Cognitive-Driven Development Helps Software Teams to Keep Code Units Under the Limit!

Gustavo Pinto
Assistant professor at UFPA.br
Head of research at ZUP.com.br

@gustavopinto
How to reason about this?

How to test this?

How to refactor this?

How to spot the bug?
If we want to keep making changes...

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.

https://twitter.com/kentbeck/status/1354418068869398538
CDD is a **simple** approach for chunking and slicing
Cognitive-Driven Development (CDD)

- CDD aims to **reduce the developers’ cognitive load** during coding activities.
- CDD does so by **posing a limit** on the number of items developers could use at once (at a class or file).
The Magical Number 7

- We are only able to process 7 (+/- 2) units of information in short-term memory.
- As we receive more information simultaneously, we lose the ability to process it (and we tend to make mistakes).
The Magical Number 7

- We are only able to process 7 (+/- 2) units of information in short-term memory.

- As we receive more information simultaneously, we lose the ability to process it (and we tend to make mistakes).
CDD in a nutshell

- CDD provides a **clear limit** indicating how much a code unit could grow
- Every class over the limit **must be** refactored
• Online training platform by Zup
• Composed by 5 services:
  ○ Core service (Java)
  ○ Search service (Java)
  ○ Menu service (Java)
  ○ Frontend Service (TypeScript)
  ○ ML service (Python)
● Online training platform by Zup

● Composed by 5 services:
  ○ Core service (Java)
  ○ Search service (Java)
  ○ Menu service (Java)
  ○ Frontend Service (TypeScript)
  ○ ML service (Python)
CDD impacts the size of the classes

- CDD seems to help keep the classes small, even as the product evolves (almost linearly)
- Developers concur that the size of the classes are probably due to CDD
CDD impacts the size of the methods

- Maintenance effort is positively correlated with method length.
- "Developers should strive to keep their methods within 24 SLOC"

Abstract

Software maintenance has long been identified as challenging for software engineers [1] and maintenance costs often exceed initial development costs [2]. Consequently, both newcomers and practitioners are keen to limit future maintenance efforts from the onset of a software project to facilitate early logging, and to avoid project optimizations [21, 34, 40, 59, 95, 99]. These maintenance efforts are even more so in early stages before the code is finalized.

One of the most widely cited maintenance indicators and change concerns is size (e.g., LOC) and complexity (e.g., McCabe, 34, 36, 42, 58, 66). These metrics capture the code component (e.g., method, classes or module), and the changes are largely a function of experience. Early identification and optimization of such components could help reduce future maintenance efforts. Therefore, small, simple, and well-organized code components are generally available and allow for easier system evolution [78].

In this paper, we investigate the relationship between maintenance indicators and buggy classes. We identify, i.e., buggy classes and compare them with non-bug, to understand what made the buggy classes different. We use 25+ buggy classes of a medium- to large-scale system (1,000,000 LOC) and seven of them have been identified by previous research as buggy. We show that buggy classes are significantly larger than non-bug classes, have a significantly lower cohesion, have significantly lower complexity (McCabe complexity), have significantly lower coupling (coupling to external classes), and are significantly more used.

Introduction

Maintenance effort correlates with method length [91]. For example, an individual with 10 years of experience in Java is expected to write 40 methods of 25 LOC each. This is equivalent to 1,000 lines of code (LOC) per year. However, for the same individual, the corresponding effort for a 100 LOC method would be 400 LOC per year. This is a significant increase in maintenance effort and is likely to be a major contributor to the increasing cost of maintenance.

Methodology

We conduct a case study of 25 buggy classes and seven non-bug classes. The buggy classes are identified by previous research as buggy. We compare these buggy classes with non-bug classes to understand what makes them different. We use seven maintenance indicators, including size (LOC), complexity (McCabe complexity), coupling to external classes, and usage frequency.

Results

We find that buggy classes are significantly larger than non-bug classes, have significantly lower cohesion, have significantly lower complexity (McCabe complexity), have significantly lower coupling (coupling to external classes), and are significantly more used. These results indicate that buggy classes are significantly different from non-bug classes in terms of size, complexity, coupling, and usage.

Conclusion

Our results suggest that buggy classes are significantly different from non-bug classes in terms of size, complexity, coupling, and usage. This indicates that buggy classes are more complicated and require more maintenance effort. Therefore, it is important to identify buggy classes early in the development process and to take appropriate measures to prevent them from becoming bigger and more complicated.

@zupinnovation
CDD impacts the **size of the methods**

- 92% of the handora's methods are under 24 SLOC (6.8, on average)
- “Every *unit of code is impacted*, because we know what the limit is and what goes into that limit.”
CDD impacts testing code

- On average, a testing method has ~8 SLOC
- “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”.
- There are ~1.3 assertions per method (no method without assertion)

| #  | SLOC | Methods | Coverage |
|----|------|---------|----------|
| S1 | 7.6k | 215     | 71%      |
| S2 | 1.3k | 41      | 61%      |
| S3 | 5.2k | 128     | 64%      |
Cognitive Driven Development

Findings

- CDD seems to help designing small classes
- CDD seems to help designing small methods
- CDD seems to help designing small testing methods
Cognitive Driven Development

Findings
- CDD seems to help designing small classes
- CDD seems to help designing small methods
- CDD seems to help designing small testing methods

Challenges
- Still requires manual effort
- We need better tools
- How to ease CDD adoption?
