Pontifícia Universidade Católica
do Rio de Janeiro

Vinícius Souza Martins

Eye-Tracking the Impact of Code Smells on Developer Comprehension

Dissertação de Mestrado

Dissertation presented to the Programa de Pós–graduação em Informática, do Departamento de Informática of PUC-Rio in partial fulfillment of the requirements for the degree of Mestre em Informática.

Advisor : Profª Juliana Alves Pereira
Co-advisor: Prof. Alessandro Fabricio Garcia

Rio de Janeiro
April 2025

Pontifícia Universidade Católica
do Rio de Janeiro

Vinícius Souza Martins

Eye-Tracking the Impact of Code Smells on Developer Comprehension

Dissertation presented to the Programa de Pós–graduação em Informática of PUC-Rio in partial fulfillment of the requirements for the degree of Mestre em Informática. Approved by the Examination Committee.

Profª Juliana Alves Pereira
Advisor
Departamento de Informática – PUC-Rio

Prof. Alessandro Fabricio Garcia
Co-advisor
Departamento de Informática – PUC-Rio

Prof. Anderson Gonçalves Uchôa
Departamento de Informática – UFC

Prof. Eduardo Magno Lages Figueiredo
Departamento de Informática – UFMG

Rio de Janeiro, April 30th, 2025

Vinícius Souza Martins

I am an MSc student in Computer Science at Pontifical Catholic University of Rio de Janeiro (PUC-Rio, Brazil). I holds a bachelor's degree in Naval Science from the Brazilian Naval Academy and served aboard ships of the Brazilian Navy before seizing the opportunity to advance his knowledge in Software Engineering.

Bibliographic data

Martins, Vinícius Souza

Eye-Tracking the Impact of Code Smells on Developer Comprehension / Vinícius Souza Martins; advisor: Juliana Alves Pereira; co-advisor: Alessandro Fabricio Garcia. – Rio de janeiro: PUC-Rio, Departamento de Informática, 2025.

v., 86 f: il. color. ; 30 cm

Dissertação (mestrado) - Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática.

Inclui bibliografia

1. Software Engineering – Teses. 2. Engenharia de software;. 3. Qualidade do código;. 4. Code smells;. 5. Esforço cognitivo;. 6. Eye tracker;. 7. Revisão de código;. 8. Compreensão de código.. I. Pereira, Juliana Alves. II. Garcia, Alessandro Fabricio. III. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. IV. Título.

CDD: 004

Acknowledgments

I would like to begin by expressing my deepest gratitude to my family, whose encouragement and understanding have made all the difference. To my wife, Julianne, thank you for your unwavering support, patience, and for standing by me through moments of stress and quiet; your belief in my work has been my greatest source of motivation. I also extend my heartfelt thanks to my father, Renato, who has always supported me despite our occasional disagreements. To my sister, your positivity and willingness to help at any moment have been a constant reminder that I am never alone on this journey. Above all, I want to acknowledge my mother, who, together with God above, continues to watch over and guide me every step of the way.

I owe a special debt of gratitude to my advisors, Prof. Dr. Juliana Alves and Prof. Dr. Alessandro Garcia, for their invaluable guidance. Prof. Dr. Juliana, your patience in correcting my mistakes and your keen insight helped me navigate challenging moments of writing both this dissertation and related articles. To Prof. Dr. Alessandro, thank you for the discussions, feedback, and willingness to help whenever I needed it. I am also grateful to Prof. Mograbri for generously providing the eye tracker that made this study possible, and to Prof. Kalinowski for recommending me for the Master's program.

My thanks also go to my colleagues at the AISE Lab and UFC, especially to Prof. Anderson Uchôa, for their collaboration and support throughout this research. Furthermore, I want to acknowledge PUC-Rio for the excellent resources and environment that allowed me to concentrate on my studies and grow as a researcher. Each of you has played a part in this achievement, and I extend my gratitude from the bottom of my heart.

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001.

Thank you.

Abstract

Martins, Vinícius Souza; Pereira, Juliana Alves (Advisor); Garcia, Alessandro Fabricio (Co-Advisor). **Eye-Tracking the Impact of Code Smells on Developer Comprehension**. Rio de Janeiro, 2025. 86p. Dissertação de mestrado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Code smells negatively impact software maintainability and evolution by harming developers' ability to comprehend programs effectively. This dissertation investigates how code smells affect developers' program comprehension, analyzing their reading patterns, visual focus using eye-tracking technology, and qualitative feedback. Key eye-tracking metrics, including fixation duration and fixation count, are leveraged to quantify the cognitive effort required to analyze well-structured versus poorly structured code snippets. Qualitative analysis of developers' explanations provides insights into the perceived difficulty and comprehension strategies. By analyzing these metrics across different types of code smells – such as Data Class, Long Method, and Feature Envy – we identify which smells demand more cognitive effort from developers. This dissertation contributes to the field of software engineering by providing empirical evidence on the impact of code smells on software comprehension and proposing practical improvements in Integrated Development Environments (IDEs) to reduce developers' cognitive load when dealing with complex code.

Keywords

Software engineering; Code quality; Code smell; Cognitive effort; Eye tracker; Code review; Code comprehension.

Resumo

Martins, Vinícius Souza; Pereira, Juliana Alves (Orientador); Garcia, Alessandro Fabricio (Co-Orientador). **Acompanhamento Visual do Impacto de Code Smells na Compreensão do Desenvolvedor**. Rio de Janeiro, 2025. 86p. Dissertação de Mestrado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Code smells impactam negativamente a manutenibilidade e a evolução do software, prejudicando a capacidade dos desenvolvedores de compreender o código de forma eficaz. Esta dissertação investiga como *code smells* afetam a compreensão do código pelos desenvolvedores, analisando seus padrões de leitura, foco visual usando tecnologia de rastreamento ocular e feedback qualitativo. Métricas-chave de rastreamento ocular, incluindo média das durações e contagem de fixações, são utilizadas para quantificar o esforço cognitivo necessário para analisar trechos de código bem estruturados *versus* mal estruturados. A análise qualitativa das explicações dos desenvolvedores fornece *insights* sobre a dificuldade percebida e as estratégias de compreensão. Ao analisar essas métricas em diferentes tipos de *code smells* — como *Data Class*, *Long Method* e *Feature Envy* — identificamos quais exigem mais esforço cognitivo dos desenvolvedores. Esta dissertação contribui para o campo da engenharia de software, fornecendo evidências empíricas sobre o impacto dos code smells na compreensão do software e propondo melhorias práticas em Ambientes de Desenvolvimento Integrados (IDEs) para reduzir a carga cognitiva dos desenvolvedores ao lidar com código complexo.

Palavras-chave

Engenharia de software; Qualidade do código; Code smells; Esforço cognitivo; Eye tracker; Revisão de código; Compreensão de código.

