The Journal of Systems & Software 206 (2023) 111830

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

In practice

Cognitive Driven Development helps software teams to keep code units under the limit! ☆

Gustavo Pinto *, Alberto de Souza

Zup Innovation, Brazil

ARTICLE INFO

Article history:
Received 14 March 2023
Received in revised form 12 July 2023
Accepted 30 August 2023
Available online 9 September 2023

Keywords:
Cognitive Driven Development
Design techniques
Software design

ABSTRACT

Software design techniques are key elements in the process of designing good software. Over the years, a large number of design techniques have been proposed by both researchers and practitioners. Unfortunately, despite their uniqueness, it is not uncommon to find software products that make subpar design decisions, leading to design degradation challenges. One potential reason for this behavior is that developers do not have a clear vision of how much a code unit could grow; without this vision, a code unit can grow endlessly, even when developers are equipped with an arsenal of design practices. Different than other design techniques, Cognitive Driven Development (CDD for short) focuses on (1) defining and (2) limiting the number of coding elements that developers could use at a given code unit.

In this paper, we report on the experiences of a software development team using CDD for building from scratch a learning management tool at Zup Innovation, a Brazilian tech company. By curating commit traces left in the repositories, combined with the developers' perception, we organized a set of findings and lessons that could be useful for those interested in adopting CDD. For instance, we noticed that by using CDD, despite the evolution of the product, developers were able to keep the code units under a small amount of size (in terms of lines of code). Furthermore, although limiting the complexity is at the heart of CDD, we also discovered that developers tend to relax this notion of limit so that they can cope with the different complexities of the software. Still, we noticed that CDD could also influence testing practices; limiting the code units' size makes testing easier to perform.

1. Introduction

The software development community has long recognized the importance of well-designed and modularized code to ease the maintainability and evolution process of a software product (Perry and Wolf, 1992). From the works of David Parnas in the 70s, indicating the need for better approaches to support software aging (Parnas, 1994; Parnas and Weiss, 1987), there has been a significant number of design metrics (Chidamber and Kemerer, 1994; Zage and Zage, 1993; Shaik et al., 2010), tools (Lefever et al., 2021; Sharma et al., 2016; Marcilio et al., 2020), and processes (Uchôa et al., 2020) that the research community envisioned to help software engineers in designing better software. Practitioners have also been fruitful in proposing well-adopted design approaches, such as clean code (Martin and Coplien, 2009), open-closed principles (Martin, 1996), etc.

Generally speaking, these design techniques aim to help developers translate requirements into optimized code units,[1] given a set of constraints (imposed by the problem and the designer) (Yau and Tsai, 1986). The research community has long recognized good design techniques as a critical factor in building reliable and understandable software (Parnas, 1994; Yau and Tsai, 1986). Moreover, some of these design methodologies became so widespread (e.g., separation of concerns or domain classes) that even novice programmers must understand some of them when applying for jobs (Galster et al., 2022).

Intriguingly, despite the growing number of design practices that aid the development of well-designed software, developers still employ subpar design decisions (Lavallée and Robillard, 2015). In the current landscape of software development, it is not uncommon to find software products that suffer from design degradation (Le et al., 2018), a scenario in which it becomes increasingly difficult to maintain the codebase while decreasing

☆ Editor: Daniel Mendez.

* Corresponding author.
E-mail addresses: gustavo.pinto@zup.com.br (G. Pinto), alberto.tavares@zup.com.br (A. de Souza).

[1] For the context of this work, a code unit is a source code file. We use the terms "code units" and "classes" interchangeably.

https://doi.org/10.1016/j.jss.2023.111830
0164-1212/

testability and reuse, impacting the overall effort to deliver new features (Sousa et al., 2018; Oizumi et al., 2019).

One potential reason these design approaches are not yet ubiquitous tools in the programmer arena is that they are *subjective*. Take, for instance, the catalog of bad design practices for refactoring testing code (Van Deursen et al., 2001). Although this catalog guides developers on what they could avoid/refactor, some of the items in this catalog are ultimately subject to one's interpretation. For instance, although assertions are undoubtedly valuable for detecting bugs, there is still little consensus on how assertions should be designed or documented (i.e., the assertion roulette bad smell Van Deursen et al., 2001) (Zhang and Mesbah, 2015; Kochhar et al., 2019; Aniche et al., 2013).

This limitation makes it difficult for researchers and tool builders to develop automated tools that find these bad practices with good precision. Another limitation is that these design practices do not provide a clear limit to developers. For example, how many smells a code unit might tolerate? What is the limit of design degradation we could handle before refactoring the bad smells? How many methods might a class have? How long should a testing method be?

Without a clear understanding of this limit, code units might grow endlessly, making it harder for developers to reason about big code chunks. Indeed, recent research suggests that component size is a primary indicator of maintenance effort (Gil and Lalouche, 2017). As pointed out by Kent Beck, one of the first agile practitioners and strong advocates for well-designed code, "*The goal of software design is to create chunks or slices that **fit into a human mind**. The software keeps growing, but the human mind maxes out, so **we have to keep chunking and slicing differently** if we want to keep making changes* (emphasis ours)".[2]

Cognitive Driven Development (CDD for short) (Barbosa et al., 2021; de Souza and Pinto, 2020; Pinto and de Souza, 2022) is a design coding technique that aims to reduce the complexity of code units (e.g., a class) by systematically posing a limit in the number of coding items – that adds complexity to that code unit – that could be used at once. Since CDD can be straightforwardly measured (in essence, it relies on counting the occurrences of complexity items in a given code unit), developers have little doubts about when and how they should apply it. Moreover, by limiting the number of items of complexity, CDD can guide developers to refactor code by increasing the awareness of classes (or methods) with higher complexity.

This paper We present the first "in vivo" study on the use of CDD in an industrial setting. This study was conducted at Zup Innovation, a Brazilian Tech company. We have studied the use of the CDD practice for over one year. This paper shares details about the studied project, the teams, and how they used CDD. Our goal with this report is to reflect on the team's (good and bad) experiences in using CDD on a daily basis to create a real-world software product. Therefore, our main research questions are the following:

RQ1. What are the positive impacts of the use of Cognitive Driven Development in a real software development product?

RQ2. What are the barriers for Cognitive Driven Development adoption in a software development team?

To provide answers to these questions, we revisited the artifacts created during the software development process to achieve this goal. In particular, (1) we mined our `git` logs, to find events that could indicate the use of the CDD practice, (2) we watched our video calls, in which the team reflected on the pros/cons of using CDD, and (3) we studied the team's documentation, in which they shared their rationale for their design and architectural decisions. After curating this data, we presented our early findings to the development team to double-check whether our results are aligned with their perceptions.

By reflecting on their experiences and analyzing the artifacts produced this year, we also contribute with lessons that could be useful for other teams interested in adopting CDD as their design technique. In summary, this paper provides the following contributions.

1. A detailed description about CDD and a step-by-step guide that interested developers can follow to use it;
2. A one-year reflection on the use of CDD as the main design practice for building an online learning management system at Zup Innovation;
3. A set of lessons learned that could be useful for other software producing teams interested in using CDD as part of their design methodologies.

2. What is CDD?

CDD is a coding design technique that aims to reduce the cognitive load developers may face by limiting the number of programming constructs they can use in a given code unit. In this section we distill our rationale for building CDD (Section 2.1), we introduce a guide that could help software producing teams interested in using CDD (Section 2.2), and we highlight CDD's key characteristics (Section 2.3).

2.1. CDD theory

CDD has its roots in the "*Magical Number Seven*" theory (Miller, 1956), which is a well-adopted psychological theory that suggests that there is a **limit** (and often small) in the number of information items that could be processed in the working memory at a given time. If a large number of information items are provided to process some task, the short-term memory may become overloaded – with an excessive cognitive load – potentially hindering one's understanding. Experimental studies suggested that humans generally hold only seven (plus or minus two) information items in short-term memory (Miller, 1956).

Another theory that CDD is built upon is the Cognitive Load Theory (CLT), proposed by Sweller (1988). CLT explains that any material has its **intrinsic complexity**, which varies according to the amount and arrangement of the information items that compose it. According to researchers (Paas et al., 2004), knowing the number of information items and their intrinsic complexities is crucial to guide instructors in presenting information at a pace and complexity level that the learners can understand.

For CDD, we borrow the notions of **limit** and **intrinsic complexity** to the software development practice.

- **Limit.** To cope with software complexity, CDD requires developers to flag each point of complexity inside a code unit. For example, suppose a code unit's number of points of complexities (inspired by Magical Number Seven) is greater than the stipulated limit. In that case, developers should refactor the code and move the complexity to other code units.
- **Intrinsic complexity.** CDD recognizes that the intrinsic load in source code (inspired by CLT) impacts developers differently. For a moment, suppose we have two developers working on the same code unit. Although they could likely disagree on what code elements hinder their understanding, this team of developers should curate a list with a few information items in the source code that they concur could make the code more complex.

