for the implementation units.

5 CONCLUSION

The human factor imposes several challenges in software development. For instance, the developers' varying level of experience or involvement in team restructuring during the software process, can affect 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 being produced and reside in final releases, even when developers widely familiar with certain programming language and development stacks. 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 only hold 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 problem of software complexity, by reducing the cognitive overload.

The main focus of this work was to assess the effects of adopting a complexity constraint on refactorings and on the productivity of developers. Although we do not have conclusive results, refactorings using conventional practices guided by a complexity limit were better evaluated when they were compared with the refactoring clusters generated without this kind of 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 or without complexity constraint. A package containing the tools, materials and more details about the experimental stages is available at <http://bit.ly/3mElnp5>.

As future investigations, we intend to explore the following factors: (i) defining an automated refactoring strategy by means of search-based refactoring and cognitive complexity constraints and (ii) carrying out new empirical-based studies in an industrial context to evaluate restructured projects with CDD principles, by exploring the number of faults and understanding development in the medium and long term.

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.
- Gonçalves, 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.
- Pawade, 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.