Table of contents

List of figures

List of tables

List of Abreviations

AFD	Average Fixation Duration
AOI	Area of Interest
CAAE	Certificate of Presentation for Ethical Appreciation
CEP	Research Ethics Committee
CDD	Cognitive Driven Development
DC	Data Class
DECOR	Detection of Code Smells by Rules
FC	Fixation Count
FE	Feature Envy
GQM	Goal-Question-Metrics
Hz	Hertz
IDE	Integrated Development Environment
IQR	Interquartile Range
JDeodorant	Java Code Smell Detection and Refactoring Tool
LM	Long Method
MFD	Mean Fixation Duration
MLCQ	Madeyski Lewowski Code Quest (dataset)
ms	Milliseconds
OBS	Open Broadcaster Software
PMD	Programming Mistake Detector
RQ	Research Question
SBES	Brazilian Symposium on Software Engineering
SLR	Systematic Literature Review
SUPR-Q	Standardized User Experience Percentile Rank Questionnaire
TCLE	Free and Informed Consent Form
TOSEM	Transactions on Software Engineering and Methodology (journal)
TX300	Tobii TX300 Eye Tracker
XML	Extensible Markup Language
iPlasma	Code Smell Detection Tool
iTrace	Integrated Traceability Environment
iTrace-Toolkit	iTrace Data Processing Toolkit

1
Introduction

Software plays a critical role in modern society, permeating nearly every aspect of daily life. As software systems evolve, maintenance becomes an essential activity to ensure their functionality, adaptability, and quality over time [1]. Software maintenance tasks include modifying existing features, implementing new functionalities, and fixing defects, all of which require an in-depth understanding of the source code. A major challenge in this context is the presence of code smells, which are indicators of suboptimal design and implementation choices that can negatively impact the comprehensibility of software systems [2].

Code smells, such as *Long Method*, *Feature Envy*, and *Data Class*, have been widely recognized as factors that hinder code comprehension by introducing unnecessary complexity and making it more difficult for developers to navigate and modify the codebase [2, 3]. Poorly structured code can lead to increased maintenance effort, higher costs, and decreased developer productivity, ultimately affecting the overall quality and sustainability of the software product [4]. Addressing code smells through their timely identification is crucial for enhancing maintainability and reducing the effort required to comprehend complex code structures [2].

Eye-tracking technology provides valuable insights into developers' reading patterns and visual attention during code analysis, offering an objective means to assess the cognitive effort exerted when dealing with smelly code [5, 6]. By analyzing eye-tracking metrics, such as fixation count and duration, researchers can identify which code elements demand more cognitive effort and understand the challenges faced by developers in interpreting smelly code [7]. Thus, this dissertation seeks to employ eye-tracking technology to analyze developers' responses to code snippets with and without code smells, ultimately contributing to the understanding of which and how certain code structures affect developers' program comprehension. This involves not only quantifying cognitive effort through metrics but also exploring developers' reasoning and subjective experiences during comprehension tasks, contributing to a deeper understanding of which and how certain code structures affect developers' program comprehension. Also, this work adopts open science and study replicability practices. All data, scripts and materials used in the research are available for other researchers who wish to explore or extend the results[8, 9].

1.1 Problem Statement and Limitations of Related Work

The presence of code smells in software systems poses significant challenges to program comprehension and maintenance. Code smells, such as *Long Method*, *Feature Envy*, and *Data Class*, can obscure the logical structure of the code, making it more difficult for developers to understand and modify software effectively. Despite the extensive research conducted on the detection and classification of code smells, there is a lack of empirical evidence on how these smells impact developers' cognitive effort and reading behavior. Understanding how developers interact with smelly code and the cognitive load imposed is crucial to developing effective strategies for refactoring and software quality improvement.

This dissertation aims to address the following research problem:

"How do different types of code smells impact the cognitive effort and perception of developers during code comprehension tasks?"

To address this research problem, we looked at the limitations in prior research to underscore the need for this dissertation's investigations. Drawing from systematic analyses of state-of-the-art studies, we identified six limitations addressed by our study:

1. **Lack of Integrated Quantitative and Qualitative Analysis**: To the best of our knowledge, there is a lack of approaches combining quantitative eye-tracking data with qualitative developer insights to understand code smell comprehension. Thus, we investigate how developers' own perceptions and classifications of code relate to their cognitive load.

2. **Overreliance on Synthetic/Automated Code Smell Datasets**: Many studies rely on synthetic or automatically labeled datasets that may not reflect the complexity of real-world software development scenarios [10].

3. **Overreliance on static code metrics**: Traditional static code metrics fail to capture dynamic cognitive processes developers engage with while reviewing code, as presented by Da Costa et al.[11].

4. **Limited understanding of cognitive impacts of code smells**: Prior work focused on long-term maintenance impacts but neglected real-time cognitive processes during code comprehension, which restricts understanding the impact of developers' cognitive effort for processing smelly codes during analysis.

5. **Lack of code smell-specific eye-tracking studies**: While the literature includes numerous eye-tracking studies in software engineering, none were found that specifically investigated code smells.

6. **Gap in application of eye-tracking tools for smell analysis**: While exists tools, like the iTrace that we used in this dissertation, to the best of our knowledge, there is no studies that utilize these tools to explore cognitive effort and reading behavior in code with smells.

This dissertation addresses the identified research gaps through two sequential studies with eye-tracking investigations that integrate experimental software engineering with cognitive effort analysis. In the first study, *Eyes on Code Smells: Analyzing Developers' Responses During Code Snippet Analysis* that was published at *38th Brazilian Symposium on Software Engineering (SBES)*, present in Chapter 3, was implemented a controlled experimental protocol where 12 developers engaged in analyzing 13 real-world Java code snippets without or with smells (data class, long method or feature envy) selected from the MLCQ dataset that was manually validated by experience developers using a Tobii TX300 eye tracker Eclipse IDE with iTrace plugin and iTrace-Toolkit. Participants performed code comprehension tasks while fixation metrics (Average Fixation Duration and Fixation Count) were quantitatively analyzed against expert-validated smell labels to understand the cognitive effort by these developers. Although post-task surveys captured subjective impressions, the analysis focused exclusively on objective eye-tracking data.

Next, the second study, *Reading between the Smells: Eye-Tracking Developer Responses to Code Smells* submitted to *Transactions on Software Engineering and Methodology (TOSEM)*, present in Chapter 4, do an in-depth analysis with qualitative and quantitative data with focus on how developer-perceived smells rather than pre-defined labels. The study was conducted with 27 developers. They examined the same artifacts presented in the first study. For qualitative analysis, it is explored 132 developers' explanations about how they perceived difficulty correlates with their smell classification. For quantitative analysis, we perform the same analysis for cognitive effort present in the first study with statistics tests. Moreover, we also explore the mental models and reading patterns developers use during program comprehension tasks.

1.2 Main Contributions

The main contributions of this research are as follows:

Empirical Evidence on Cognitive Effort with Code Smells. This dissertation provides detailed empirical evidence on the cognitive impact of different types of code smells. In the study presented in Chapter 3, we used eye-tracking metrics, such as *fixation count* and *avarage fixation duration*, to quantify the cognitive effort required to analyze code snippets with and without code smells. The results show that *Long Method* and *Feature Envy* impose a significantly higher cognitive load compared to *Data Class*, with *Long Method* being the most demanding due to its structural

complexity. In Chapter 4, we confirmed that this cognitive load is correlated with developers' perceptions of the presence of code smells, regardless of the accuracy of their classifications, highlighting the relevance of subjective impressions in the comprehension process.

Insights into Developers' Visual Attention and Reading Patterns. The analysis of eye-tracking data revealed distinct patterns of visual attention and reading. In Chapter 3, we identified that developers spend more time fixating on elements such as names and control structures when analyzing *Long Method* and *Feature Envy*, indicating that these elements are crucial for comprehension. In Chapter 4, we observed that *Long Method* and *Feature Envy* lead to bottom-up mental models, where developers adopt a detailed, line-by-line approach, while *Data Class* encourages more organized and sectional reading patterns. These findings enhance the understanding of how developers visually interact with code smells, paving the way for targeted IDE and refactoring tool improvements.

