To What Extent Cognitive-Driven Development Improves Code Readability?

Leonardo Barbosa
leonardopfb@gmail.com
UFPA
Belém, PA, Brazil

Victor Hugo Santiago
victor.santiago@ufpa.br
UFPA
Belém, PA, Brazil

Alberto Luiz Oliveira Tavares de Souza
alberto.tavares@zup.com.br
Zup Innovation
Belém, PA, Brazil

Gustavo Pinto
gustavo.pinto@zup.com.br
UFPA & Zup Innovation
Belém, PA, Brazil

ABSTRACT

Cognitive-Driven Development (CDD) is a coding design technique that aims to reduce the cognitive effort that developers place in understanding a given code unit (e.g., a class). By following CDD design practices, it is expected that the coding units to be smaller, and, thus, easier to maintain and evolve. However, it is so far unknown whether these smaller code units coded using CDD standards are, indeed, easier to understand. In this work we aim to assess to what extent CDD improves code readability. To achieve this goal, we conducted a two-phase study. We start by inviting professional software developers to vote (and justify their rationale) on the most readable pair of code snippets (from a set of 10 pairs); one of the pairs was coded using CDD practices. We received 133 answers. In the second phase, we applied the state-of-the art readability model on the 10-pairs of CDD-guided refactorings. We observed some conflicting results. On the one hand, developers perceived that seven (out of 10) CDD-guided refactorings were more readable than their counterparts; for two other CDD-guided refactorings, developers were undecided, while only in one of the CDD-guided refactorings, developers preferred the original code snippet. On the other hand, we noticed that only one CDD-guided refactorings have better performance readability, assessed by state-of-the-art readability models. Our results provide initial evidence that CDD could be an interesting approach for software design.

CCS CONCEPTS

• **Software and its engineering → Designing software**.

KEYWORDS

readability, cognitive-driven development, cognitive load

1 INTRODUCTION

Cognitive-Driven Development (or CDD for short) is a novel coding design technique that aims to reduce code complexity by limiting the number of language constructs that could be used at once in a given source code file [4, 18, 19]. CDD aims at developing strategies that reduce the developer cognitive load.

Instead of being based on anecdotal experience, CDD has its roots in two well-known psychology theories: the Magical Number Seven Theory [13] and the Cognitive Load Theory (CLT) [22, 23]. In Miller's work [13] known as "Magical Number Seven", it is explained that we probably have a hard limitation in simultaneous processing information. Experimental studies.[13] have suggested that humans generally hold only seven plus or minus two information units in short-term memory. CLT [22, 23], on the other hand, explains that any material has an intrinsic complexity depending on the amount and arrangement of the elements that compose it. CLT is an instructional design theory that reflects our "cognitive architecture", an important aspect to presenting information at a pace that learners can fully understand. According to Sweller [22], knowing the number of information elements and their interactivity is crucial to support learners.

In terms of source code, it may not be a surprise that a high number of information elements, e.g., control structures and language constructs, can harden ones understanding. CDD focuses on directing developers to create and maintain code units respecting the limited human cognition capacity. Therefore, the more elements are concentrated in the same code unit, the greater the effort developers will have to place to understand it. While the practice of CDD could indeed ease software maintenance (since CDD-guided source code uses less code elements), little is known whether the code that was developed using CDD has also better readability. Therefore, the goal of this work is to provide answers to the following research questions:

RQ1. Do CDD-guided refactorings improves code readability, according to professional software developers?

RQ2. Do CDD-guided refactorings improves code readability, according to the state-of-the-art readability model?

To answer these questions, we performed a two-phase study. To answer **RQ1**, in the first phase of this work we started by surveying 133 professional developers. We present to them 10 pairs of code snippets (pre and pos CDD-guided refactorings) and ask our subjects to vote on which code snippet they believe is more readable. We also asked them to offer their rationale behind their vote. To answer **RQ2**, in the second phase, we leverage the state-of-the-art readability model proposed by Posnett et al. [20]. For each pair of code, we studied whether the CDD-guided refactorings might have, indeed, improved code readability. Among the 10 pairs of code snippets evaluated, we observed that in seven of them, our participants concur that the CDD-guided refactorings improved readability (in two pairs the results were balanced, while in just one the participants preferred the original code). However, when performing the state-of-the-art readability model, we noticed a different figure. In this case, just one of the CDD-guided refactorings was considered as more readable, according to the model. This work makes the following contributions.

(1) We curated a set of 10 CDD-guided refactorings (along with their original versions), created by professional software developers;

(2) We conducted a survey with 133 professional software developers to assess their perception about the 10 pairs of code snippets.

(3) We applied the state-of-the-art readability model in the 10 pairs of code snippets.

(4) We made available all data used in this research, to facilitate further replication.

2 HOW DOES CDD WORK?

Complexity is part of the software and CDD recognizes that the intrinsic load effects from code units are different for people. Suppose we have two developers analyzing the same program. They will likely disagree on the difficulty of understanding. Nonetheless, when code needs to evolve, the difference in observed complexity will likely impact the maintenance cost. CDD aims to create an unified concept for intrinsic complexity for the code units.

The first step of the CDD approach is to define the "Intrinsic Complexity Points" (ICPs) [4], which are the elements inside the code that can affect the understanding according to their usage frequency. Example of such elements are for instance: code branches (if-else, loops, when, switch-case, do-while, try-catch and etc.), functions as an argument, contextual coupling (i.e., coupling with specific project classes), and inheritance. CDD is not limited to those code elements, though. Indeed, developers can include any other code elements, from SQL instructions, to annotations, to assertions, etc. Any programming construct that the team consider relevant can be considered as ICP.

After the selection of ICPs, a constraint should be defined for all code units. CDD can contribute to a practical perspective of what makes understanding compromised for a given context. For example, how many if statements should I have in a given code unit? As everyone is involved in creating the code, a complexity limit can be defined collaboratively for the code units regardless of the specialization degree of developers. Therefore, development teams can adopt the CDD method according to their interest, the project nature and experience, i.e., the cognitive complexity limit needs to be customized; this is also true for the elements included in this constraint. The main reward of this strategy is that the code units can be kept under a limit even with the exponential growth of the software complexity.