[2] https://twitter.com/kentbeck/status/1354418068869398538.

Table 1
Overview about Handora's services. Column "SLOC" includes testing code while column "Commits" excludes merge commits.

#	Lang.	Description	SLOC	Code units	Commits
S1	Java	Core service	16K	230	767
S2	Java	Search service	3K	36	115
S3	Java	API provider service	12K	183	103
S4	Java	Grading service	1K	40	66
S5	TypeScript	Frontend service	18K	79	334
S6	Python	ML grading service	<1K	<10	<10

By the disciplined use of CDD, we believe that developers could design smaller and easier-to-test code units.

2.2. CDD practice

Developers are frequently impacted by cognitive overload during the software development process (Hermans, 2021). Usually, there are too many items of complexity that developers cannot efficiently process. More concretely speaking, in terms of source code, the programming constructs (their relationships) can be seen as our intrinsic items of complexity. Therefore, the first step in using CDD in a software development project is to define the Intrinsic Complexity Points (ICPs). Our definition of an ICP is as follows:

> Definition: *ICP refers to the measurement of coding elements that have the potential to elevate the inherent complexity of a specific unit of code.*

Some examples of ICPs are (but not limited to): code branches (e.g., `if-else`, `for` loops, `switch-case`, `do-while`, etc.), conditionals, high order functions, exception handlers (e.g., `try-catch-finally`), inheritance, etc. These are items of complexity because, when used (or abused), they may burden developers' cognitive load.

The ICPs are though not limited to the ones presented above. Indeed, the beauty of CDD is that developers can list any code element that could hinder their understanding, including testing code, SQL instructions, Infrastructure as a Code instructions, and the like (more on this in Section 7). For the sake of this work, we focused on Java-based ICPs.

After the selection of the ICPs that make sense to the development team, the team must define the cost of each ICP. Based on the Magical Number Theory, we believe that the overall cost of the code unit should be small. Therefore, our experience suggests that the individual cost of an ICP could vary from 0.5 up to 1, while the maximum cost of a code unit should not be greater than 10. If so, it is a flag indicating it is time to refactor to reduce its complexity.

2.3. CDD characteristics

CDD has at least two essential characteristics, which we present next.

1. **CDD is straightforwardly measured.** CDD relies solely on counting the number of ICPs; Therefore, it is not only easier to compute CDD-related bad smells, but they are also easier to communicate.
2. **CDD is flexible:** Philosophically speaking, CDD could be seen as a design theory rather than design practice. This is because CDD gives to practitioners a general guide on what they may compute within a given code unit. Thus, as any general guide, the decision of what to compute is left to the practitioner (and her team). The only hard constraint CDD places is regarding the disciplined use of the limit. Any code unit that is over the limit must be refactored.

3. CDD at Zup

In this section we describe our context at Zup Innovation (Section 3.1) and our approach to use CDD to build Handora (Section 3.2).

3.1. Context

Here we explain a bit about the context of company and the product we created (using CDD) to achieve the company growth's needs.

Zup Innovation[3] is a sizeable Brazilian tech company. It currently has ~3.5k employees, which work distributed mainly over Brazil, but also from other countries (from North America and Europe). As a way to attract novice developers, Zup has maintained several bootcamps – that is, short-term accelerating programs – focusing on training soft and hard skills to novice developers. To join the bootcamp program, these novice developers participate in a fierce selection process. The selected participants were hired and become immediately part of Zup staff. Since they are hired, they are expected to dedicate full time to the training process. These new hires spent 3–4 months in the training program, and during this period, no production-ready code is written; novice developers are expected to focus on their learning process. After the training program, the novice developers join real software development teams in the company. In 2021 alone, 12 bootcamps were delivered, and more than 300 novice developers were hired through these bootcamp programs.

3.2. Building Handora

To support our tailored training process, we built our learning management system, called Handora.[4] Unlike other online training tools, Handora was designed to handle the bootcamps' specificities, such as the flexible duration and the synchronous and asynchronous events. Handora is architectured as a set of six services.[5] A general overview of Handora is present in Table 1.

Four of the six services are written in Java, which are explored in this research. The TypeScript one (the frontend service) and the Python one (a machine learning service that provides a complementary grading process for the students' assignments) were not developed using CDD, so they are not the subject of our investigation. We discuss this limitation at Section 7. Moreover, the software development team used CDD only to design and implement the production code; the testing code was not written using the CDD principles. This is also part of our limitations.

3.2.1. Handora's team

The software development team comprises eight engineers: one CTO, one senior engineer, one mid-level engineer, one mid-level UX designer, one senior UX designer, one UX researcher, one mid-level data scientist, and one senior infrastructure engineer. The team work in a distributed, remote-first environment. Asynchronous communication is often preferred over synchronous ones, although the team meets synchronously for ~1 h almost every weekday.

[3] https://www.zup.com.br/.
[4] https://handora.zup.com.br/.
[5] Given their granularity level, we cannot consider them as microservices.

3.3. CDD at Handora

In this section we discuss how we used the CDD principles to guide the development of Handora.

3.3.1. Defining the ICPs

The first step towards using CDD is defining the ICPs. The ICPs can be seen as language constructs that, if abused, could hinder one's understanding. The Handora development team decided to compute five kinds of ICPs. Since the team had to annotate the ICPs manually, choosing five ICPs would provide enough ICPs to use the practice, while not burdening the developers with the manual effort. The selected ICPs are:

1. **Code branches:** this includes `if-else`, `switch-case`, the use of ternary operator, and all kinds of loops. This ICP resembles the cyclomatic complexity metric. The team counts 1 point for every branch. That is, while one `if` counts 1, one `if-else` counts 2.
2. **Conditions:** this computes the number of condition in `if`, loops, etc. The team believes that conjoined conditions in an `if` statement could potentially hinder code comprehension. The team counts 1 point for every condition. For instance, the expression `if (a > b && c < d)` computes 3: 1 for the `if`, one for the Boolean expression `a > b` and 1 for the other Boolean expression `c < d`.
3. **Exception handling:** this computes the use of `try-catch-finally` blocks. The team counts 1 point per block. For instance, a code snippet that contains a `try-catch-finally` blocks would count 3 ICPs: 1 for the `try` block, 1 for the `catch` block, and another one for the `finally` block.
4. **Internal coupling:** this includes the uses of classes of the same project. We compute ICPs only for a subset of the domain classes, for instance, classes that deal with the database are considered. The team counts 1 point when using these classes.
5. **External coupling:** this includes the use of code units from the JDK or external libraries and frameworks. Like internal coupling, the team considered only a subset of these external classes. Unlike internal coupling, though, the team counted only variable declarations. The team counts 0.5 points for each external coupling.

Regarding the costs, except for *external coupling*, all other ICPs had the same cost, that is, there was no weighting in the ICPs. The ICPs and the rationale for the costs were documented in the ADRs.

3.3.2. Defining the costs of each ICP

Afterward, one should also define the cost of each ICP. As a general rule of thumb, each ICP costs 1. However, the team could vary this definition accordingly. After a long discussion, the Handora team decided that external coupling cost 0.5 only. The rationale is that even though the team frequently deals with these cases, they still spend a few seconds trying to figure out their meaning; all other ICPs cost 1.

3.3.3. Defining the limit

The limit is an essential concept in CDD. It does not only indicate that one given class has more ICPs than expected, but it also requires developers to refactor that class to reduce its complexities. The development team was taught that a class over the limit conveys the same meaning as a class that does not compile: something is not right and should be fixed. The software development team decided 10 as the upper bound limit. This limit was a first suggestion that the team adopted, but did not change throughout the software evolution. Therefore, any class with more than 10 ICPs must be refactored; either to remove the ICPs or to extract the complexity of the ICPs to another class.

However, it is important to view the limit as a guiding principle, instead of a harsh limit. Although developers should strive their best to follow the guideline, in some circumstances, it is not always possible (as we shall discuss in Section 5.2).

```java
@RestController
@RequestMapping("/certificates")
@ICP(8)
public class CertificateDetailsController {

  @ICP(1)
  private CertificateRepository repo;
  @ICP(1)
  private TrainingCompleted trainingCompleted;

  @ICP(2)
  @GetMapping("/{certificateId}")
  public CertificateResponse execute(
                Long id, Student student) {
    @ICP(1)
    var potentialCertificate = repo.findById(id);
    var certificate = potentialCertificate.get();

    @ICP(2)
    if (certificate.doesNotBelongTo(student)) {
      throw new ResponseStatusException(NOTFOUND);
    }
    @ICP(1)
    var training = certificate.getTraining();

    return trainingCompleted.check(
        training, student,
        () -> new CertificateResponse(certificate));
  }
}
```

Fig. 1. A simplified version of an annotated class using CDD.