Recommendations for Improving Integrated Development Environments (IDEs). The results of both studies generated practical recommendations for enhancing IDEs. In Chapter 3, we suggested that tools could use eye-tracking data to propose decomposition strategies for *Long Methods* and visualize dependencies in *Feature Envy*. In Chapter 4, we added the recommendation to prioritize the refactoring of code smells that most impact comprehension, such as *Long Method*, based on developers' perceptions of difficulty. These improvements aim to reduce cognitive load and enhance productivity when dealing with complex code.

Pioneering the Use of Eye Tracking for Code Smell Analysis. This research is pioneering in applying eye-tracking technology to investigate the relationship between code smells and program comprehension. In Chapter 3, we introduced a novel experimental framework that combines software artifact analysis with physiological metrics, establishing a quantitative benchmark for the complexity induced by code smells. Chapter 4 complements this approach with qualitative data, enriching the analysis with developers' perspectives. The proposed framework can serve as a valuable tool for practitioners and researchers to validate the impact of other types of code smells, enabling a more systematic and evidence-based approach to improving code quality and developer experience. Moreover, it offers a new paradigm for assessing cognitive effort in real time.

Foundational Work for Future Research. The studies establish a foundation for future investigations by proposing the integration of additional metrics, such as pupil dilation, and by identifying the need to explore other types of code smells. The progression from quantitative (Chapter 3) to qualitative (Chapter 4) suggests

a path for more holistic research on code comprehension, contributing to open new avenues for interdisciplinary research combining software engineering with cognitive science and human-computer interaction.

Advancing the Understanding of Code Smell Refactoring. This dissertation advances the understanding of refactoring by empirically demonstrating that *Long Method* and *Feature Envy* are among the most cognitively demanding and perceptually complex code smells. These smells consistently impose a higher mental load on developers, affecting their ability to comprehend and navigate code efficiently. The combined quantitative and qualitative evidence from both studies underscores the importance of prioritizing these smells during refactoring efforts. It also highlights the need for development best practices and supportive tools that can proactively identify and mitigate such smells, ultimately enhancing code maintainability and reducing developer fatigue. These findings have the potential to influence both academic research and industry standards, advancing the state of the art in software engineering.

1.3 Summary of Methodology and Key Finding

This section presents the key elements of the experimental design, analytical procedures, and the most significant results, offering the reader a comprehensive understanding of how the research objectives were pursued and achieved.

1.3.1 Methodological Overview

This dissertation employed a two-stage empirical approach to investigate the impact of code smells on developer comprehension. Our approach integrates both quantitative and qualitative analyses grounded in controlled experimentation. In the first study, a controlled experiment was conducted with 12 developers, who analyzed 13 real-world Java code snippets - containing or not containing code smells such as Long Method, Feature Envy, and Data Class. These smells were selected from the manually validated MLCQ dataset. Using a Tobii TX300 eye tracker in conjunction with the Eclipse IDE equipped with the iTrace plugin and iTrace-Toolkit, we recorded detailed eye-tracking metrics, focusing primarily on Average Fixation Duration (AFD) and Fixation Count (FC). These metrics were used to objectively quantify the cognitive effort required to analyze each code snippet. The experimental procedure included a preparatory introduction, the presentation and analysis of code snippets, and post-task questionnaires to capture subjective developer impressions. In the second study, we expanded the number or subjects with 27 participants and incorporated qualitative analysis by examining 132 textual explanations provided by the developers. This phase focused on how participants

perceived and classified code smells, their reported difficulty, and the mental models and reading patterns employed during comprehension. Statistical analyses were conducted to compare cognitive effort across smell types and to relate subjective difficulty to objective eye-tracking data. These analyses offered a comprehensive view of the interplay between code structure, developer perception, and comprehension strategy.

1.3.2 Key Results and Contributions

The results revealed that code smells exert a impact on both the cognitive load and reading strategies of developers. Quantitative analysis demonstrated that Long Method and Feature Envy significantly increase cognitive effort, as evidenced by higher fixation durations and counts, compared to Data Class, which demanded less cognitive resource allocation. Notably, Long Method was identified as the most cognitively demanding smell, attributed to its structural complexity and the necessity for more exhaustive, line-by-line reading approaches. Feature Envy also posed substantial challenges, especially when dependencies were distributed across multiple contexts, making their correct identification difficult for many participants. In contrast, Data Class smells elicited more organized and sectional reading patterns with lower cognitive load. Qualitative analysis further indicated that developers' subjective perceptions of difficulty were often aligned with increased eye-tracking metrics, regardless of their accuracy in smell identification. The study also identified specific visual attention patterns. For instance, there was increased fixation on code elements such as function names and control structures in the presence of more complex smells. These findings highlight the relevance of considering both objective physiological data and subjective developer feedback in assessing the impact of code smells. This dissertation's empirical evidence lays the groundwork for improvements in IDE tooling, prioritization of refactoring activities, and further research into developer cognition, ultimately contributing to enhanced software quality and maintainability.

1.4 Dissertation Structure

This dissertation is structured into five chapters, each addressing different aspects of the study conducted to investigate the impact of code smells on developer comprehension using eye-tracking technology. The organization of the dissertation is designed to provide a logical progression from background concepts to empirical findings and conclusions, ensuring a comprehensive understanding of the research problem and its contributions.

Chapter 2 presents the background and related works. The background introduces fundamental concepts related to code smells, cognitive effort in software

comprehension, and eye-tracking technology. The related works section reviews existing studies on code smells and eye-tracking applications in software engineering.

Chapter 3, titled *Eyes on Code Smells: Analyzing Developers' Responses During Code Snippet Analysis*, presents the first study conducted in this research. This chapter details the study design, data collection procedures, and analysis methods employed to understand how developers respond to code snippets with and without code smells. It also discusses key findings related to cognitive effort, reading patterns, and developers' perceived difficulty in comprehending smelly code, providing empirical evidence on the impact of code smells.

Chapter 4, titled *Reading between the Smells: Eye-Tracking Developer Responses to Code Smells*, builds upon the findings from the previous chapter by incorporating qualitative analyses of developers' explanations and responses. This study investigates the relationship between developers' reading behavior, cognitive load, and their ability to accurately identify code smells. The chapter provides a deeper understanding of how different code smells influence comprehension strategies and mental models, offering insights into the challenges developers face when analyzing smelly code.

Chapter 5 presents the conclusions and future work, summarizing the key contributions of the dissertation and discussing their implications for software engineering research and practice. It outlines the limitations of the study and suggests directions for future research, such as exploring additional code smells, incorporating different developer experience levels, and refining eye-tracking methodologies for code comprehension analysis.

2
Background and Related Studies

This chapter presents the background of this dissertation. Section 2.1 introduces basic concepts of code smells, and specific types of code smells as well as discusses related work in the field. It discusses the definition and classification of code smells, highlighting the three smell types analyzed in this study: *Long Method*, *Feature Envy*, and *Data Class*. Furthermore, Section 2.2 explores how eye-tracking technology is utilized to measure cognitive effort and attention allocation in software engineering tasks. Next, Section 2.3 discusses existing studies on code comprehension and eye-tracking applications in software engineering, synthesizing foundational research and recent advancements to contextualize the current state-of-the-art in the field. Finally, Section 2.4 concludes this chapter.

2.1
Code Smells

Code smells are structural or design patterns in software systems that indicate potential quality issues, often leading to increased maintenance complexity. Introduced by Fowler et al. [2], code smells are symptoms of violations of object-oriented design principles, such as high coupling, low cohesion, or functional redundancy. They hinder code evolution, raising maintenance costs and defect risks [12].