3 RESEARCH METHODOLOGY

This work was built upon the work of Pinto et al.[18] and Santos and Gerosa [5]. Regarding the work of Pinto et al.[18], we relied on the refactoring produced by developers from industry that took part of the experiment (more in Section 3.1). After cleaning and filtering representative refactoring examples, we adopted the survey approach presented by Santos and Gerosa [5] to gather developers' opinion on which code snippets have better readability (more in Section 3.2). We though extend the Santos and Gerosa work by applying the state-of-the-art readability model, proposed by Posnett et al. [20] (more in Section 3.3).

3.1 Curating refactoring examples

Pinto et al. [18] carried out an experimental study to determine whether refactoring using cognitive driven constraints leads to better software than traditional style refactoring. The authors have conducted the experiment in an industrial setup and evaluated software quality via software quality metrics. Nonetheless, since the industry participants performed several scattered changes, the authors were unable to understand the impact of isolated CDD-guided refactorings on code readability. We started by selecting refactorings that were produced from this research. The authors invited 18 industry participants to refactor three classes of two well-known Java projects, namely SSP[1] and feign[2]. The participants were divided in two groups: the first group performed their refactorings using their own intuition, whereas the second group followed a disciplined CDD approach. To carry out their work, both groups should meet the following requirements:

(1) The original project's packages should not be modified;

(2) New packages should not be created during refactorings;

(3) Automated tests should continue working without changes, and;

(4) Public, protected, or package private methods from original class should not be modified.

Additionally, the participants in the CDD group had to adhere with the following set of ICPs: code branches (i.e., if-else, loops, when, switch/case, do-while blocks), exception handling (i.e., try-catch blocks), functions as an argument, and contextual coupling. It is important to highlight that the participants of the CDD group were free to define a feasible complexity constraint for the code units based on possible ICPs. According to the authors, a single recommendation to define this constraint is that it could be equivalent to twice the number of ICPs chosen for accounting.

This group was encouraged to follow a progressive reduction strategy of ICPs. For instance, such participants were trained to manually identify the aforementioned set of ICPs in the classes to be

---

[1]https://github.com/Jasig/SSP
[2]https://github.com/OpenFeign/feign

restructured. After that, the participants had to refactor the classes in order to reduce the total number of ICPs to satisfy the constraint previously defined. According to the authors, their hypothesis was that the CDD-guided refactorings could lead to more quality in terms of static metrics. The authors though used CK metrics [3] (namely CBO, LCOM, RFC and WMC) as a proxy of readability.

In this work we extend the work of Pinto et al. [18] by focusing on understanding to readability of isolated CDD-guided refactorings. We asked the authors their dataset, which they gently provided. The tool named Meld [3] was adopted to compare files and visualize the changes performed by the participants.

When we started visualizing the code changes, we quickly noticed that not all code changes were performed following CDD guidelines. For example, some changes were targeted to streamline methods and remove code duplication. We then focused our search on "isolated CDD-guided refactorings", that is, code changes that we could assure that were guided by CDD. We used the following approach to identify the "isolated CDD-guided refactorings":

(1) We searched for code changes that could match with the CDD requirements employed. For example, if the changes tried to reduce the `if`, `for` or `try-catch` statements, chances are that they were guided by CDD;

(2) We sought to observe indications that an intrinsic complexity constraint was defined to guide the refactorings. For example, when the target classes for refactorings were changed and the new classes were created following a clear definition for a complexity limit considering the basic control structures aforementioned;

(3) We focused on code changes that adhere with the seminal refactoring definition: code changes that preserve external behavior [16]. If the refactored source code introduced additional code elements that could potentially change the program's behavior, we discarded that code change.

This process was independently performed by two researchers and was revised by a third researcher. This process took about six weeks. We limited the number of CDD-guided refactorings to 10 pairs, 20 at total (10 before and 10 after the refactoring). This number is similar to the number of examples used in the work of Santos and Gerosa [5], which chose 11 examples.

The selected pairs have are fairly small (on average, 25 lines of code); more at Table 3. This is aligned with the goals of our work. For our **RQ1**, we intended to ask developers about their perceptions of each refactoring example — and having a high number of examples would potentially tire the respondent (which in turn could lead to survey dropouts). For **RQ2**, our applied model have better performance for small code snippets (the authors conducted their work with code snippets ranging from 4 to 11 lines of code; code snippets with 200+ lines of code were all ranked as less readable.

3.2　Designing and Deploying the survey

Our survey design was inspired by the work of Santos and Gerosa [5]. In their work, the authors asked developers about their perceptions regarding 10 pairs of code snippets. Since the authors were interested in visually assessing the pairs of code examples, they were not not able to rely on tools such as Google Form or Survey Monkey,

[3]https://meldmerge.org/

because they do not offer such visual features. Instead, they opted to built their own survey tool. The survey participants were mostly composed by students (55 respondents), in addition to seven professional developers. In the next, we discuss how our work differs from theirs. Also, our survey was based on the recommendations of Kitchenham et al.[10], we followed the phases prescribed: planning, creating the questionnaire, defining the target audience, evaluating, conducting the survey, and analyzing the results. We discuss each one of these phases next.

Planing. We identified 12 representative refactoring examples. However, as aforementioned, we had to discard two examples to avoid occupying too much the participants. Therefore, we group the refactoring examples into categories and discarded those from most common categories. We ended up with 10 pairs of code snippets.

Moreover, we decided not to mention that the refactorings were CDD-guided since this could lead to biases for (or against) CDD. Instead, we just asked the participant about their preference without further information about the pairs of examples. We also decided to randomly alternate the sides of the examples, otherwise participants could naively favor one of the sides (left or right). Finally, since readability is a subjective concept and that different factors can influence code readability, before presenting to developers the pairs of examples, we present to them our definition of readability: "Readability is a human judgment about the cognitive effort required to understand a source code. Visual features such as spacing, indentation, capitalization, function names, language constructs, among others, can affect the readability and, thus, the understanding of a code snippet.". This definition is inspired by the works of Oliveira et al [15], which describes that readability is "what makes a program easier or harder to read and apprehend by developers".

Creating the questionnaire. In this work, we relied on TypeForm, which is an online survey service that allows dynamic forms. Using TypeForm, we were able to present developers with two images and ask them their preference. We used this feature to present our pairs of examples, as Figure 1 shows. Knowing that syntax highlight could improve code comprehension [9], different than Santos and Gerosa [5], we used the `carbon.now.sh` tool to create code snippets with syntax highlight in dark mode (which offer better contrasts to the figures). After the participant vote of their preferable code snippet, we asked their rationale for this decision.