3.3.4. Computing the ICPs

Since the team did not employ tools for searching and computing the defined ICPs in the source code, the team had to compute them manually. To do so, the engineers created the `@ICP` annotation, which they used to annotate each ICP instance within a class. This facilitated the process of counting the ICPs because the team used the search features of the IDE. After computing the individual ICPs, the class is annotated with the total ICPs. An example of how the team calculated the ICPs is presented in Listing 1.

From top to bottom: the first `@ICP(8)` means that this class used 8 ICPs overall. Inside the class, the team flags every use of the selected ICPs. There are two `@ICP(1)`, one for each variable: `CertificateRepository repo` and `TrainingCompleted trainingCompleted` (indicating an internal coupling). On top of the `execute` method, there is `@ICP(2)`, indicating an internal coupling with the `Student` class (a parameter) and another one with the `CertificateRespone` (the response type); the team does not consider the use of the `Long` type as a form of coupling. The next use of `@ICP(1)` is regarding the coupling with the `repo` var (from the `CertificateRepository` class). There is also one `@ICP(2)` for using an `if` statement (1 ICP) and for having one condition (other 1 ICP). The final `@ICP(1)` refers to the coupling with `training`. Note that the lambda expression was not annotated because the team decided not to consider it an ICP.

3.3.5. Revisiting the ICPs

The development cycle could be summarized as "1 + 4 weeks": the first week is dedicated to planning activities while the other four weeks are for building activities. The first week is

also known as the "slow down" week, where the engineers could relax a bit from the four building weeks. During this planning week, the engineers also had one regular spot to discuss the use of CDD. During this meeting, the engineers shared the decisions they made to keep the classes within the limit, and the challenges in doing so. The meetings were recorded and analyzed in this study. During this study, we observed ∼40 cycles.

3.3.6. Disagreements with the defined ICPs

Since there is a large variety of what different programmers consider "complex", it is expected that team members would have different opinions of which ICPs they should consider. This indeed happened with the Handora team. The first set of ICPs were defined by the team leader, but it was constantly revised by the team members during the CDD-related meetings. In summary, the team should try to find consensus regarding the set, the cost, and the limit of the ICPs. If engineers disagree or were neutral, the CTO was responsible to give the final answer.

4. Research methodology

The study is primarbased on an observation of the software development team: while the first author was not a participant of the team, the second author played the CTO role in the team. We complement these observations with data and metadata from the repositories, data from the team's meetings, data from the online documents, and others.

4.1. Data collection

The data collected in this work covers October 2021 (when the software development project started) until September 2022, when the first release of the product was made. We collected data from three different sources: git logs, ADRs documents, and video retrospectives.

4.1.1. Mining `git` logs

We mined `git` logs in order (1) to understand how classes evolved over time, and (2) to find and understand CDD-driven code changes. More concretely, we analyzed all commits performed in the `main` branch of the three studied services, totaling 985 commits; we excluded merge commits. These commits were performed by a team of five developers. For each commit, we then performed `git checkout` to restore the working tree files, and then we run a set of tools, such as Spoon (Pawlak et al., 2015), to compute general coding and CDD metrics, and Refactoring miner (Tsantalis et al., 2022), to assess the presence of refactorings. As we shall see in Section 5, this mining part contribute to understanding how CDD was used, and how it impacted Handora's development.

4.1.2. Revisiting the ADRs

We revisited the Architecture Decision Records (ADRs), which are text-based documents describing the main design/architectural decisions made during the software development process. ADRs are versioned in git, but in a different repository. The first author revised these documents and employed a qualitative coding technique to categorize them. On average, one ADR has 138 words. Among the 8 ADRs, just one contained CDD-related information. In this particular ADR, the development team updated the set of ICPs used, and the decision to use them.

4.1.3. Rewatching the CDD retrospectives

We then watched the recorded online meetings in which the team reflected on the use and adoption of CDD. Although we had a regular spot for this meeting during the planning week, only six videos were found and analyzed. This could be due to two reasons: (1) not all meetings were recorded and (2) some meetings might be canceled due to other schedule priorities.

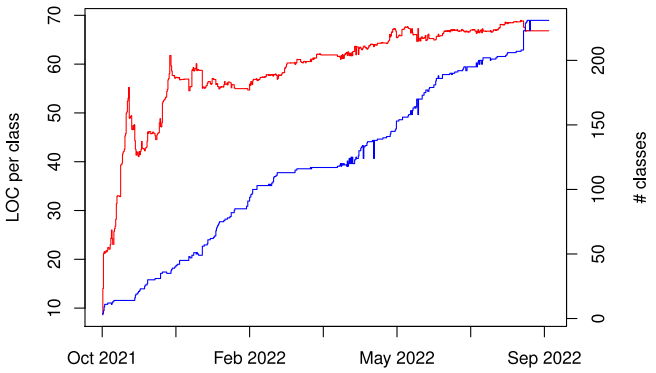


Fig. 2. The average of (1) the sizes (in terms of SLOC) of the classes (red line) and (2) the number of created classes throughout software evolution (blue line). We used the `wc` UNIX tool to calculate lines of code. We considered blank lines and comments. We did not consider testing classes here.

4.1.4. Data analysis

The data analysis was performed as follows: the first author studied the documents found and designed initial observations. The second author, who was part of the software development team, could confirm or refute those observations. This author eventually brought up his perception, which was not always documented. Whenever we have potential pieces of evidence about the benefits/challenges in the use of CDD, we present them to the remaining engineers in the team. To mitigate potential biases, we first asked their perception (e.g., "do you think CDD could also impact the size of the testing methods?"), and only after their response we showed our findings. We also asked the engineers individually, in no particular order. Three engineers gave regular feedback, which we used to refine our claims. Before submission, this report was read by the whole software development team. Since the software engineering team was unaware of the conclusions reached in this study, we could validate the findings we drew from the data during this process. In general, the engineers agreed with our observations.

5. Findings

In this section we group the main findings of this study in terms of the research questions.

5.1. What are the positive impacts of the use of cognitive driven development in a real software development product?

CDD limits the size of code units. In Fig. 2, the red line shows the evolution of the averaged lines of code. In contrast, the blue line shows the evolution of the number of classes created during the evolution of the project.

Here we noticed our first striking observation: the number of lines of code (red line) per class dramatically increases at the beginning of the project. Then, however, around 1/5 of its evolution, the growth starts to plateau. On the other hand, we could notice a nearly linear growth of the number of classes (blue line). Taken together, these two lines indicate that code units can be kept under the limit, even with the (near) linear growth of the software product. We presented this finding to the software development team, who concurred that this was primarily due to the CDD practice.

We then took a closer look at the latest Handora release. Fig. 3 shows the result. We noticed that, on average, an Handora class has 59 lines of code (17.2 standard deviation). This observation corroborates with our previous findings, indicating that CDD

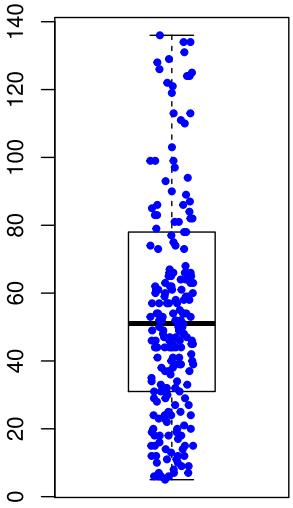


Fig. 3. SLOC distribution.

could impact the size of the classes. The software development team reinforced that the small size of the classes is probably due to the limit that CDD places. One of the team members also indicated that if the limit was, say, 30, the classes would probably be more extensive. This suggests that the disciplined use of CDD could drive the lengths of the classes.

Key impact #1: By using CDD, the software team could keep code units under the limit, even with the (near) linear growth of the software.

CDD impacts ICP size. Another way to observe the growth of complexity over time is by computing the number of ICPs per code unit. In Section 3.3.1 we described the ICPs considered by the software development team. To compute these ICPs, we built a static analysis tool using Spoon (Pawlak et al., 2015). Fig. 4 shows the average number of ICPs per class throughout software evolution.

This figure shows a similar behavior observed in Fig. 2: it seems that in the very first commits of Handora, there is a high fluctuation in the use of computed ICPs. However, as time passes by, there is also a sort of plateau. There seems to be a correlation between the size of classes and the number of used ICPs.

Key impact #2: There seems to be a correlation between the size of the class and the number of ICPs used in the class.