According to the literature, code smells negatively affect software maintainability and evolution [13, 14]. Empirical studies associate their presence with challenges in code comprehension, increased cognitive load for developers, and higher error-proneness during modifications [15]. For instance, Long Method are frequently cited as barriers to readability and modularity [16].

They can also be classified according to their granularity, reflecting the scope and impact of the design problems they indicate within a software system. In the literature, code smells are typically categorized into different levels of granularity: implementation smells (method level), design smells (class level), and architecture smells (component or system level) [17]. In this study, we focus on method-level and design-level smells. Method-level smells, such as Long Method and Feature Envy, represent the smallest granularity and typically affect code readability and maintainability at a localized scope [17]. Class-level smells, such as God Class and Data Class, have a broader impact and often indicate violations of object-oriented design principles [17]. These different granularities help developers understand the severity and scope of potential design issues.

2.1.1 Types of code smells

In this study, we focus on the following three types of code smells [18]:

- *Long method:* a method with many lines of code and multiple responsibilities. It may hinder code comprehension by overwhelming developers with excessive information and multiple concerns in a single code block. Conversely, they might enhance comprehension by keeping related concerns together, reducing the need to navigate between multiple methods.
- *Feature envy:* a method that shows more interest in data of other class(es) than in the one in which it is currently located. It could harm code comprehension by violating the principle of encapsulation, increasing class coupling and making it harder to understand each class's responsibilities. Conversely, it might improve comprehension by centralizing related operations within the calling method, which might reduce the need for navigation across multiple classes when properly designed.
- *Data class:* a class that only serves as a container for data fields. This smell may negatively impact code comprehension by encouraging procedural-style programming in an object-oriented context, leading to a disconnect between data and behavior. On the other hand, they could enhance comprehension by providing a clear and simple structure for data organization, making it easier to understand the core entities in the system.

We selected these three types of smells as *(i)* they are quite different from each other in terms of structural problems they represent; *(ii)* these different structural problems may stimulate the experiment participant in a wide variety of different ways; *(iii)* choosing more smell types would make our experiment too complex – for instance, Data Class' structural may encourage holistic class-level analysis, while Feature Envy's inter-class dependencies may force developers to track cross-class method calls; and *(iv)* complex experiments tend to make the subjects feel very tired, demotivated or stressed, which would unavoidably interfere in the results. In future research, studies can replicate our experiments using other smell types.

2.1.2 Impacts of Code Smells on Software Quality

Code smells negatively impact software quality, particularly in maintainability and developers' cognitive effort. Empirical studies, such as those by Yamashita and Moonen [19], demonstrate that classes or methods with code smells (*e.g.*, God Class, Feature Envy) are strongly associated with maintenance challenges, including increased defect rates and higher effort during code modifications. For instance, God Class and Feature Envy were frequently linked to code comprehension difficulties and unintended side effects during evolution due to their widespread dependencies [19].

Metrics-based studies, such as those by Marinescu [12], highlight that design flaws like God Class violate fundamental principles such as cohesion and coupling, enabling targeted detection of problematic code structures. These violations are quantified through composite metrics, facilitating the identification of classes requiring design improvements. The study presented by Palomba et al. [20] demonstrates that methods affected by Feature Envy tend to co-change more frequently with methods of external classes, suggesting excessive reliance on external data. This behavior aligns with the definition of Feature Envy, where methods violate encapsulation principles, potentially increasing maintenance challenges due to heightened interdependencies.

Furthermore, code smells significantly affect developer productivity. Yamashita and Moonen [19] observed that systems with smells like God Class required substantially more effort to modify compared to smell-free code, with a higher likelihood of defect introduction. This aligns with findings from the SLR, which emphasizes that smell such Long Method are among the most detrimental to maintenance efficiency, partly due to their prevalence in large-scale systems [21]

2.1.3 Detection of Code Smells

The detection of code smells can be performed manually or automatically. Manual detection involves code inspection by developers, who identify smells based on their expertise and domain knowledge. However, this approach is impractical for large systems due to the time and effort required.

Automated detection leverages static code analysis techniques, including software metrics, machine learning algorithms, and search-based methods. Widely adopted tools include *DECOR* [22], which uses domain-specific rules for smells like *Blob* and *Spaghetti Code*; *JDeodorant* tool [23], which specializes in refactoring opportunities for smells such as *Feature Envy*; and *iPlasma* [24], a metrics-based tool for detecting design flaws. Popular metric-based tools are *PMD* [25], *Checkstyle* [26], and *SonarQube* [27] that apply predefined thresholds to metrics such as cyclomatic complexity and cohesion to identify smells. Moreover, Machine Learning (ML) approaches [28], often implemented with frameworks like *Weka* or libraries such as *scikit-learn*, employ algorithms like decision trees and support vector machines to classify code smells dynamically.

The problem is that automated detection faces challenges, like subjectivity in smell definitions, lack of consensus on metric thresholds, and variability in effectiveness across different smell types and system contexts. These limitations highlight the need for hybrid approaches combining metrics, historical data, and visualization to improve accuracy.

2.2 Eye Tracker

Eye-tracking technology offers a deep understanding of how people interact with visual elements. They collect data on how they navigate reading material [29] and respond to visual prompts when searching [30]. This technology is particularly helpful for understanding the cognitive processes involved in comprehension and problem solving. As cognitive functions direct one's gaze, analyzing eye movements can offer valuable insights into the cognitive efforts employed during various software engineering activities. This aspect of eye tracking makes it a powerful tool for studying how individuals interact cognitively with different stimuli. The way someone moves their eye gaze can reveal much about their thought process.

The relation between eye gaze and cognitive processing is based on two assumptions from the theory of reading: the immediacy assumption and the eye-mind assumption [31]. The immediacy assumption proposes that interpretation of the stimuli begins immediately as a participant sees it, *e.g.*, as soon as a reader reads a word. The eye-mind assumption states that participants fix their attention only on the part of the stimulus that is being processed currently [31]. These two assumptions are the foundation of how eye gaze represents the participant's cognitive processes. Eye gaze data indicate both the target of the participant's attention and the effort (or lack thereof) and length of time used to understand the stimulus. Furthermore, based on physiological studies [32], psychologists assume that participants do not have conscious control over many attributes of their eye gaze, *e.g.*, their pupil size, other than the location of their attention.

In the context of software development, an eye tracker becomes a powerful tool for assessing a developer's attention and cognitive processes. By capturing the precise gaze and movements of the eyes, an eye tracker allows researchers and developers to understand how individuals interact with visual stimuli on a screen [29, 30]. By analyzing which elements attract the most attention, developers can gain a deeper understanding of the visual hierarchy within a codebase, helping prioritize essential components and optimize code readability.

When working with an eye tracker, we can analyze several metrics, such as saccade, pupil dilation, constriction, areas of interest, and fixation [33]. For this study, we will focus on developers' fixations and, from there, explore how they analyze code snippets with or without code smells. Fixations are the areas of the stimulus where the participant's visual attention is concentrated, leading to cognitive processes. Most fixations last between 100 and 600 milliseconds, but this varies greatly based on context and other relevant factors [33].