Our survey had 27 questions (25 were required, 12 were open). We asked five demographic questions (Technical Profile {Developer, QA, Manager, etc}, Technical Level {Novice, Intermediate, Senior, Principal, etc}, Age { <20y, 21–30y, 31–40y, 41–50y, >50y }, Years of experience { <1y, 1–5y, 6–10y, 10–15y, >15y }, Experience with Java {1–10}). Since we had to separate the questions regarding the refactorings preference from the developers rationale for their choice, we ended up with 20 questions for the 10 refactorings. In the end of the survey, we had two final open questions: one whether the participant had any questions/comments regarding our study and the other if she is interested in participating in a follow up interview. The estimated time to complete the survey was 10–15 minutes.

Figure 1: Most readable snippet selection screen.

Table 1: Participants roles & programming experience.

Role	Population	<1y	1–5y	6–10y	10–15y	>15y
Development	125	22	32	34	19	18
Management	6	1	0	2	1	2
Testing & QA	1	0	0	1	0	0
Infrastructure	1	0	0	1	0	0
	133	23	32	38	20	20

Evaluating the survey. Before deploying the actual survey, we conducted a pilot with professional software developers and researchers. These developers and researchers were close personal contact to the authors of this work. The goal of this pilot survey was to assess the clarity of the questions and the quality of the refactorings examples. The participants of this pilot survey were instructed that their feedback regarding the questions and the code snippets was more important than the answers themselves. After a period of one week, we received 8 answers to the survey and a few comments requesting clarifications. We applied the suggestions and removed the 8 answers from the database.

Conducting the survey. After incorporating improvements suggested in the pilot survey, we administered the actual survey. To do this, we created two online questionnaires. The first questionnaire was deployed at a large Brazilian software producing company, while the other questionnaire was shared in two social platforms: Twitter and LinkedIn.

For the *first questionnaire*, one of the author work as a researcher in the company and had access to the company communication's channels. However, instead of sending the questionnaire to all ~3.5k company's employees (which is against the company privacy culture), we shared the link of the questionnaire in the company's general Google space (similar to a Slack channel). Periodic reminders were sent at that same Google space.

The *second questionnaire* that was shared in social networks had exactly the same questions and options of the first one, but a different URL address. We decided to have two questionnaires because we had to report the perception of the company's employees. Given the nature of the approach we used to invite participants, we were unable to track the number of participants that received the questionnaire. However, Typeform tracks the number of access to the questionnaire. For instance, (255+328) participants opened the questionnaire and (210+201) started filling the questionnaire. After a period of two weeks, we received a total of 133 responses, 73 for the first questionnaire (34.8% of completion rate), and 60 for the second one (30% of completion rate). For both surveys, participation was voluntary.

Target audience. The target audience of our work are professional software developers. Table 1 summarizes our participants demographics. Regarding their seniority, 9% consider themselves as novice developers, 45% are intermediate developers, and 33% are senior developers. The remaining 13% play different roles, such as Tech lead and C-level. As for their ages, 49% have between 21 and 30 years old, 38% have between 31 and 40 years old, 11% have between 41 and 50 years old, and only 1.4% have more than 50 years old. As for their software development experience, 22% have up to 1 year, 29% have between 2 to 5 years of experience, 23.3% have between 6 and 10 years, another 12% have between 11 to 15 years, and 14% have more than 15 years of software development experience. In a scale from 0 to 10, where 0 means no knowledge at all and 10 means being an expert, our participants averaged 7.8 in Java programming (Figure 2).
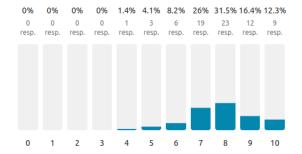

Figure 2: Java Experience.

Analyzing the results. We performed both quantitative and qualitative analysis methods. For the quantitative method, we used the Chi-Square (χ^2) statistical test. χ^2 is a nonparametric test (i.e., it does not require normality). The test indicates whether our hypothesis holds or not. We defined the α level to the conventional level of 0.05. For the qualitative method, one of the authors followed qualitative coding techniques to categorize the respondents' perception. This approach was performed by one author and revised by another author. The most interesting observations are discussed throughout Section 4 along with quotes from the survey. Among similar observations, we chose to quote only the one we considered the most representative for each case.

3.3 Applying Posnett et al. readability model

Although it is consensus that code readability is a subjective matter, in the last decades, researchers have proposed several metrics, tools, and models to assess code readability [1, 2, 8, 12, 20, 21].

In this work we leverage the model proposed by Posnett and colleagues [20]. This model is a simplified mix of two other approaches: the Buse model [1] and the Hastead metrics [7]. Posnett

and colleagues proposed a simplification of these two other works that consider only three variables: *lines of code*, *volume*, and *entropy*. Volume and Entropy are two similar metrics. The author differs *volume* and *entropy* as the follows: "*Entropy calculations depend on the relative distribution of the tokens/characters in the code body under consideration, with uniform distributions giving the highest entropy, and highly skewed distributions yielding lower entropy; whereas volume attempts to determine the number of bits needed to represent all operators and operands multiplied by the total number of tokens*."

To calculate the Posnett [20] model, we used the Java-based implementation provided by Mendonça and colleagues [11]. To run the tool, we have to create the files for each code snippet (20 files at total). According to the tool documentation, "*The content of the file should be just a Java method body (and not an entire Java class)*". If the source code contained a `class` definition, the tool raised an exception and finished execution. This limitation might be due to the fact that the seminal work of Posnett and colleagues [20] did not include class definition in their evaluated code snippets. Therefore, we could only calculate Posnett et al. readability model for methods (more details in Section 4).

3.4 Dataset availability

In order to foster replications of our work, we made available all data used in this research online at https://bit.ly/3LG0eIm. It includes 1) the curated sample of 10 pairs of code snippets, 2) the online questionnaire, and 3) the answers anonymized. We encourage others to replicate our work.

4 RESULTS

We organize our results in terms of the research questions.

RQ1. Do CDD-guided refactorings improves code readability, according to professional software developers?

Table 2 presents the overview of our survey results. As one could observe, for the majority of pairs, participants agreed that the CDD-guided refactorings were more readable. In particular, P3, P10, P4, P2, and P6 were fairly well-voted (they acquired 93%, 92%, 89%, 86%, and 80% of the votes, respectively). For two of the pairs (P1 and P7), we were unable to derive consensus. Finally, for only P5, participants voted against CDD. The results for and against CDD were statistically significant. Given this scenario, we were unable to reject null hypothesis.