CDD impacts method size. According to Gill and Kemerer (1991), the module size can indicate code quality since maintenance effort is positively correlated with method length. Moreover, Chowdhury and colleagues (Chowdhury et al., 2022) discovered that methods with 24 lines of code or fewer are less prone to bugs for open-source Java projects. In conclusion, the authors suggested that "developers should strive to keep their methods within 24 SLOC". We used this evidence to shed some light on whether CDD might also influence the quality of the size of the methods in Handora.

To do so, we extracted data from S1, S2, and S3 only because they are the services with the greatest number of code units and methods. Following mining best practices, we filter out `getters` and `setters` methods, testing methods, as well as `hascode()` and `equals()` implementations, because they could add noise to the data (Abid et al., 2019). Overall we removed 525 methods; 677 methods remained for analysis. In particular, S1 had 230 code units and 338 methods, S2 had 36 code units and 72 methods, and S3 had 183 code units and 267 methods. Fig. 5 depicts the distributions found.

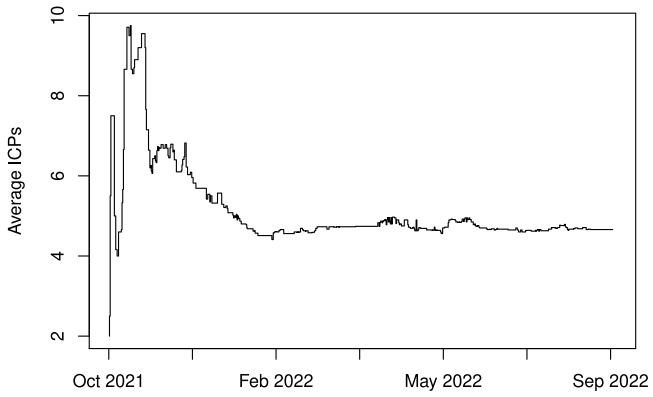


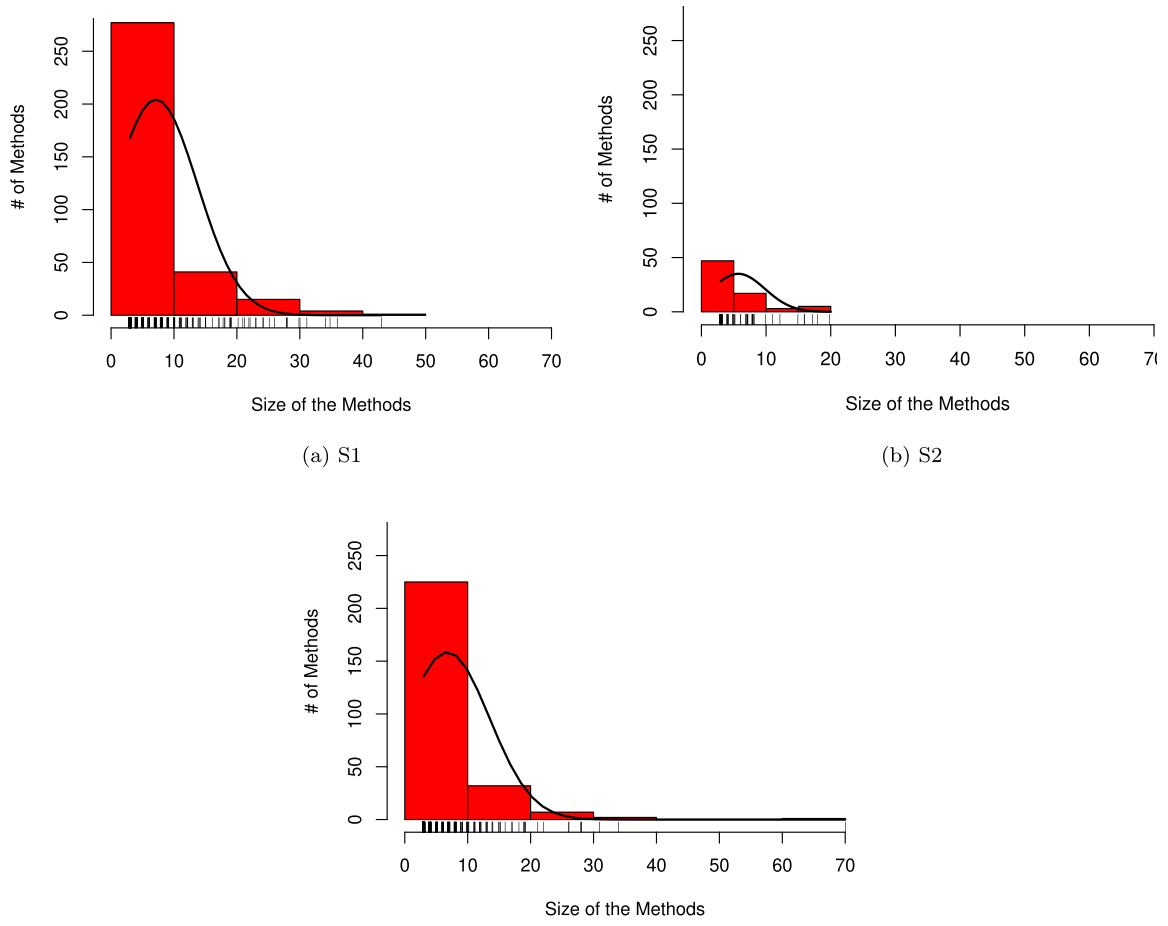
Fig. 4. The averaged value of ICP per commit. We did not consider testing classes here.

As one can observe, overall, the majority of the methods in these services are under the 24 SLOC limit. More concretely, for S1, 96.4% of the methods are under the 24 SLOC limit (S2: 100.0%; S3: 97%). On average, a method at Handora has 6.8 lines of code (median: 4; standard deviation: 6.4). We then asked the development team whether they see any influence on the use of CDD on the size of the methods. They concur that the size of the methods might also be influenced; one of the engineers commented that: "Every unit of code is impacted, because we know what the limit is and what goes into that limit".. The most extensive method at Handora has 70 lines of code, at S3. This particular method implements a visitor pattern that navigates through Markdown files. This method computed 8 ICPs, whereas the class computed 10 ICPs overall. We also asked the development team whether they had any intention of refactoring this particular method. The engineers said that since the class is under the limit, there is no purpose in refactoring a method, even if a large one. Another engineer noted that the use of CDD highly influences the decision of when to refactor. Therefore, they hardly refactor classes/methods that are under the limit.

Key impact #3: By using CDD, achieving the 24 lines of code threshold becomes a reasonable easy task for the software development team.

CDD impacts testing code. As mentioned in Section 3.2, although the team did not use CDD for testing purposes, here, we also aimed to understand whether CDD also impacted the testing code. Table 2 shows a summary of the testing code at Handora.

As shown in the table, except for S4 (the minor service), all other Handora's services have reasonably good code coverage. By means of comparison, code coverage at Google is around ~80% (Ivanković et al., 2019). On the other hand, the proportion of testing code units is reasonably smaller than the production code units. On average, there are 3.2 testing methods per testing unit, while there are 2.4 methods per production code unit. That is, Handora has fewer testing units, although these testing units tend to concentrate a higher number of methods, when compared to production code units. In terms of size, on average, testing



(a) S1 (b) S2

(c) S3

Fig. 5. Distribution of the methods' lengths at Handora. This figure does not include testing methods.

Table 2
Overview of testing characteristics at Handora. The column "Testing Units %" means the percentage of testing classes compared to the overall classes at Handora. Coverage data was calculated using JaCoCo.

#	SLOC	Testing units		# Methods	Coverage	
		#	%		Instr.	Branch
S1	7.6k	77	33.4%	215	66%	71%
S2	1.3k	19	52.7%	41	37%	61%
S3	5.2k	53	28.9%	128	58%	64%
S4	0.3k	7	17.5%	25	2%	0%

Table 3
Descriptive statistics on the number production methods and testing methods at Handora.

	Min	Avg	Median	Max	SD	Histogram
Production	3	6.8	4	70	6.4	
Testing	3	8.1	7	34	5.0	

methods have 8.1 lines of code (median 7; max 35; standard deviation: 5), which is inline with recent research that suggests that small testing methods (10 SLOC or less) as a good testing practice (Kochhar et al., 2019).

Table 3 compares the production code's size and the testing code's size. In this table, we can observe that, on average, the size of a testing method is generally more significant than the production method (8.1 SLOC vs 6.8 SLOC, respectively). On the other hand, production methods have a longer tail, meaning that production code units have larger methods. This finding might also be aligned with the study of Kochhar and colleagues, which observed practitioners concurred that "a good suite contains lots of small test cases (with fewer LOC) and few large test cases" (Kochhar et al., 2019).