As a practical example of the application of the eye-tracking metrics adopted in this study, consider the code snippet presented below, which was included in the experiment to evaluate the cognitive effort of developers.

```
@AutoValue.Builder
abstract static class Builder<T> {
    abstract Builder<T> setHosts(List<String> hosts);
    abstract Builder<T> setPort(Integer port);
    abstract Builder<T> setKeyspace(String keyspace);
    abstract Builder<T> setEntity(Class<T> entity);
    abstract Builder<T> setUsername(String username);
    abstract Builder<T> setPassword(String password);
    abstract Builder<T> setLocalDc(String localDc);
    abstract Builder<T> setConsistencyLevel(String
       consistencyLevel);
    abstract Builder<T> setMutationType(MutationType
       mutationType);
    abstract Write<T> build();
}
```

The presented snippet exhibits typical characteristics of the *Data Class* smell. During the experiment, participants analyzed similar code snippets while their visual interactions were recorded using metrics such as Fixation Count (FC) and Average Fixation Duration (AFD). These metrics allowed the identification of reading patterns and the estimation of the cognitive load required to comprehend this type of structure.

2.3 Related Studies

In this section, we conducted a narrative literature review based on structured keyword searches in selected scientific databases. The process involved defining specific search strings relevant to *eye trackers* and *code comprehension*. We reviewed the literature over the past ten years (2014-2024), aiming to identify research gaps and potential contributions of this study. Our goal is to understand the state-of-the-art in this area and evaluate how eye-tracking technology has been applied to analyze the cognitive processes involved in reading and reviewing code.

We defined two search strings that combine terms related to *eye tracking*, *code comprehension* and *code smell* to search in three databases: ScienceDirect, IEEE and ACM. The reason for having two string is that ScienceDirect have limited use of booleans. The primary search string used for ScienceDirect was:

- *("code smell" OR "design flaw") AND ("eye tracker" OR "eye tracking") AND ("cognitive effort" OR "code comprehension")*

For ACM Digital Library and IEEE Xplore, we applied a more detailed version of the search string:

– *("code smell" OR "code smells" OR "bad smell" OR "design flaw") AND ("eye tracking" OR "eye tracker" OR "gaze tracking" OR "visual attention") AND ("cognitive effort" OR "mental load" OR "developer comprehension" OR "code comprehension")*

The searches were conducted within the full text of the publications and filtered to include academic research articles and journals. The results showed the identification of 7 papers in ScienceDirect and 8 papers in the ACM Digital Library, but no relevant work was found in IEEE Xplore.

After analyzing the abstracts and contents of the retrieved papers, we selected 11 studies related to our topic. We highlight their findings, relevant contributions, and main differences in comparison with this research.

We grouped the studies by common themes using a *clustering* strategy to identify shared objectives. This approach allowed us to classify the works into specific research areas, facilitating the analysis of trends in studies on *code comprehension*, *cognitive effort*, and *code smells*.

The results indicate that few studies directly address the impact of *code smells* on code comprehension using physiological metrics. Most of the retrieved articles focus on general analyses of cognitive processes in software maintenance and code review tasks, often without explicitly mentioning *code smells*. This highlights the relevance of our investigation, which aims to provide a deeper integration between *eye tracking*, *cognitive load*, and *structural elements* of source code.

Next, we present a detailed synthesis of the related works, grouped by areas of interest. We highlight their methodologies, key findings, and relevant contributions to our research.

Feitelson [34] highlights that traditional complexity metrics often fail to predict code comprehension accurately, emphasizing the need to account for human factors and context. While his study explores how code complexity influences comprehension through experimental evaluations, it does not specifically focus on the role of code smells.

Politowski et al. [35] examine the effects of anti-patterns (*Blob* and *Spaghetti Code*) on program comprehension. The researchers analyzed data from 372 tasks completed by participants across multiple universities, measuring time spent, accuracy, and effort during comprehension activities. Their findings reveal that while single instances of these anti-patterns have minimal impact, multiple occurrences increase task completion time and effort, with *Spaghetti Code* leading to a 39% increase in time spent and a 25% reduction in correctness.

Pinto et al. [36] introduce Cognitive Driven Development (CDD), a technique aimed at reducing cognitive load by limiting complexity in code units. Through a one-year case study at Zup Innovation, the researchers found that CDD maintained small class sizes, improved testing practices, and guided evidence-driven refactoring.

The study provides empirical evidence on how limiting the cognitive load in code units can improve code comprehension and refactoring practices.

Müller et al. [37] investigate the role of biometric measures, including eye tracking, in identifying developers' emotions and progress during software maintenance tasks. Their study demonstrates that emotions such as frustration and happiness are closely linked to perceived progress and that biometric indicators, including eye-tracking metrics like fixation, can effectively predict emotional states. They achieved 71.36% of accuracy in distinguishing between positive and negative emotions.

Da Costa et al. [11] conducted an eye-tracking study to investigate the impact of code transformations on the comprehension of novice programmers across three programming languages (C, Python, and Java). The study found that certain transformations, such as *Extract Method*, significantly improved task performance by reducing visual effort and increasing comprehension accuracy. Their results underscore the limitations of static code metrics in capturing dynamic cognitive processes, emphasizing the value of visual metrics like fixation duration and regressions. Our work focuses on developers' responses to pre-existing code smells rather than applied refactorings.

Li et al. [38] explore the use of eye tracking to evaluate the impact of augmented visualization cues on pilots' monitoring performance in aviation. The study assessed how augmented primary flight displays affected fixation patterns and pupil dilation, finding that these enhancements led to shorter fixation durations and smaller pupil dilatations, indicating reduced cognitive load. The context differs from software engineering but the study demonstrates the effectiveness of eye-tracking in evaluating visual attention and cognitive workload.

Andaloussi et al. [39] proposed a novel approach to estimate the cognitive load of developers at a fine-grained level using eye tracking and machine learning models. Their study demonstrated that by analyzing metrics such as fixations, saccades, and pupil reactions, it is possible to identify mentally demanding code fragments with high accuracy (F1: 85.65%).

Although not directly related to software engineering, Albaghli et al. [40] investigate web usability in Kuwaiti universities using eye-tracking with Standardized User Experience Percentile Rank Questionnaire (SUPR-Q) metrics. The study combined fixation durations and qualitative feedback to identify design issues that increased cognitive load during navigation tasks. Websites with poor layouts exhibited longer fixation durations, reflecting higher mental effort. While focused on web design, this study demonstrate the use of eye tracker for analyzing cognitive load and visual attention and correlating fixation metrics with user performance.

Merino et al. [41] present a systematic literature review on software visualization evaluation, analyzing 181 studies. The review notes that only 29% of these studies employed experimental methods, with eye-tracking mentioned as a data collection technique in a small subset. It calls for more robust evaluation practices integrating quantitative and qualitative data. Although centered on visualization

tools, the study highlights the importance of a mixed-methods approach, combining different data collection methods.

2.4 Summary

This chapter has laid the groundwork for understanding the core concepts and related research that support this dissertation. We introduced the notion of code smells, focusing on three specific types – *Long Method*, *Feature Envy*, and *Data Class* – and discussed their implications for software quality and maintenance. Additionally, we explored the role of eye-tracking technology in capturing developers' cognitive processes during code comprehension tasks, highlighting its potential to reveal insights into how code smells affect visual attention and effort. The review of related studies contextualized our work within the broader landscape of software engineering research, identifying gaps that our empirical investigations aim to address. Together, these elements provide a solid foundation for the subsequent chapters, where we present controlled experiments designed to analyze developers' responses to code smells using eye-tracking metrics and qualitative feedback.

3
Eyes on Code Smells: Analyzing Developers' Responses During Code Snippet Analysis

A key challenge in software maintenance is understanding the impact of code smells on developers' cognitive effort during code comprehension. Code smells are indicators of potential design or implementation flaws that can significantly hinder comprehension. Analyzing how developers interact with code that may present code smells, seeking to understand their difficulties and mental effort, is essential for improving software quality.