To better understand the reasons behind developers intuition of readability, after presenting each pair, we asked their rationale for their votes. We now present the results of the analysis of these comments. We summarize their opinions for each pair next.

P1: Encapsulating error handling (50% / 50%). In this example, the original code had a `try-catch` block handling the `SendFailedException` exception. The CDD-refactored version of this code extracted the exception handling responsibility and moved the `try-catch` block into a new method called `sendMessageToAdvisor`. Figure 3 shows this pair. We noted a remarkable similarity in the votes; 66 participants preferred the original code, whereas other 66 preferred CDD-guided version. When analyzing the comments of this pair,

```
public void method() {
    final EarlyAlert saved = getDao().save(earlyAlert);
    try {
        sendMessageToAdvisor(saved, earlyAlert.getEmailCC());
        } catch (final SendFailedException e) {
        LOGGER.warn("Could not send Early Alert message to
            advisor.", e);
        throw new ValidationException("Early Alert was NOT
            created.", e);
    }
}
```

$$\Downarrow$$

```
public void method() {
  final EarlyAlert saved = getDao().save(earlyAlert);

  sendMessageToAdvisorService.sendMessageToAdvisor(
    saved, earlyAlert.getEmailCC(), LOGGER);
}
```

Figure 3: P1: Encapsulating error handling

we noticed that developers that favored the original code have a certain habit with this coding practice, as one respondent argued: "*Logging with* `try-catch` *is so common that I didn't even need to read the code to understand the purpose.*". Another participant mentioned that "*there is more visibility of the actions performed in the code, what is being executed is more explicit.*" On the other hand, the participants that favored the CDD-refactored version pointed out that the refactored code is closed to plain English, as one respondent mentioned: "*the code is shorter and with a language closer to the natural one*". Other respondents mentioned that the code has a clear purpose: "*the removal of* `try-catch` *avoids distraction from the most important part of the code*", and "*Isolating error handling within a method gives me a partial view, according to the responsibility of each method*". The decoupling leaves the most important part in the main code and therefore easier to understand, in addition to the method created having a name well suited to its purpose makes it easier to understand.

P2: Encapsulating business rules (14% / 86%). This pair is somehow similar to P1, although com rather different developers' perception. The original version had three `if`s statements testing conditions and raising an exception message if the conditions were met. The refactored version encapsulated each condition into a specific method, abstracting away the business logic. There were 19 votes in favor of the original version. One of these respondents mentioned that it is important to known which exception could be raise, while another respondent did not see any value in creating an specific validation class. On the other hand, the participants that favored the CDD version pointed out that the refactored version is simpler and intuitive. For instance, one respondent highlighted that *the method signature is clear and it makes it easy to understand its behavior*", whereas another respondent said that "*The second snippet, besides being leaner, may represent an opportunity for code reuse*". Indeed, several participants that the CDD version creates room for code reuse. Participants also highlighted the importance of the method

Table 2: Received Votes per Code Snippet Pair.

Pairs	Short Description	Original	CDD-guided	% Original	% CDD-guided	χ^2
P1	Encapsulating error handling	67	66	50%	50%	0.9309
P2	Encapsulating business rules	19	114	14%	86%	0.0000
P3	Concatenating logical expressions	9	124	7%	93%	0.0000
P4	Extracting class	14	119	11%	89%	0.0000
P5	Listing all imports	45	88	34%	66%	0.0001
P6	Encapsulating business rules	27	106	20%	80%	0.0000
P7	Asserting conditions	60	73	45%	55%	0.2596
P8	Functional checking style	95	38	71%	29%	0.0000
P9	Extracting class	44	89	33%	67%	0.0000
P10	Encapsulating for loops	10	123	8%	92%	0.0000

```java
public EarlyAlert create(@NotNull final EarlyAlert
    earlyAlert) {
  if (earlyAlert.getPerson() == null) {
    throw new ValidationException("Person is missing.");
  }

  if (earlyAlert.getCreatedBy() == null) {
    throw new ValidationException("CreeatedBy is missing.");
  }

  if (earlyAlert.getCampus() == null) {
    throw new ValidationException("Campus is missing.");
  }
}
```
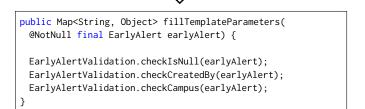
$$\Downarrow$$

```java
public Map<String, Object> fillTemplateParameters(
  @NotNull final EarlyAlert earlyAlert) {

  EarlyAlertValidation.checkIsNull(earlyAlert);
  EarlyAlertValidation.checkCreatedBy(earlyAlert);
  EarlyAlertValidation.checkCampus(earlyAlert);
}
```

Figure 4: P2: Encapsulating business rules

signatures to be consistent with their operation, since it makes the code much simpler and intuitive.

P3: Concatenating logical expressions (7% / 93%). In this practice, we present a method with four if statements (three nested ones). In the CDD-version, the logical expressions of these four ifs were grouped into a single if statement. A total of 124 participants (93%) voted for the CDD version as the most readable one. One CDD voter commented that this was the "*Classic example of cyclomatic complexity. I couldn't understand the example with several nested if's*". Other respondents also commented that the excessive indentation of the nested ifs hinders readability. However, one participant that voted in the original version brought an interesting perspective: "*it is easier to debug the code when ifs are separated [...] It also eases the use of InteliĴ features such as 'evaluate expression'*".

P4: Extracting class (11% / 89%). In this practice, in the original code, we presented the implementation of a Comparator abstract class within the body method. On the other hand, in the CDD version, we introduced a new class, MessageParamsComparator, that implemented the Comparator behavior. The new class is accessed by using MessageParamsComparator::compare in the original method. The vast majority of the voters (89%) favored the CDD version. One CDD voter commented that "*The separation into a new method, with a suitable name, makes the code more consistent and easier to understand*". Another respondent commented that the CDD version "*Decreased cognitive complexity after extracting business logic*", which is in sharp alignment with the CDD purpose. Still, one CDD voter also highlighted the potential for code reuse: "*Besides being more readable, it can be reusable*". Regarding the comments in favor of the original code, one respondent mentioned that "*It's the way I'm most used to do. But I consider it a 'less elegant' way*". Other respondent thought that "*the first option is clearer for those who don't know lambda functions*".