We also asked the team about the testing effort. One of the engineers mentioned that creating new testing code—at the latest release of Handora— somehow requires *less* effort than the effort they placed at the project's beginning. The engineer complemented by suggesting that this might be due to the reuse of the existing testing infrastructure. When we asked whether the size of the production classes might influence their testability, one engineer said, "I think there is a relationship because the complexity of the test can be seen as a proxy of the complexity of the code under test". Another engineer complemented that "there is an indirect relationship between CDD and the size of the testing code. Since CDD leads to small code units, these code units are also easier to test".

One of the engineers also complemented that "penalizing conditionals tends to ease the test writing process". This happens because, according to more sophisticated coverage criteria such as MC/DC (Yu and Lau, 2006), the higher the number of conditions, the higher the number of test cases required to cover them. We found that, on average, there are 1.3 conditions per `if` statement in the codebase.

Finally, we tried to assess any association between the size of the production methods and the testing methods. Since these

sets of production and testing methods have different lengths, we *downsampled* the production set to match the size of the testing set. We ran the Shapiro–Wilk, which indicated that both sets are not normally distributed (production = 0.63494, *p*-value = 0.0001; testing = 0.80003, p-value = 0.0001). We ran the Pearson correlation because it does not assume normality. Pearson correlation, though, indicated no association exists between the size of the two sets of methods (p = −0.09742104).

Key impact #4: Testing code is consistently smaller than production code. The team recognized that the effort to create new testing code is reasonably small, which might be due to the size of the production code.

CDD-driven commits. During the development process, the team was also instructed to perform commits with the pattern "`cdd(class): description`" whenever they needed to perform changes to adhere to the CDD principles. We mined these commits to shed additional light on the developers' intentions when performing them. We found 38 commits with this directive: 32 in S1 and 6 in S3. We found no CDD commit in the other services. While manually examining these commits, we discovered a few intentions:

- **Annotating classes**. The development team does not always use the `@ICP` annotation. In the beginning of the project, the value of the ICP was documented as a comment in the source code. After introducing the `@ICP` annotation, the team had to annotate the unannotated classes.
- **Recomputing ICPs**. Whenever a new point of complexity is added to the code, the team has to recompute the class's ICPs. This kind of commit usually performs minor changes in the source code, in addition to the new ICP value for the class.
- **Committing feature first, CDD later**. The software development team was also instructed to focus first on completing the coding task. After implementing a new feature, the team refactored the code to adhere to CDD. To avoid technical debt, the team was also instructed not to delay these refactorings. The team intended to make the code units fit under the stipulated limit in these kinds of commits.

Considering that the team often performs these CDD-driven commits as a way to conform the code to the CDD guidelines, we hypothesized that these commits are also more likely to perform refactoring operations. We then ran RefactoringMiner (Tsantalis et al., 2022), the state-of-the-art tool in finding refactoring operations in commits. RefactoringMiner found 29 kinds of refactorings, performed 266 times at Handora. On average, one CDD-driven commit includes seven refactoring operations. Among the 29 kinds of refactorings, 11 (38%) are annotation-based refactorings, mostly due to the need to adapt the `@ICP` annotation. The annotation-based refactorings include: Add Attribute Annotation (25 instances; 9.3%), Modify Class Annotation (26 instances; 9.7%), and Remove Parameter Annotation (17 instances; 6.4%). Other than annotation-based refactorings, we also found a few instances of class-based refactorings (8.6%), such as Move Class (7 instances; 2.6%), Rename Class (12 instances; 4.5%), and Extract Class (4 instances; 1.5%). Indeed, these refactorings tend to be less common since they touch on more stable parts of the code (i.e., class declarations). We then compared the presence of these class-based refactorings in the other Handora commits. On average, an ordinary commit in Handora includes 2.6 refactoring operations. Class-based refactorings also appear less often: overall, they represent 6.8% of the refactorings performed. The class-based refactorings found are: Change Class Access Modifier (12 instances; 0.6%), Extract Class (12 instances; 0.6%), Move Class (45 instances; 2.3%), and Rename Class (63 instances; 3.2%).

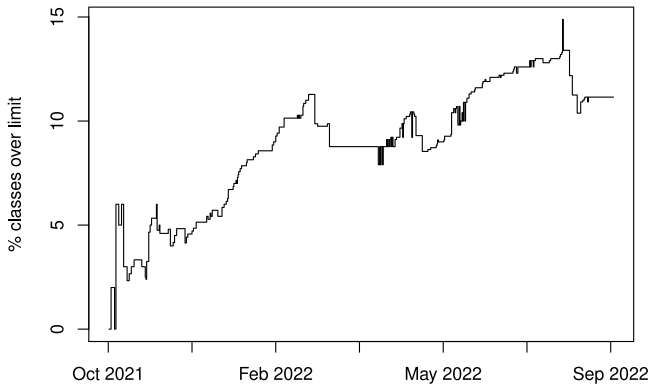


Fig. 6. Percentage of classes over the limit. We did not consider testing classes here.

Key impact #5: By instructing the team to finish the task first and then adhere to CDD practices, developers tend to perform refactoring operations more often.

5.2. What are the barriers for cognitive driven development adoption in a software development team?

CDD's limit is not written in stone. There are, obviously, classes that are bigger than the stipulated limit. Fig. 6 shows the percentage of classes over the limit during the software evolution.

As shown in this figure, the percentage of classes over the limit increases over time. At the peak, there were 14% of classes over the limit. Although the software development team strongly advocated in favor of CDD, we observed some scenarios in which the development could not reduce the complexity of the class to fit into the limit. Two examples are following:

- **Core classes:** Handora is an online learning management system, and as such, it is based on several learning theories. The classes that implement these learning theories are also less likely to stay under the limit because they are core classes in the system, and as such, they have to cope with many responsibilities.
- **Classes with rich contracts:** Handora is built as a set of orchestrated services. These services talk to each other through well-established contracts, implemented as Data Transfer Objects (DTOs). Though contracts cannot be reduced or simplified, and thus the DTO classes are less likely to stay under the CDD limit.

Key challenge #1: Keeping code units under the limit is not always feasible, in particular, for core classes of the system.

Adhering the limit might lead to workarounds. We noticed alternative solution that the software development team employed in order to adhere to the CDD limit. For instance, to minimize the size of **core classes**, the team adopted the use of *partial classes*. A partial class is a feature available in C#, which the team incorporated into the product. By using partial classes, a single class's functionality can be split into multiple files, and these

files are merged into a single one. However, the use of partial classes also brings shortcomings. For instance, when splitting a class into three files, the team used the following naming convention: `PartialClass$1.java`, `PartialClass$2.java`, and `PartialClass$3.java`. Although the team was able to come up with better names in some cases, the above naming convention was still common. The development team acknowledged that this convention could potentially hinder the understanding of a new developer onboarding the team. Moreover, these classes are grouped in isolated packages; therefore, navigating between these classes might not also be straightforward.

The team also reported that they could not use the partial class approach to refactor the **classes with rich contracts**. This happened because the contracts are in JSON format, and the JSON document is automatically built using the public attributes defined in the classes. If the team split the classes into several files, this would generate JSON documents with different formats, impacting the clients. Knowing that DTO classes would hardly fit in the 10 ICPs limit, the development team decided that, for these kinds of classes, the limit would be 20 ICPs.

Key challenge #2: Some of the strategies to deal with the CDD limit could also introduce addition burden in the development team.

Automated prototypes did not help CDD adoption. The software development team at Zup had to compute the ICPs manually. During the meetings, the team reported that computing them is a tedious and (often) error-prone task, requiring a non-trivial cognitive effort. Although we have built two static analyzers for CDD (one integrated with IntelliJ IDEA, and one stand-alone tool), the software development team did not embrace them. In particular, since the stand-alone tool was not seamlessly integrated within their work environment, one team member said that it was not natural to use the tool. As for the plugin developed for IntelliJ, it required the team to use an old version of the IDE, since the newer releases of IntelliJ broke the contract with the static analyzers (and fixing the plugin was hardly a priority).

Key challenge #3: CDD detection tools should be well-integrated in programming environments.

CDD may increase developers' effort. Although one of the primary advantages of CDD is to reduce developers' cognitive load, our observations and feedback from the engineering team indicate that some effort is required for developers to become accustomed to utilizing CDD effectively.

One engineer expressed that, despite using CDD for over a year, it remains somewhat challenging for them to grasp fully. Additionally, by mining CDD-driven commits, we noticed several instances where commits focused solely on updating the `@ICP` annotation, without any further changes. This observation suggests that developers may not always accurately count the ICPs and subsequently need to make adjustments.

Key challenge #4: CDD seems to require some effort from the engineering team to become accustomed to its use, as evidenced by commits focused on updating annotations without other changes.