Limited attention has been given to how developers visually engage with code snippets containing these smells and how their cognitive effort varies. To address this gap, this chapter presents the paper *"Eyes on Code Smells: Analyzing Developers' Responses During Code Snippet Analysis"*, published and presented at the *38th Brazilian Symposium on Software Engineering (SBES)*.

This dissertation reports on a controlled experiment designed to assess the cognitive effort imposed by three types of code smells: *Long Method*, *Feature Envy*, and *Data Class*, due to a combination of theoretical rationale and practical considerations (see Chapter 2.1). We use eye-tracking metrics, fixation count and fixation duration, to quantify developers' cognitive load while analyzing code snippets with and without code smells. Another metric is Area of Interest, where we use the syntactic categorization functionality based on the syntactic categories of the code (see Section 3.2.7.1).

The results indicate that *Long Method* and *Feature Envy* impose a higher cognitive burden compared to *Data Class*, reinforcing findings that structural complexity affects comprehension. Specifically, developers spent more time fixating on code snippets containing Long Method and Feature Envy, whereas Data Class smells resulted in shorter fixation durations, indicating lower cognitive demands. We also show that developers invested much time in understanding the names within the code, which demonstrates the importance of good and clear nomenclature even for tinier elements, for the sake of supporting code readability and maintainability.

3.1
Introduction

Code smells are claimed to be a key influencing factor in harming program comprehension, as they are symptoms of poor design and implementation decisions [2]. Thus, addressing code smells is important for companies to avoid rework,

increased costs, and decreased productivity in software development projects [4]. This highlights the need to thoroughly understand the impact of code smells on program comprehension.

Eye tracking is a sophisticated technology used to monitor eye movements and gaze patterns, providing a valuable tool for examining the effects of code smells on program comprehension [33]. By using eye-tracking technology, researchers can explore developers' physiological responses during program analysis, identifying which parts of the code they focused on, which elements cause distraction, and how long certain stimuli or triggers capture attention. Analyzing eye movements is essential to understand the cognitive process, as they guide and orient visual attention to regions of interest, which are subsequently processed by the brain [5].

Eye trackers have been widely used in research to develop new insights into how developers interact with software systems over a wide variety of tasks [5]. Sharafi et al. [5] highlight that this non-invasive technology leverages metrics such as pupil diameter, saccade patterns, scanning paths, and fixation points, allowing researchers to achieve pioneering discoveries and advancements in software engineering research. However, currently there is no empirical study on using eye trackers to understand the impact of code smells on program comprehension activities.

In this context, this paper reports on a study in which we have used an eye tracker to investigate the influence of code smells on developers' program comprehension. The eye tracker was used to monitor participants' visual attention, enabling a quantitative evaluation of their visual efforts while they engaged with analytical tasks on code snippets. During the experiment, participants performed tasks on code snippets with and without code smells, allowing us to examine their interactions with the code. Specifically, we utilized eye-tracking information such as fixation duration and areas of interest (AOI) to understand how the presence of code smells influenced developers' program analysis. Fixation duration assesses the cognitive effort expended by developers during the analysis of code snippets. This metric indicates in which part of the code and for how long developers are focusing their attention. For the metric AOI, we use the syntactic categorization functionality of iTrace based on the syntactic categories of the code.

To capture and analyze data, we used the iTrace Eclipse Plugin [6], iTrace-Toolkit [42] and OBS Studio[1]. Additionally, questionnaires were administered to gather complementary information from participants. By analyzing eye-tracking data while developers reviewed code snippets with and without code smells, this study identifies the key aspects that developers focus on during their analysis and compares their responses. Through this detailed analysis, our aim was to gain a deeper understanding of how specific code attributes influence developers' perceptions and responses to structural problems within the code. These insights are crucial to improving the development of more effective tools and practices for

[1] `https://obsproject.com/pt-br`

code refactoring and software maintenance.

Our key findings and related implications are as follows:

1. We observed that the smell *Data Class* leads to a lower cognitive effort, while the smells *Feature Envy* and *Long Method* imposed a considerably higher effort. That explains, for instance, why recent studies have reported that the refactoring of *Feature Envies* and *Long Methods* has been much more common across projects [1, 43, 44, 45].

2. *Long methods* were smells that clearly yielded the greatest effort. Knowing which code smell demands the most from the developer can help one formulate best practices. Moreover, IDE features should better equip developers with clues to support (re)writing of *Long Methods* and help developers prioritize refactoring efforts. For example, IDE support could automatically suggest which parts of a long method could be further decomposed into two, three, or more methods, taking eye-tracking measures to support the decision.

3. We observed that most participants struggled to correctly identify the presence of *Feature Envies* in the programs they analyzed. Some participants did it right. However, the ones that did wrong, engaged in complex cognitive processes as they were looking all around to understand all the dependencies. For them, it was difficult to determine if the "envy" should or not be moved to another class. The difficulty comes from the fact that: *(i)* there were many dependencies with different weights and responsibilities all around, and *(ii)* each dependency carried a different semantics, which could only be inferred after analyzing varying context-specific variables. Existing IDEs and refactoring tooling should indicate which dependencies were better conditioned to be removed.

4. Our findings also highlight the importance of good and clear nomenclature for code readability and maintainability. Certain categories such as `function_decl` and `if` indicate a deeper analysis of functions and control flow structures, likely due to their complexity and potential impact on program execution.

Audience and contribution. The audience for this paper is researchers and professionals in the field of software engineering, particularly those involved in code refactoring, software quality assessment, and software maintenance and evolution, who seek a deeper understanding of the cognitive processes and behavioral responses of developers during code comprehension. Eye-tracking measures can be leveraged in real time as developers write, review, or understand the source code. By integrating eye-tracking technology into development environments, tools can identify the cognitive patterns developers exhibit in response to specific code smells, leading

to more nuanced and precise detection algorithms that surpass traditional static, dynamic, and repository-based measures (*e.g.*, change history metrics) commonly reported in the literature. This integration enables tools to provide immediate feedback by alerting developers to high cognitive loads, highlighting problematic areas of code, and suggesting prioritization strategies. Moreover, it can prompt developers to take breaks or seek help, thereby improving overall productivity and well-being. This paper aims to advance the field of software engineering by inspiring further research and the development of more effective tools for identifying and refactoring code smells.

3.2 Study Design

In this study, we investigate how developers react to code smells by analyzing their cognitive responses during code review. To capture these responses, we used an eye tracker, aiming to identify the behavioral patterns developers present during code analysis. This approach allows us to detect code sections that are difficult to understand and, consequently, more likely to contain code smells.

To ensure the success of the experiment, we divided our research methodology into 7 stages (see Figure 3.1): (1) preparation of the experiment, (2) selection of a state-of-the-art dataset, (3) selection of the code snippets, (4) pilot study, (5) call for volunteers, (6) execution of the experiment, and (7) data analysis.

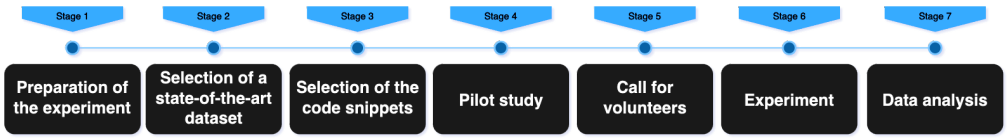

Figure 3.1: Overview of our research methodology.

3.2.1 Preparation of the Experiment

It should be noted that the experiment in question was submitted and approved by the research ethics committee (CEP) through the *Plataforma Brasil* under the Certificate of Presentation of Ethical Appreciation (CAAE) number 74286223.4.0000.5235. Any data or elements that could identify the participant, such as their name or image, have not been and will not be disclosed or exposed, being kept anonymous. If the participant agreed to participate, they agreed with the "Free and Informed Consent Form" (TCLE), a document containing all the information regarding data collection and how their data will be treated.