P5: Listing all imports (34% / 66%). Here we compared a code snippet in which the imports are implicit (when we used (*) wildcard to hidden classes of the same package) and another (the CDD version) where the imports are explicit (when we listed all imports). This question had some interesting comments. Most of voters favoring the original code believed that the implicit imports bring better readability because it uses fewer lines of code and, then, it makes it easier to identify groups of imports. In this regard, respondent mentioned that "*The shorter list is clearly more readable, although it might be importing classes that will not be used.*". On the other hand, developers that advocate in favor of the CDD design argue that it is important to understand what is being imported, as one respondent said: "*Despite making the files more extensive, it is preferable to import classes individually, since it eases their location and understanding. Also, it does not import unnecessary things to the code*". Some developers also mentioned that the explicit approach eases code maintenance, since it reduces the chances of importing the wrong class.

P6: Encapsulating business rules (34% / 66%). This is the same kind of practice reported in P2. The original code had two conditionals to check if an array is not empty and, if not, inserts several elements at once in the array, using the addAll method.

The CDD approach encapsulated the two `if` statements into two methods. Similar to the results of P2, in this practice, the majority of the respondents (106 of them) also preferred the CDD version. The reasons for (and against) CDD were also similar, as the ones provided in P2. For instance, those that favored the CDD approach commented that "*The reduction (or even encapsulation) of* `if`*s improves a lot the code readability*" and "*The [newly introduced] method's name is self-explanatory*". One final interesting observation was regarding the negation in the condition (i.e., `if (!requestTemplate.queries().isEmpty())`), as one participant highlighted: "*The negation used in the original condition increases cognitive complexity and hinders the code's understanding*".

```java
public Request request() {
  if (!this.resolved) {
    throw new IllegalStateException("template has not been
        resolved.");
  }
  return Request.create(method, url(), headers(), body,
      this);
}
```

⇓

```java
public Request request() {
  Asserts.booleanStateMustBeTrue(this.resolved, "template
      has not been resolved.");
  return Request.create(method, url(), headers(), body,
      this);
}
```

Figure 5: P7: Asserting conditions

P7: Asserting conditions (45% / 55%). This practice is somewhat similar to P2 and P6, in the sense that the `if` statement is encapsulated, but in here the CDD-approach used a native Java `Assert` method to perform the comparison and throw the exception (if needed). Developers were undecided, with a slight advantage for the code refactored with CDD. Regarding the comments in favor of the original code, one developer mentioned that the "`Assert` *methods are more conventional to testing code*". Another one said that "*The if is simple, I don't see the need to extract it into a separate method*". On the other hand, those that favored CDD commented, once again, on the reuse opportunity ("*Code reuse is guaranteed*". Another respondent complemented that "*The assertion makes explicit the expected condition (and what will happen if it isn't met).*".

P8: Functional checking style (71% / 29%). Here we present a validation using a ternary operator and the CDD-version that followed a functional style using `Optional`. This was the only case in which the original code was vastly preferred. Those that voted for CDD mentioned that ("*Reflects more the OO paradigm*") Regarding the original code, developers commented that the `map` function was brought too much complexity ("*The ternary is an easy resource and there was no complex logic to justify a* `.map`"). One respondent also mentioned that the ternary operator was of better use "*than chaining* `if`*s statements*".

```java
public String method() {
  return (method != null) ? method.name() : null;
}
```

⇓

```java
public String method() {
  return Optional.ofNullable(method)
              .map(HttpMethod::name)
              .orElse(null);
}
```

Figure 6: P8: Functional checking style

```java
public String url() {
  StringBuilder url = new StringBuilder(this.path());
  if (!this.queries.isEmpty()) {
    url.append(this.queryLine());
  }
  if (fragment != null) {
    url.append(fragment);
  }

  return url.toString();
}
```
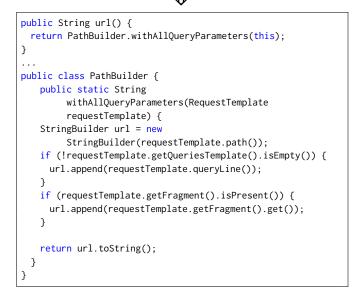
⇓

```java
public String url() {
  return PathBuilder.withAllQueryParameters(this);
}
...
public class PathBuilder {
    public static String
        withAllQueryParameters(RequestTemplate
        requestTemplate) {
    StringBuilder url = new
        StringBuilder(requestTemplate.path());
    if (!requestTemplate.getQueriesTemplate().isEmpty()) {
      url.append(requestTemplate.queryLine());
    }
    if (requestTemplate.getFragment().isPresent()) {
      url.append(requestTemplate.getFragment().get());
    }

    return url.toString();
  }
}
```

Figure 7: P9: Extracting class

P9: Extracting class (33% / 67%). This is the second occurrence of this kind of pair. In here, the original code used a `StringBuilder` and `if` statements to concatenate URL elements. The CDD version moved the `StringBuilder` and the `if` statements into a newly introduced class (called `PathBuilder`), and the URL is built using `PathBuilder.withAllQueryParameters(this)` (Figure 7). The results of the questionnaires showed a slight advantage for the code refactored with CDD version (89/133 developers voted for this

solution). One of the participants that favored the CDD version said that this version "*encapsulates the logic, and makes it easier to read. If necessary I enter the method to understand what it does.*". Interestingly, one developer that favored the original code had the opposite idea: "*Unless there is* PathBuilder *reuse, the code in* url() *will end up the same way inside the* withAllQueryParameters *method. The developer, in a sense, will need to go inside this method to see what is being done. It just increased one more layer to get to the code, since the* url() *method wouldn't have any other logic or flow.*"

P10: Encapsulating for loops (8% / 92%). In this final example, the original code concatenated added elements into a list using two for loops and one if statement, whereas the CDD version abstracted the two fors by using one native method of the List interface. Those in favor of the CDD approach mentioned that "*Even not knowing the whole API, just by the name of the method it is easy to understand its purpose*" and *Concatenating lists makes more sense and it is cleaner to use a lambda and an* addAll *than doing a* for *or* foreach. *The idea is the same, we use less line, and, we don't have to maintain this piece of code*".

In general, when considering the 10 pairs of code snippets chosen for this study it is possible to observe the influence of a cognitive constraint on reducing the presence of ICPs in all refactored classes. It is important to note that CDD focuses on improving code units, i.e., the classes (in object-oriented languages) are the main structures to apply the CDD principles. For this reason, we do not count in such code snippets the number of ICPs before and after refactorings because we would need to consider the whole class. However, this reduction can be perceived as if the CDD soft-forced the developers to restructure the classes to meet a satisfactory understanding threshold.