It takes time to find the ideal set of ICP. To get started with CDD, the team leader defined the first set of ICPs. However, this set was not immutable; throughout the CDD-related meetings, the team had the opportunity to adjust the initial selection of ICPs. While rewatching the retrospective, we found at least two moments in which the team disagreed with the set of selected ICPs:

- First, there was a long discussion about whether the team should consider using lambda expressions as part of the ICPs. While one team member argued that using lambda expressions could hinder his understanding, two other engineers had different opinions. Therefore, they decided *not* to consider lambda expressions as an ICP.
- Second, the team once thought that they should consider all forms of variables (class variables, method variables, method parameters, etc.) as part of the ICPs. However, they soon realized that keeping track of all forms of variables would greatly increase the team effort in adopting CDD. After this realization, the team discussed and decided they would only consider method's parameters as part of the ICPs, because for them, it was more important to quickly figure out a method's meaning — and having a long list of parameters would hinder this understanding.

Key challenge #5: It is important yet hard to find an effective pre-defined set of ICPs.

6. Lessons learned

In the previous section, we report the team's experience using CDD to build a real software product at Zup Innovation. In this section, we reflect on this experience and extract a set of lessons learned that could be useful for practitioners interested in adopting CDD.

CDD's flexibility is good. As we presented in Section 2.3, CDD has two unique characteristics: (1) it is flexible so that developers can choose the ICPs they believe are the most important ones. After defining, (2) the ICPs can be objectively measured. This work also highlighted only one hard constraint: the limit. However, we also noted that the software development team was able to flex it a bit. For instance, to cope with the rich contracts of the services, the development team decided to change the CDD limit of these kinds of classes (from 10 to 20). Based on this experience, we envision that software development teams could define different limit levels for different domain classes.

CDD's flexibility is bad. Since CDD allows the team to decide the types, the costs, and the limit of the ICPs, these decision could become personal; other teams could decide differently. Although we presented the configuration adopted by the Handora's team, it is hard to provide a general guidelines.

There is no silver bullet set of ICPs. Given the myriad of ICPs one could choose, the team had to work hard to define (and refine) the ICPs. The ones that the development team chose (Section 3.3.1) were the ones that the team believed could ease maintenance efforts. As one engineer said: "*in my opinion, it is not about finding the perfect ICP, but finding the ones that work in that given context; and this is the beauty of CDD*". Therefore, engineers interested in adopting CDD could start from our set of ICPs, and adapt as needed.

The success of CDD highly depends on well-defined ICPs. Defining the right set of ICPs is not only crucial but also presents a unique set of challenges. The importance of defining the right ICPs cannot be overstated, as it significantly impacts the software development process. Without adequate tool support, analyzing and tracking ICPs would demand substantial effort from the

team. Moreover, the process itself is inherently challenging due to the potential divergence of opinions among team members regarding the selection of relevant ICPs. Consequently, achieving consensus becomes an arduous task, especially within diverse teams encompassing individuals from various backgrounds and experience levels.

CDD might *guide* programmers to perform refactorings. Even if developers are aware of the presence of code smells in the codebase, they may not always refactor the code to remove them (Silva et al., 2016). By using CDD, we believe developers can perform refactoring operations more conscientiously, due to the notion of limit. The limit reinforces the need to refactor. Indeed, the software development team indicated that they only refactor a class to reduce its complexity *if* that class is above the limit; otherwise, they do not refactor.

CDD might *prevent* programmers from performing refactorings. When developers rely mostly on the CDD metric to perform refactorings, they could refrain from refactoring code that falls below the defined limit. While CDD provides a valuable measurement tool, it should not override the programmers' holistic understanding of the codebase and its specific requirements. Therefore, it is essential to communicate CDD as a complement to intuition, encouraging developers to use their judgment to determine when refactoring is necessary, even if the metric is not exceeded.

CDD helps in identifying potential god classes earlier. God classes "feature a high complexity, low inner-class cohesion, and heavy access to data of foreign classes. It violates the object-oriented design principle that each class should only have one responsibility" (Olbrich et al., 2010). For developers, avoiding god classes is of paramount importance, and researchers have been studying approaches to detect and refactor them for quite some time (e.g., Lanza and Marinescu, 2007; Marinescu, 2001). Our experience suggests that – by design – CDD helps developers avoid god classes. This happens because CDD increases the awareness of the size of the code units so that any code unit getting big is clearly noticed in advance before it gets huge.

The lack of tool support hampers CDD adoption. To ease CDD adoption, developers interested in adopting CDD still need production-ready tools that can be used in different programming environments. Not only for computing ICPs, but tools that could ease visualization, perform code transformations, or increase developers' awareness of refactoring opportunities; this could be impactful in contributing to CDD adoption.

Software teams must exercise self-discipline. Due to its manual characteristic, CDD requires a lot of discipline from the teams. As a consequence, teams that operate under a tight schedule might have a hard time trying to adopt this design practice. Indeed, the benefits that we perceived in using CDD encouraged us to advocate in favor of CDD to other software development teams at Zup. However, our experience so far in spreading the use of CDD was not successful. Even though the teams seemed to see the value in adopting CDD, these teams were operating behind schedule, so adopting a new design theory requires some time that these teams were unable to commit.

Limiting the size of a code unit through ICPs is better than through SLOCs. One observant reader might argue that our approach could be even easier to employ if we consider the SLOC of a class as our notion of limit. However, our experience suggests that limiting the size in terms of SLOC could lead to biased results. This could happen due to at least two reasons: (1) small classes could still have a high level of complexity, and (2) SLOC may as well consider elements that do not add complexity to the code (e.g., comments, classes, and methods declarations, etc.). If developers have to filter out these elements, this could, in turn, hinder its adoption by developers.

7. Limitations

Unlike traditional MSR studies, which investigate an ever increasing number of data points, our study is based on observing a single team working on one single software product. Handora itself is not a mission-critical product. Handora's codebase is written in an established programming language, using popular frameworks. Although our findings contribute to understanding how CDD was helpful in creating Handora, we highlight that these findings would hardly generalize to other teams or products, either on Zup or from other companies. It is also unclear how CDD could contribute to designing more complex software projects.

Another limitation is regarding the team's experience using CDD to develop Java-based services. Although Handora has other services written in different programming languages (such as TypeScript and Python), the team decided not to use CDD in these additional services. The rationale for this choice is that, since the software development team was still getting acquainted with CDD, adopting it in different scenarios would burden the team more. Still, we did not checked whether the ICP annotations are complete (in terms of false negatives). Indeed, we noticed commits that solely changes the `@ICP` annotation, which indicates that developers may miscount the ICPs.

Moreover, as discussed in Section 8.4, CDD flexibility allows one to compute any kind of ICP. In this paper, we limited our observations to Java-based ICPs. Still, we also understand that there are many other ICPs in other programming languages and frameworks that the team could consider. Still, we did not use CDD for designing testing code because the team was unsure which ICPs they should use and follow. It is unclear how the use of different ICPs could drive the software development process, in particular, when choosing ICPs that could be more complex to compute. We believe these limitations could drive new CDD-related research in the future.

A final limitation is regarding how the team perceived the benefits of CDD. Since the team members are early adopters of the CDD practice, they might be positively biased towards CDD; for instance, being more motivated to produce high-quality code. Moreover, their interest in using CDD might have overcame the challenges. When we asked the team about the potential difficulties that CDD brings to the software development process, the team talked mostly about (1) the lack of tools and (2) the need to recompute the ICPs. To some extent, this limitation was alleviated because these were professional programmers. Still, the team did not know they would be part of a research study.

8. Related work

In this section we group the works on software metrics for code complexity (Section 8.1), those that assess developers' understanding of code complexity (Section 8.2), and the studies that suggest different forms of code complexities (Section 8.3). We close this section by presenting some early work on CDD (Session 8.4).

8.1. Software metrics

Software metrics are unquestionable tools in the programmer's arsenal. However, they also have known limitations. Take, for instance, three well-known metrics: (1) McCabe's cyclomatic complexity (McCabe, 1976), a metric that computes the number of independent paths of a program, Hasltead's complexity (Halsted, 1977), a vocabulary-based metric, and SLOC, that counts the source code lines of code. If we calculate these three metrics for the program shown in Fig. 1, these three metrics will yield different (perhaps conflicting) results. Since one of the goals of these metrics is to guide practitioners to parts of the code that need more attention, the different calculations these metrics perform could cause confusion.

Moreover, in a systematic mapping study conducted by Varela et al. (2017), it was observed that only during the period from 2010 to 2015, there were more than 300 code metrics proposed or analyzed; code complexity, in particular, was one of the most common category observed in this study. This high number of complexity metrics can challenge software engineers.