The study took place at PUC-Rio; it lasted approximately 1 hour, depending on the level of experience and detail in which the developer analyzed the code snippets and responded to the survey carried out together. It was carried out in an isolated room, away from noise and interruptions. Information was provided

before the experiment for developers to familiarize themselves with the materials (eye tracker, keyboard, mouse, and IDE where the developers analyze the code snippets), instructions on how they have to experiment, and any questions they may have. We emphasize to the participant that everyone has their own way of approaching when solving coding problems. The interest lies in observing their distinct perspectives, and they should not concern themselves with the correctness of their answers or the time taken to analyze each code file. They are encouraged to take the necessary time. The primary objective is to observe the answers they gave for the tasks presented, where there are no incorrect answers.

The eye tracker was calibrated with the participant (see Section 3.2.6). Participants also had a brief introductory discussion with the study authors to familiarize themselves with the eye-tracking equipment used. This dialogue was supposed to calm down the subjects so that their emotional state would be neutral, which would only make them concentrate on the experiment task of code smell identification and classification. This preparatory step is crucial to minimize any external influences on the data collected through the eye tracker.

3.2.2 Selection of a State-of-the-Art Dataset

There are several code smells datasets available, such as the one published by Palomba et al. [16], which contains 243 instances of five types of code smells and another containing 17,350 instances of 13 types of code smells. There are also two datasets developed by Fontana et al. that work with binary classification [28] and severity scale [46]. The reason for not using these datasets is that they were either labeled automatically by software tools or by students and researchers, missing developers working in the industry.

Thus, we choose to use the dataset called "MLCQ" (Madeyski Lewowski Code Quest) developed by Lech Madeyski and Tomasz Lewowski [10], which was manually labeled with the support of 26 developers from industry who participated in reviewing the code snippets. All developers were actively involved in activities related to code smells. In total, 4,770 code samples from 792 open-source and industry-relevant projects were reviewed, totaling 14,739 reviews. Moreover, the authors provided detailed information about the background of the reviewers involved in the labeling process. This provided us with a detailed level of information when evaluating the data. The reviewers focused on 4 code smells: *Feature Envy* and *Long Method* at the method level; and *Data Class* and *Blob* at the class level. These were chosen due to a literature review carried out by the authors as the most popular code smells present in the literature. They also classified code smells according to 4 severity degrees based on knowledge and experience. The 4 severity degrees are: *none, minor, major*, and *critical*. We must emphasize that *none* is the classification given when the developer did not see the presence of the smell attributed to the code snippet. A *minor* code smell means a relatively low-impact issue in the code.

A *major* code smell suggests a more significant issue in the code that could impact maintainability and readability. A *critical* code smell indicates a severe issue that can significantly impact the software's maintainability and readability. A questionnaire with 59 questions was applied to the 26 reviewers, of which 20 responded. The authors carefully selected the number of questions to allow a detailed understanding of the background of the reviewers participating in the study, to obtain a more comprehensive and in-depth view of their skills and experience. For more details on how the dataset was developed, see the original study [10] and access the dataset at `https://zenodo.org/records/3666840`.

3.2.3 Selection of the Code Snippets

To select the code snippets from MLCQ, it was decided to filter by the background of the reviewers who manually labeled the code snippets. To do this, we applied the following inclusion criteria (IC):

(IC1) Experience in software development greater than 3 years.

(IC2) Experience in the software industry greater than 3 years.

(IC3) Experience with the Java language greater than 1 month.

Moreover, we applied the following exclusion criteria (EC):

(EC1) Developers who did not answer any questions related to the code smells that were presented.

(EC2) Code snippets longer than 44 lines.

Although the iTrace Eclipse Plugin allows developers to use the scroll bar, we chose to consider EC2 for two main reasons: (1) Keeping the snippets more readable on standard screen resolutions, without the need for constant scrolling within the IDE (but allowing participants to explore the complete code file). (2) Longer code snippets would require more time for developers to analyze from a total of 13 code snippets. These factors were confirmed in a pilot experiment, where developers spent over two hours finishing the experiment, guiding our decision to limit the code snippet length, and ensuring a balance between data quality and participant workload.

We selected a total of 13 code snippets. 1 code snippet was selected for the participant to become familiar with the presentation format, while the remaining 12 were used for data collection. For these 12 snippets, we randomly selected 4 snippets related to each code smell, except for the Blob code smell. We decided to eliminate Blob because including 16 code snippets would take too much of the participants' time, as highlighted in the pilot study (Section 4.2.5).

Before starting the experiment, two researchers evaluated these 13 code snippets to ensure their clarity and comprehensibility.

3.2.4 Pilot Study

We piloted the survey with two practitioners to estimate its length and clarity. The pilot study was carried out based on the guidelines provided by Sharafi et al. [32]: *(i)* we ensured that the eye tracker and room were set up correctly; *(ii)* we verified that the recording process properly acquired and saved data to disk; *(iii)* we checked the quality of the recorded data to ensure that the lighting conditions were appropriate for capturing eye movements; *(vi)* we observed how the participant reacted to the research questions, setup, and tasks; *(v)* we recorded the time taken by the participant to complete the study; and *(vi)* we analyzed the data to evaluate the results and prevent any data loss.

The first participant took 90 minutes to analyze 16 code snippets and noted fatigue, discomfort with the chair, and errors found in the forms to be filled out. This feedback led to minor adjustments. Due to the extensive duration of the pilot, we decided to focus on 3 code smells and select 12 code snippets. As a result, the duration of the second pilot was reduced to 60 minutes, including the time allocated for completing a questionnaire on the researcher's background, which was conducted after the code snippet analysis.

3.2.5 Call for Volunteers

We selected participants under a few constraints: *(i)* participants must have contact with Java or other similar syntax programming language, and *(ii)* participants must have heard about code smells. With these minimum requirements, we seek to ensure that the participants have a minimum understanding of the code snippet and software quality assurance. We seek participants from universities and companies. We use the snowballing strategy [47], asking each participant to refer our survey to colleagues with similar experiences and interest in joining.

3.2.6 Experiment

The survey was carried out via Google Forms, containing both multiple-choice and free-text questions. It begins by outlining the survey's purpose and research goals, emphasizing the confidentiality of participants' responses. The survey is divided into three phases: before, during, and after the experiment.

The first phase presents the definition and types of code smells. The second phase involves analyzing the code snippets using the eye tracker. Finally, in the third phase, participants are asked about their feelings during and after the analysis and their perceptions regarding the use of biosensors. Additionally, to better understand their background, participants' knowledge of the software engineering area is col-

lected. All responses were supplemented by audio and video recordings. The survey is available in our supplementary material [48].

3.2.6.1 First Phase: Introduction to Code Smells

In the first phase, the developer is introduced to the concept of code smell and 7 types of code smells (*Long Method*, *Data Class*, *Duplicate Code*, *Data Clumps*, *Feature Envy*, *Refused Bequest*, and *Message Chains*). We include a wider variety of code smells to ensure no bias in the responses. These measures were adopted to establish a knowledge base, as many are aware of code smells but do not know how to identify or classify them correctly. Thus, the participant can be more confident evaluating the code snippets.