RQ2. Do CDD-guided refactorings improves code readability, according to the state-of-the-art readability model?

In here we present our findings regarding our second research question. Table 3 describes the results after applying Posnett model. In this table there are some × symbols that indicate that we were unable to calculate the metric. This happened due to two reasons. First, the pairs P4 and P9 moved part of the code to a newly introduced class. As we mentioned in Section 3.3, the tool we used to calculate Posnett model does not process an entire Java class, only method bodies. Since P4 and P9 contains class declarations, we were unable to run the tool on them. We were also unable to run the tool in P5 (which compares two approaches for listing imports), because Java methods do not accept imports declarations.

This table shows a couple of interesting observations. First, we could perceived that for 9 out of the 10 CDD-guided refactorings, the CDD version had fewer lines of code, when compared to the original version; on average, the CDD versions used 37% less lines of code. The only exception is P5, which doubled the number of lines of code used. This happened because the CDD version adopted the *explicit* approach (which lists all imports), while the original version used the *implicit* approach (which hide some of the imports), using the asterisk (*) wildcard.

Table 3: The results of Posnett model. We calculate the number of lines of code (LOC column) using the wc **UNIX tool (considering blank lines). The symbol** × **indicates when it was not possible to calculate it.**

Pairs	LOC Before	LOC After	Posnett Before	Posnett After
P1	9	5	0.0208	0.0174
P2	13	6	0.0013	0.0023
P3	16	11	0.1205	0.3368
P4	16	16	0.7029	×
P5	5	10	×	×
P6	12	7	0.0053	0.0092
P7	5	3	0.0192	0.0304
P8	2	2	0.0149	0.0276
P9	10	10	0.0066	×
P10	9	5	0.0127	0.0253

Moreover, regarding the results of the Posnett model, we noticed that CDD excelled in only one out of the seven pairs that we were able to calculate the metric; P1, in that case (the model performance was 0.0174, when compared to 0.0208 of the original version). For the all other pairs of code snippets, the original version had better performance then the CDD-guided refactorings. For this metric, the lower the value, the better readability the code snippet has.

However, when taking a closer look at this result, we could observe that the model performance did not varied much between the pairs of code snippets. For instance, for P1, P2, P6, P8, and P10, the performance variation was less than ~0.01. On the other hand, P3 and P7 showed a larger performance variation. For P3, in particular, the CDD-guided refactorings combined four if statements into a single if with several logical expressions. It is important to note that the Posnett model is, in part, based on the diversity of the code vocabulary, that is, the sum of unique operators and unique operands. Since the CDD-refactored version reduced the number of if statements, the operands were not reused anymore among the if statements. This might have worsened its performance due to the higher number of unique operators and operands.

5 DISCUSSIONS

Cognitive-Driven Development (CDD) is a coding technique that aims to reduce code complexity by always aiming to reduce the developer's cognitive load. These strategies are not fixed and can be adapted to each development team. The research in question evaluates refactorings based on CDD. However, although most of the practices are favorable to refactoring, we cannot affirm that CDD by itself is responsible for improving readability. However, it favors using other refactoring practices and good practices of Object-Oriented Programming.

The P2 is evidenced in Figure 4, where the name of the method, being self-explanatory, proved to be a very efficient practice, being mentioned by approximately 30 of the 124 participants in favor of the CDD approach. They highlighted in their comments that the method's name made all the difference in refactoring, making the language very natural and easy to understand. On the other hand, practice P2 (Figure 4) fostered reuse and was a point identified by

the research participants. Similarly, the use of stable classes such as `List` and `Assert` favor the use of CDD as they naturally decrease the number of ICPs. We also observed that CDD-inspired code snippets are smaller than the original ones.

As a final observation, we noticed that CDD makes the code more horizontally aligned. It happens because most of the refactorings in the code result in new methods or classes that, when reaching a limit, continue to be modified to fit the human mind, i.e., a satisfactory understanding degree.

6 RELATED WORK

We group our works in terms of 1) empirical studies that aims to assess code readability (Section 6.1) and 2) early work on CDD (Section 6.2).

6.1 Assessing Code Readability

Several researches have been working on the topic of code readability. While some proposed coding standards and conventions to assess code readable, others related readability to the cognitive and complexity metrics. We next discuss some of the closest works to ours.

Empirical studies on code readability. Works evaluating the readability of code snippets have already been done by researchers. Gerosa et al. [5] conducted a survey with software developers evaluating coding convention patterns and showed that most of these patterns positively influence the readability perceived by developers. A survey was conducted with 55 students of the software engineering course of the Computer Science course and another smaller group formed by 7 professional programmers of a large Brazilian software company. Buse and Weimer's [1] practices and the Scalabrino's model [21] were evaluated. At the end of the analysis it was identified that 8 out of 11 coding practices affected the readability perceived by the research participants. In our study, 10 pairs of code snippets were chosen from a previous study involving refactorings based on CDD. Sivaprakasam et al. [8] provided a tool to support their proposed method, which takes java methods themselves as input and returns refactored, readable source code by inserting blank lines after each block of valid code. Experimental results of the research showed that the automatic insertion of blank lines left the code with relevant lightness and a better understanding of the snippet due to the standardized organization and spacing of the parts with less cognitive load leaving the code more pleasant to read.

Descriptive model for code readability In the study by Buse and Weimer [1], readability was defined as the human understanding of the ease of understanding a text and that the readability of a program is related to its maintainability. The hypothesis of the study is that programmers have some intuitive notion to point out program features and characteristics that will be good indicators for readability. With this, a descriptive model of software readability based on programmers' opinions and notions of software quality was presented. To construct the model Buse and Weimer they conducted a study with 120 students of different levels of coding experience, asking participants to provide subjective rating scores of reading code snippets. A Survey was conducted where

each participant was given the same set of snippets. Participants could select a number close to five for "most readable" snippets or close to one for "least readable" snippets, with a score of three indicating neutrality. The result of this study was a set of code snippets accompanied by 12,000 evaluations on readability.