8.2. Assessing developers' understandability

Other studies focused on understanding developers' challenges in program comprehension tasks. dos Santos and Gerosa (2018) asked 62 survey participants to grade well-known coding practices. The authors found that four (out of the 11 tested practices) tend to decrease readability. Wiese et al. (2021) studied how students (mis)understand code snippets containing multiple Boolean expressions. The authors found that students tend to misunderstand conditions when several Boolean expressions are conjoined; if the expressions are separated in individual `if` statements, students tend to understand better.

More recently, some researchers are employing eye-tracking tools to trace participants' eye movements and thus figure out the parts of the code on which they focus their effort. For instance, Nahla and colleagues (Abid et al., 2019) found that developers tend to focus most of their time and attention on the body of the methods instead of the methods' signature, which contrast with previous studies. They also observed that experienced developers tend to revisit the body of the methods more often as the method size increases.

Other studies used fMRI technologies to observe how programmers comprehend short code snippets. For instance, Peitek and colleagues (Peitek et al., 2021) conducted a study using fMRI with 19 participants. Among the findings, the authors found that using simple metrics, such as LOC and McCabe, have good performance when predicting cognitive load.

8.3. Different forms of code complexities

Some studies have provided initial evidence that counting the number of structural elements may not always reflect cognitive complexity experienced by humans (Ajami et al., 2019; Jbara and Feitelson, 2017; Hermans, 2021), Indeed, these studies indicate that there are other forms of understanding/assessing understandability.

For instance, in the study of Ajami and colleagues (Ajami et al., 2019), the authors perceived that "if a program embodies certain business rules, a reader who does not know these rules will find it difficult to understand". In another study, Jbara and Feitelson (2017) studied repetitions of code segments (such as a chain of if statements). Such repetitions tend to be longer and, if measured by metrics like MCC, may also be seen as complex. However, the authors noticed that repetitive code is not harder to comprehend than the non-repetitive alternatives, which might be because repeated instances are easier once the initial ones are understood.

Similarly, Hermans in her book delves deeper on how the different coding development processes affect the cognitive process (Hermans, 2021). For instance, the author noticed that factors such as code smells or naming conventions could contribute significantly to how humans perceive and process code, influencing its overall understandability.

Therefore, knowing that understanding code involves a complex interplay of various cognitive processes beyond structural complexity metrics, CDD could be seen as a complementary tool to ease code understanding.

8.4. Early works on CDD

Finally, although there are some early works on CDD (e.g., Barbosa et al., 2021; Pinto and de Souza, 2022; de Souza and Pinto, 2020), none of these works assessed the developers' perception in dealing with CDD in practice. Our work extends these works by bringing a unique perception of the use of CDD from the trenches.

9. Conclusion

CDD is a coding design technique that aims to reduce the cognitive load that developers may face when reading code by limiting the number of programming constructs that could be used in a given code unit. Here we shared our experience in using CDD to build a Java-based online learning management tool at Zup Innovation, a Brazilian tech company. By combining our observations from the developers' work with data and metadata from the repositories, we could curate a set of findings on the CDD practice. We highlight that:

1. CDD could help software development teams to keep the size of the code units reasonably small, potentially helping to identify god classes earlier;
2. by having small code units, CDD also impacts testing practices; according to an engineer "since CDD leads to small code units, these code units are also easier to tests";
3. CDD brings more awareness for refactoring operations; instead of refactoring based on their intuitions, developers following CDD practices have now the evidence of when refactorings are really needed.

9.1. Future work

First, we plan to use CDD in the other services that compose Handora. This was not possible during this research, because the team working on them was not yet used to CDD. For instance, we are interested in understanding how CDD could help the development of the machine learning service (S6) and the frontend service (S5). Moreover, we did not use CDD for writing testing code. We understand that by limiting the size of production classes and methods, the testing classes may also be positively impacted. In the future, we plan to conduct experiments to understand to what extent it makes sense to use CDD in designing testing code.

Second, we observed that CDD helped the software development team to keep the classes under the defined limit. However, as a potential side effect, these small classes might have a more significant number of relationships. We then plan to understand whether these small classes require more navigation from the developer, which in turn could hinder program comprehension.

Third, we have outlined plans to conduct experiments involving varying sets of ICPs. In this research, we noticed that it is hard to come up with an effective set of ICPs, in particular, when considering the differing perspectives among team members. We seek to examine the relationship between CDD (and its different sets of ICPs) and metrics such as productivity or understandability.

Finally, we plan to build production-ready static analysis tools that support CDD. Using this tool, we plan to perform controlled experiments to understand to what extent the tool could help developers find classes worth refactoring according to CDD principles. We also plan to create a bot that computes the CDD metrics during the CI/CD process. We believe this bot could ease the CDD adoption in software-producing teams.

CRediT authorship contribution statement

Gustavo Pinto: Ideas, Formulation or evolution of overarching research goals and aims, Development or design of methodology, Application of statistical, Conducting a research and investigation process, Specifically performing the experiments, or data/evidence collection, Writing – original draft, Writing – review & editing, Management and coordination responsibility for the research activity planning and execution. **Alberto de Souza:** Ideas, Formulation or evolution of overarching research goals and aims, Critical review, commentary or revision, Management and coordination responsibility for the research activity planning and execution.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Professionals at Zup Innovation.

Data availability

The data that has been used is confidential.

Acknowledgments

We thank the reviewers for their helpful comments, and Zup Innovation to support our work. This work also supported by FAPESPA, Brazil (#053/2021) and CNPq, Brazil (#308623/2022-3).

References

Abid, N.J., Sharif, B., Dragan, N., Alrasheed, H., Maletic, J.I., 2019. Developer reading behavior while summarizing Java methods: Size and context matters. In: Proceedings of the 41st International Conference on Software Engineering. ICSE '19, IEEE Press, pp. 384–395, [Online]. Available: https://doi.org/10.1109/ICSE.2019.00052.

Ajami, S., Woodbridge, Y., Feitelson, D.G., 2019. Syntax, predicates, idioms - what really affects code complexity? Empir. Softw. Eng. 24 (1), 287–328, [Online]. Available: https://doi.org/10.1007/s10664-018-9628-3.

Aniche, M.F., Oliva, G.A., Gerosa, M.A., 2013. What do the asserts in a unit test tell us about code quality? A study on open source and industrial projects. In: Cleve, A., Ricca, F., Cerioli, M. (Eds.), 17th European Conference on Software Maintenance and Reengineering, CSMR 2013, Genova, Italy, March 5-8, 2013. IEEE Computer Society, pp. 111–120, [Online]. Available: https://doi.org/10.1109/CSMR.2013.21.

Barbosa, L., Pinto, V.H.S.C., de Souza, A.L.O.T., Pinto, G., 2021. To what extent cognitive-driven development improves code readability? In: ESEM '22: ACM / IEEE International Symposium on Empirical Software Engineering and Measurement, Helsinki, Finland, September 11-15, 2022. ACM, pp. 24:1–24:11.

Chidamber, S.R., Kemerer, C.F., 1994. A metrics suite for object oriented design. IEEE Trans. Softw. Eng. 20 (6), 476–493.

Chowdhury, S.A., Uddin, G., Holmes, R., 2022. An empirical study on maintainable method size in Java. In: IEEE/ACM 19th International Conference on Mining Software Repositories, MSR 2022, Pittsburgh, PA, USA, May 23-24, 2022. IEEE, pp. 252–264, [Online]. Available: https://doi.org/10.1145/3524842.3527975.

dos Santos, R.M., Gerosa, M.A., 2018. Impacts of coding practices on readability. In: Proceedings of the 26th Conference on Program Comprehension. ICPC '18, Association for Computing Machinery, New York, NY, USA, pp. 277–285, [Online]. Available: https://doi.org/10.1145/3196321.3196342.

Galster, M., Mitrovic, A., Malinen, S., Holland, J., 2022. What soft skills does the software industry *really* want? An exploratory study of software positions in New Zealand. In: Proceedings of the 16th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement. ESEM '22, Association for Computing Machinery, New York, NY, USA, pp. 272–282, [Online]. Available: https://doi.org/10.1145/3544902.3546247.

Gil, Y., Lalouche, G., 2017. On the correlation between size and metric validity. Empir. Softw. Eng. 22 (5), 2585–2611, [Online]. Available: https://doi.org/10.1007/s10664-017-9513-5.

Gill, G., Kemerer, C., 1991. Cyclomatic complexity density and software maintenance productivity. IEEE Trans. Softw. Eng. 17 (12), 1284–1288.

Halsted, M., 1977. Elements of Software Science. Elsevier, North-Holland New York.

Hermans, F., 2021. The Programmer's Brain: What Every Programmer Needs to Know About Cognition. Simon and Schuster.