Posnett et al. [20] extended the Buse and Weimer's model [1] providing a simple and intuitive readability theory based on code size and entropy. They also point out some details that may have negatively influenced the Buse and Weimer model. They performed an extraction of code snippets longer than 200 lines and compared the two models addressed. As a result, it was possible to show that the proposed model outperforms the Buse and Weimer model as a readability classifier in small code snippets.

In 2018, Mannan et al. [24] used Posnett's readability metric to evaluate readability in large open-source projects. However, the results found a very low correlation between source code bad smells and readability. Considering the results and that Posnett's model was initially evaluated with small projects, Mannan et al. concluded that there are deficiencies in current readability models and therefore, there is a need to identify better metrics to evaluate readability.

In 2018 Fakhoury et al.[6], explored the effect of poor source code lexicon and readability on cognitive load as measured by a state-of-the-art minimally invasive functional imaging technique called functional near-infrared spectroscopy (fNIRS). The research results evidenced a significant increase in participants' cognitive load when anti-linguistic patterns and structural inconsistencies were introduced to the source code; for passages considered more readable, there was a decrease in cognitive load overload.

Programming language features and code elements In a work by Mendonça et al.[11], the impact of lambdas functions on JAVA programmers' understanding is evaluated. There is a common understanding that code refactoring with lambdas functions, in addition to other potential benefits, simplifies code and improves program understanding. A survey was carried out with 158 pairs of code snippets extracted from GitHub. As a result of the work after comparing with the Buse and Weimer model [1] and Posnett.[20] a contradictory result was found, both models suggested that refactoring by lambda functions does not improve the understandability of the source code, however in the qualitative result (survey) indicated that the introduction of lambda expressions in legacy code improves the understanding of the code in particular cases.

6.2 Early Work on CDD

Souza and Pinto [4] described the concepts that support the CDD. Continuous expansion is part of the complex nature of software. However, the understandability cannot follow in the same proportion. The lack of a clear relationship between software complexity and program comprehension contributes to the software not evolving healthy. As a consequence, software developers spend a considerable part of their time on program understanding [14]. Developers must know when to restructure the code and possibly improve the separation of responsibilities. According to the authors, when we do not have a well-defined rule for intrinsic complexity for source code, it will be increasingly common to find classes that contribute to a cognitive overload for developers.

Pinto at al.[18] (detailed in 3.1) reached the conclusion that refactorings using conventional practices guided by a complexity constraint were better evaluated when they were compared with the refactoring clusters (all classes created or modified) without such rule.

Pereira at al. [17] provided a tool called "Cognitive Load Analyzer" to support the CDD, a plugin for IntelliJ IDEA and Java language. The intrinsic complexity of the code is calculated through static analysis during programming, and the tool observes the limit of complexity. When the complexity limit is reached for some code unit, a notification is displayed to suggest possible refactorings.

In recent research, Pinto and Souza [19] evaluated the effects of adopting a complexity constraint in the early stages of software development. Three projects adopted by some companies for hiring new software engineers were selected to be developed by 44 experienced developers, divided into Non-CDD and CDD groups. Both groups were aware of the importance of quality metrics and the need to produce high-quality code for other developers to understand. The CDD group received different training that included practices guided by a cognitive complexity limit, including suggestions for elements to set a constraint. The result suggested that CDD can guide the developers to achieve lower dispersion for the quality metric measures (CK metrics).

The concept of Cognitive-Driven Development opens the door for extensive experimental research to measure the effectiveness of this strategy regarding the measurement of complexity from source code. Although most of the works discussed here involve the program understanding, qualitative studies involving readability criteria have not yet been carried out.

7 LIMITATIONS

As any empirical work, our also have limitations and threats to validity.

First, we spent several weeks cleaning and filtering the data provided by Pinto and colleagues [18]. Despite our best efforts to find representative CDD-guided refactorings (see details at Section 3.1), we may not have selected a diverse set of code snippets. To mitigate this threat, after grouping the code snippets into categories, we sought to have at most two code snippets per category.

Some participants also pointed out that some of our examples were very simple and, thus, not adequate. It is worth noting that such simplicity was intentional to make the survey feasible and less tiring. If we provide more complex examples that require more cognitive effort, we could potentially discourage participants from answering the survey. Still regarding the code snippets, manually selecting and formatting the code snippets may have, in some way, influenced the opinions of the participants. For example, by showing the code snippet as an image instead of a text, we may have limited the way our participants interacted with the examples (for instance, there were not able to copy the code to their IDEs), negatively influencing the opinions of some research participants.

Another limitation is regarding the way we present the code snippets. In the daily development routine, developers rarely have to spend time reading short methods without further navigation and navigation. Therefore, our experiment hardly assembles the real world development routine. We believe that experiments such

as this one could help researchers and tool builders create better models that could, in turn, be used to guide developers in writing software of better quality.

Finally, our data is based on the responses provided by 133 professional software developers. Although these developers belong to a group of great interest to our researcher, that is, professional developers with extensive software development experience, these developers pertain to a relatively restricted group of professional programmers with experience in Java. Therefore, it is not possible to generalize the conclusions obtained to general groups of programmers.

8 CONCLUSIONS

In this paper we evaluate the extent to which CDD improves code readability. To achieve this goal, we conducted a two-phase study. We conducted a survey with professional software developers invited to vote (and justify their reasoning) for the most readable pair of code snippets (out of a set of 10 pairs); one of the pairs was coded using CDD practices. We received 133 responses. In the second phase, we applied the state-of-the-art readability model to the 10 pairs of CDD-guided refactorings. We observed some with conflicting results.

The results allowed us to answer the research question RQ1, the developers perceived that seven (out of 10) CDD-guided refactorings were more readable than their counterparts; for two other CDD-guided refactorings the developers were undecided, while for only one of the CDD-guided refactorings did the developers prefer the original code snippet. Regarding research question RQ2, we note that only two CDD-driven refactorings show better readability, as evaluated by state-of-the-art readability models [20].

We conclude that by following CDD design practices, coding units are expected to be smaller and thus easier to maintain and evolve and thus easier to understand. It is important to stress that CDD only indicates a decrease in intrinsic Class complexity, so other refactoring methods are combinatorial and favor readability from CDD.Our results provide initial evidence that CDD may be an interesting approach to software design.

As perspectives for future work, we can perform the analysis by other readability methods and serve as a basis for their evolution, because our results demonstrate sometimes conflicting positions about some situations on the same piece of code and sometimes a near unanimity in favor of a piece of code. This information can be valuable to encourage other studies in the area.

REFERENCES

[1] Raymond PL Buse and Westley R Weimer. 2009. Learning a metric for code readability. *IEEE Transactions on software engineering* 36, 4 (2009), 546–558.
[2] W. R. Edwards C. M. Chung and M. G. Yang. 2010. Static and Dynamic Data Flow Metrics. In *Policy and Information*, Vol. 13. IEEE, 1–6.
[3] Shyam R Chidamber and Chris F Kemerer. 1994. A metrics suite for object oriented design. *IEEE Transactions on software engineering* 20, 6 (1994), 476–493.
[4] Alberto Luiz Oliveira Tavares de Souza and Victor Hugo Santiago Costa Pinto. 2020. Toward a Definition of Cognitive-Driven Development. In *IEEE International Conference on Software Maintenance and Evolution, ICSME 2020, Adelaide, Australia, September 28 - October 2, 2020*. IEEE, 776–778. https://doi.org/10.1109/ICSME46990.2020.00087
[5] Rodrigo Magalhães dos Santos and Marco Aurélio Gerosa. 2018. Impacts of coding practices on readability. In *Proceedings of the 26th Conference on Program Comprehension, ICPC 2018, Gothenburg, Sweden, May 27-28, 2018*, Foutse Khomh, Chanchal K. Roy, and Janet Siegmund (Eds.). ACM, 277–285.

[6] Ma Y. Arnaoudova V. & Adesope-O. Fakhoury, S. 2018. The effect of poor source code lexicon and readability on developers' cognitive load. In *In Proceedings of the 26th Conference on Program Comprehension - ICPC '18 New York, New York, USA: ACM Press*. ACM, 286–296.

[7] Maurice H Halstead. 1977. *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc.

[8] Christoph Hannebauer, Marc Hesenius, and Volker Gruhn. 2012. An accurate model of software code readability. *International Journal of Engineering Research and Technology. ESRSA Publications* 1, 6 (2012).

[9] Christoph Hannebauer, Marc Hesenius, and Volker Gruhn. 2018. Does syntax highlighting help programming novices? *Empirical Software Engineering* 23, 5 (2018), 2795–2828.

[10] Barbara A Kitchenham and Shari L Pfleeger. 2008. Personal opinion surveys. In *Guide to advanced empirical software engineering*. Springer, 63–92.

[11] Walter Lucas Monteiro Mendonça, José Fortes, Francisco Vitor Lopes, Diego Marcílio, Rodrigo Bonifácio de Almeida, Edna Dias Canedo, Fernanda Lima, and João Saraiva. 2020. Understanding the Impact of Introducing Lambda Expressions in Java Programs. *Journal of Software Engineering Research and Development* 8 (Oct. 2020), 7:1 – 7:22. https://doi.org/10.5753/jserd.2020.744

[12] Keung J. Xiao Y. Mensah S.& Gao Y. Mi, Q. 2018. Improving code readability classification using convolutional neural networks. In *Information and Software Technology*,. 60–71. https://linkinghub.elsevier.com/retrieve/pii/S0950584918301496

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

[14] Roberto Minelli, Andrea Mocci, and Michele Lanza. 2015. I know what you did last summer-an investigation of how developers spend their time. In *2015 IEEE 23rd International Conference on Program Comprehension*. IEEE, 25–35.

[15] Delano Oliveira, Reydne Bruno, Fernanda Madeiral, and Fernando Castor. 2020. Evaluating Code Readability and Legibility: An Examination of Human-centric Studies. In *IEEE International Conference on Software Maintenance and Evolution, ICSME 2020, Adelaide, Australia, September 28 - October 2, 2020*. IEEE, 348–359. https://doi.org/10.1109/ICSME46990.2020.00041

[16] William F Opdyke. 1990. Refactoring: An aid in designing application frameworks and evolving object-oriented systems. In *Proc. SOOPPA'90: Symposium on Object-Oriented Programming Emphasizing Practical Applications*.

[17] Jherson Haryson A. Pereira, Alberto Luiz Oliveira Tavares de Souza, and Victor Hugo Santiago C. Pinto. 2021. Cognitive Load Analyzer: A Support Tool for Cognitive-Driven Development. In *SBES '21: 35th Brazilian Symposium on Software Engineering, Joinville, Santa Catarina, Brazil, 27 September 2021 - 1 October 2021*, Cristiano D. Vasconcellos, Karina Girardi Roggia, Vanessa Collere, and Paulo Bousfield (Eds.). ACM, 468–473. https://doi.org/10.1145/3474624.3476011

[18] Victor Hugo Santiago C. Pinto, Alberto Luiz Oliveira Tavares de Souza, Yuri Matheus Barboza de Oliveira, and Danilo Monteiro Ribeiro. 2021. Cognitive-Driven Development: Preliminary Results on Software Refactorings. In *Proceedings of the 16th International Conference on Evaluation of Novel Approaches to Software Engineering, ENASE 2021, Online Streaming, April 26-27, 2021*, Raian Ali, Hermann Kaindl, and Leszek A. Maciaszek (Eds.). SCITEPRESS, 92–102. https://doi.org/10.5220/0010408100920102

[19] Victor Hugo Santiago C. Pinto and Alberto Luiz Oliveira Tavares. 2022. Effects of Cognitive-driven Development in the Early Stages of the Software Development Life Cycle. In *Proceedings of the 24th International Conference on Enterprise Information Systems - Volume 2, Online Streaming, April 25-27, 2022*. SCITEPRESS, 40–51.

[20] Daryl Posnett, Abram Hindle, and Premkumar T. Devanbu. 2011. A simpler model of software readability. In *Proceedings of the 8th International Working Conference on Mining Software Repositories, MSR 2011 (Co-located with ICSE), Waikiki, Honolulu, HI, USA, May 21-28, 2011, Proceedings*, Arie van Deursen, Tao Xie, and Thomas Zimmermann (Eds.). ACM, 73–82.

[21] C. Vendome M. Linares-Vasquez D. Poshyvanyk S. Scalabrino, G. Bavota and R. Oliveto. 2010. Automatically assessing code understandability: How far are we? *in 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2010), 417–427.

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

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

[24] I. Ahmed U. A. Mannan and A. Sarma. 2011. Towards understanding code readability and its impact on design quality. In *in Proceedings of the 4th ACM SIGSOFT International Workshop on NLP for Software Engineering*. IEEE, 18–21. https://doi.org/10.1145/3283812.3283820