Ivanković, M., Petrović, G., Just, R., Fraser, G., 2019. Code coverage at Google. In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 955–963.

Jbara, A., Feitelson, D.G., 2017. How programmers read regular code: a controlled experiment using eye tracking. Empir. Softw. Eng. 22 (3), 1440–1477, [Online]. Available: https://doi.org/10.1007/s10664-016-9477-x.

Kochhar, P.S., Xia, X., Lo, D., 2019. Practitioners' views on good software testing practices. In: Sharp, H., Whalen, M. (Eds.), Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2019, Montreal, QC, Canada, May 25-31, 2019. IEEE / ACM, pp. 61–70, [Online]. Available: https://doi.org/10.1109/ICSE-SEIP.2019.00015.

Lanza, M., Marinescu, R., 2007. Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems. Springer Science & Business Media.

Lavallée, M., Robillard, P.N., 2015. Why good developers write bad code: An observational case study of the impacts of organizational factors on software quality. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, Vol. 1. IEEE, pp. 677–687.

Le, D.M., Link, D., Shahbazian, A., Medvidovic, N., 2018. An empirical study of architectural decay in open-source software. In: 2018 IEEE International Conference on Software Architecture (ICSA). IEEE, pp. 176–17609.

Lefever, J., Cai, Y., Cervantes, H., Kazman, R., Fang, H., 2021. On the lack of consensus among technical debt detection tools. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP). IEEE, pp. 121–130.

Marcilio, D., Furia, C.A., Bonifácio, R., Pinto, G., 2020. SpongeBugs: Automatically generating fix suggestions in response to static code analysis warnings. J. Syst. Softw. 168, 110671, [Online]. Available: https://doi.org/10.1016/j.jss.2020.110671.

Marinescu, R., 2001. Detecting design flaws via metrics in object-oriented systems. In: Proceedings 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems. TOOLS 39. IEEE, pp. 173–182.

Martin, R.C., 1996. The open-closed principle. More C++ Gems 19 (96), 9.

Martin, R.C., Coplien, J.O., 2009. Clean Code: A Handbook of Agile Software Craftsmanship. Prentice Hall, Upper Saddle River, NJ [etc.], [Online]. Available: https://www.amazon.de/gp/product/0132350882/ref=oh_details_o00_s00_i00.

McCabe, T.J., 1976. A complexity measure. IEEE Trans. Softw. Eng. (4), 308–320.

Miller, G.A., 1956. The magical number seven, plus or minus two: Some limits on our capacity for processing information.. Psychol. Rev. 63 (2), 81.

Oizumi, W., Sousa, L., Oliveira, A., Carvalho, L., Garcia, A., Colanzi, T., Oliveira, R., 2019. On the density and diversity of degradation symptoms in refactored classes: A multi-case study. In: 2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE). pp. 346–357.

Olbrich, S.M., Cruzes, D.S., Sjø berg, D.I., 2010. Are all code smells harmful? A study of god classes and brain classes in the evolution of three open source systems. In: 2010 IEEE International Conference on Software Maintenance. IEEE, pp. 1–10.

Paas, F., Renkl, A., Sweller, J., 2004. Cognitive load theory: Instructional implications of the interaction between information structures and cognitive architecture. Instruct. Sci. 32 (1/2), 1–8.

Parnas, D.L., 1994. Software aging. In: Proceedings of 16th International Conference on Software Engineering. IEEE, pp. 279–287.

Parnas, D.L., Weiss, D.M., 1987. Active design reviews: principles and practices. J. Syst. Softw. 7 (4), 259–265.

Pawlak, R., Monperrus, M., Petitprez, N., Noguera, C., Seinturier, L., 2015. Spoon: A library for implementing analyses and transformations of java source code. Softw.: Pract. Exp. 46, 1155–1179, [Online]. Available: https://hal.archives-ouvertes.fr/hal-01078532/document.

Peitek, N., Apel, S., Parnin, C., Brechmann, A., Siegmund, J., 2021. Program comprehension and code complexity metrics: An fmri study. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE). IEEE, pp. 524–536.

Perry, D.E., Wolf, A.L., 1992. Foundations for the study of software architecture. SIGSOFT Softw. Eng. Notes 17 (4), 40–52, [Online]. Available: https://doi.org/10.1145/141874.141884.

Pinto, V.H.S.C., de Souza, A.L.O.T., 2022. Effects of cognitive-driven development in the early stages of the software development life cycle. In: Filipe, J., Smialek, M., Brodsky, A., Hammoudi, S. (Eds.), Proceedings of the 24th International Conference on Enterprise Information Systems, ICEIS 2022, Online Streaming, April 25-27, 2022, Vol. 2. SCITEPRESS, pp. 40–51.

Shaik, A., Manda, B., Prakashini, C., Deepthi, K., Reddy, C., 2010. Metrics for object oriented design software systems: a survey. J. Emerg. Trends Eng. Appl. Sci. 1 (2), 190–198.

Sharma, T., Mishra, P., Tiwari, R., 2016. Designite - A software design quality assessment tool. In: 2016 IEEE/ACM 1st International Workshop on Bringing Architectural Design Thinking Into Developers' Daily Activities (BRIDGE), pp. 1–4.

Silva, D., Tsantalis, N., Valente, M.T., 2016. Why we refactor? confessions of github contributors. In: Proceedings of the 2016 24th Acm Sigsoft International Symposium on Foundations of Software Engineering, pp. 858–870.

Sousa, L., Oliveira, A., Oizumi, W., Barbosa, S., Garcia, A., Lee, J., Kalinowski, M., de Mello, R., Fonseca, B., Oliveira, R., Lucena, C., Paes, R., 2018. Identifying design problems in the source code: A grounded theory. In: Proceedings of the 40th International Conference on Software Engineering. ICSE '18, Association for Computing Machinery, New York, NY, USA, pp. 921–931, [Online]. Available: https://doi.org/10.1145/3180155.3180239.

de Souza, A.L.O.T., Pinto, V.H.S.C., 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, pp. 776–778.

Sweller, J., 1988. Cognitive load during problem solving: Effects on learning. Cogn. Sci. 12 (2), 257–285.

Tsantalis, N., Ketkar, A., Dig, D., 2022. RefactoringMiner 2.0. IEEE Trans. Softw. Eng. 48 (3), 930–950.

Uchôa, A., Barbosa, C., Oizumi, W., Blenílio, P., Lima, R., Garcia, A., Bezerra, C., 2020. How does modern code review impact software design degradation? an in-depth empirical study. In: 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE, pp. 511–522.

Van Deursen, A., Moonen, L., Van Den Bergh, A., Kok, G., 2001. Refactoring test code. In: Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP2001). Citeseer, pp. 92–95.

Varela, A.S.N., Pérez-González, H.G., Martínez-Pérez, F.E., Soubervielle-Montalvo, C., 2017. Source code metrics: A systematic mapping study. J. Syst. Softw. 128, 164–197, [Online]. Available: https://doi.org/10.1016/j.jss.2017.03.044.

Wiese, E.S., Rafferty, A.N., Moseke, G., 2021. Students' misunderstanding of the order of evaluation in conjoined conditions. In: 29th IEEE/ACM International Conference on Program Comprehension, ICPC 2021, Madrid, Spain, May 20-21, 2021. IEEE, pp. 476–484, [Online]. Available: https://doi.org/10.1109/ICPC52881.2021.00055.

Yau, S.S., Tsai, J.J.-P., 1986. A survey of software design techniques. IEEE Trans. Softw. Eng. (6), 713–721.

Yu, Y.T., Lau, M.F., 2006. A comparison of MC/DC, MUMCUT and several other coverage criteria for logical decisions. J. Syst. Softw. 79 (5), 577–590.

Zage, W.M., Zage, D.M., 1993. Evaluating design metrics on large-scale software. IEEE Softw. 10 (4), 75–81.

Zhang, Y., Mesbah, A., 2015. Assertions are strongly correlated with test suite effectiveness. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering. In: ESEC/FSE 2015, Association for Computing Machinery, New York, NY, USA, pp. 214–224, [Online]. Available: https://doi.org/10.1145/2786805.2786858.

Gustavo Pinto is a research engineer at Zup Innovation and a part-time assistant professor of computer science at the Federal University of Pará. His research interests include interactions between people and code, spanning the areas of software engineering and programming language. Dr. Pinto received his Ph.D. from the Federal University of Pernambuco, in 2015. Contact him at gustavo.pinto@zup.com.br.

Alberto de Souza is the creator of the YouTube Channel "Dev eficiente" (https://www.youtube.com/c/deveficiente), and serves as the director of technology and education at Zup Innovation. His research interests include education as a driver for performance improvement and software quality. Contact him at alberto.tavres@zup.com.br.