3.2.6.2 Second Phase: Analysis of Code Snippets

In this phase, the developers performed the analysis of the code snippets. The first code snippet was an example so that the participant could clarify any doubts with the researcher. In subsequent sections, the participant carried out the analysis without any interference, aiming for the integrity of the data. We presented the code snippets in the same order for all participants. To mitigate the risk of order effects, such as learning or mental fatigue, we diversified the types of code smells throughout the experiment. We show in our supplementary material [48] that there was no significant variation in the time taken or fixation metrics for code snippets placed at the end of the questionnaire, indicating that neither learning effects nor mental fatigue significantly impacted the results.

During the analysis of each code snippet, participants were asked to *(i)* describe how the code snippet works, *(ii)* whether it was difficult or not to understand and explain why, *(iii)* if it has a code smell and, if so, what is its severity and why choose this severity, and, lastly, *(iv)* how the participant felt when analyzing the code snippet on a scale of 1 to 5. Knowing that each participant has their own style and approach to solving coding problems, the authors were interested in seeing each participant's perspective. Attention check questions were also included to identify whether the participant remained attentive during the experiment.

3.2.6.3 Third Phase: Collecting Participant Background Data

In the third phase, the participant responded to a subset of questions derived from the original MLCQ article [10]. This inclusion is intended to allow future research to establish a link between both studies. The questions focused on the participant's history as a developer and to understand their development experience. In particular, we explored aspects such as the duration of their programming career,

their experience with software development, the programming languages they are familiar with, and their knowledge of the concept of code smells, among others. The objective of this phase is to segment the data more precisely and to contextualize the analysis based on the participants' backgrounds. The background information was collected at the end of the experiment, rather than at the beginning, to avoid any impact of participant fatigue on the eye-tracking data. Collecting background data at the end ensured that participants could perform the main experimental tasks without prior cognitive fatigue, thus preserving the reliability of physiological measures. Additionally, we solicited the participants' opinions on the utility of the data gathered by the eye tracker in understanding their assessments.

3.2.7 Data Analysis

In the Data Analysis, we conducted a thorough investigation into the developers' responses while analyzing code snippets, using advanced tools for precise data collection and analysis. The primary tool used was the iTrace-core software [6]. This software, when employed in the experiments, generated detailed XML files, allowing us to understand where each participant was looking at specific times, the duration of their gaze, and the specifics of the code being analyzed.

In addition to iTrace-core, we also used the iTrace ToolKit[42], a complementary tool to process raw data. This toolkit was crucial in creating a database and calculating fixations. Among all collected data, the fixation and syntactic categories were of particular importance. We chose to work with fixation because it is a proven metric related to the cognitive process [32], which indicates where developers are focusing their attention in the code and is derived from time. We chose the I-VT algorithm [42], to calculate fixation duration, recommended for eye trackers with a refresh rate higher than 200Hz aligning with our equipment (see Section 3.2.8).

Data preparation initially consisted of analyzing the fixation durations and eliminating outliers. To do this, we used boxplots to visualize the severities and determined the upper and lower limits as 1.5 times the Interquartile Range (IQR). For our analysis, we utilized several eye-tracking metrics, including Average Fixation Duration (AFD), also known as Mean Fixation Duration (MFD), which calculates the average duration of all fixations within the area of interest (AOI); and fixation count (FC), which measures the total number of fixations in each AOI [7].

3.2.7.1 Research Questions

To investigate how developers react and comprehend code snippets, we focused on analyzing the following research questions (RQs):

RQ1 What is the average time that developers spend when analyzing code snippets with potential code smells?

RQ2 Which sections of the code are most frequently examined by developers when analyzing code snippets?

RQ3 How do fixation patterns differ between developers who accurately identify code smells, and those who do not?

The data collected through the eye tracker were analyzed to identify patterns in the developer's behavior and answer the research questions above. This analysis took into account physiological responses, time spent on each code snippet, and the developer's reading pattern according to its accuracy in detecting the code smell and code smell type.

Analysis of RQ1. Developers often spend time reviewing and analyzing code snippets, either their own or those of others, to identify refactoring opportunities. The time spent on this process can vary widely based on several factors, including the complexity of the code, the developer's familiarity with the codebase, the presence and nature of the code smells, and the developer's experience level. In this context, this research question seeks to analyze how developers' fixations behave during the analysis of code snippets, calculating the Average Fixation Duration (AFD) multiplied by the Fixations Count (FC). A high amount of fixations indicates that more effort is required to maintain and evolve the system [49, 50, 51]. Knowing which code smell demands the most from the developer can help formulate best practices for writing code and help prioritize refactoring efforts.

Analysis of RQ2. This research question aims to discover the regions of the code that receive the most attention from developers. Understanding these Areas of Interest (AOI) can shed light on critical aspects of the code that require closer analysis for effective code smell detection. For this analysis, we used a syntactic hierarchical model extracted from the database generated by the iTrace ToolKit. The syntactic context column stores an arrowed list of tags that describes where the text is located contextually [42]. Using AFD times FC, we calculated the time in minutes for the syntactic categories that developers spent the most time looking at. After identifying the 10 categories, we break the chain to identify which abstract synthetic information appears most within the 10 identified categories.

Analysis of RQ3. This research question aims to explore the relationship between developers' fixation patterns and their ability to identify code smells. The goal is to determine whether there is a significant difference in fixation patterns between developers who successfully identified smells and those who did not. First, we verified the smelly and non-smelly instances reviewed by professionals. Then, we compared the fixation count between these reviews. To carry out this analysis, we performed a Min-Max normalization. We also analyzed the $AFD * FC$ boxplots, so that we can understand if the pattern of fixations varies for each code smell.

3.2.8 Data Collection and Availability

To collect data from participants, the Tobii TX300 eye tracker was used. According to its manufacturer, combination of 300Hz sampling rate, very high precision and accuracy, robust tracking and compensation for large head movements extends the possibilities for unobtrusive research of oculomotor functions and human behavior [52]. In addition, we use Eclipse IDE[2] to present the code excerpt in conjunction with Itrace Eclipse plugin and Open Broadcaster Software (OBS Studio)[3] to record our experiment. Google Forms was also used to collect participants' responses.

All files we used for the elaboration of the study and to display the graphics and data tables present in this paper can be accessed at our GitHub repository [48].

3.3 Results

In this section, we will discuss the participants' characterization and the research questions based on the results obtained from the data collected by the eye tracker.

3.3.1 Participants Contextualization

In total, we carried out 11 valid survey responses and 12 experiments where we collected valid data via eye tracker.

Careful Data Sanitization. To analyze the validity of the data, we first checked the completeness of the collected data; *i.e.*, we checked whether all the questions were appropriately answered, ensuring that there were no missing entries that could impact the integrity of the analysis. Also, after the experiment and generation of the databases, it was checked whether the data collected via eye tracker were consistent. We checked for (ab)normality bases with little data, discrepancies in the size of the generated database file, and few fixations. All data collected via the eye tracker were also validated, except for experiment 10, code snippet 2, as the eye tracker was not working due to an iTrace error.

The difference between the number of experiments and surveys was due to 1 of the surveys not being saved due to internet connection problems. Moreover, among the 11 surveys collected, experiment 1, code snippet 12; and experiment 2, code snippet 13, were not collected due to errors. Thus, when data analysis was directly related to these code snippets or responses missing from the survey, data were disregarded.

[2]https://www.eclipse.org/ide/
[3]https://www.obsproject.com/

Figure 3.2 shows that most participants, either already heard about code smells (54.55%) or know what they are (36.36%). Only a few participants (9.09%) only heard about the concept during the experiment. Figure 3.3 shows that more than 50% of the participants have at least a bachelor's degree in Science or Engineering.

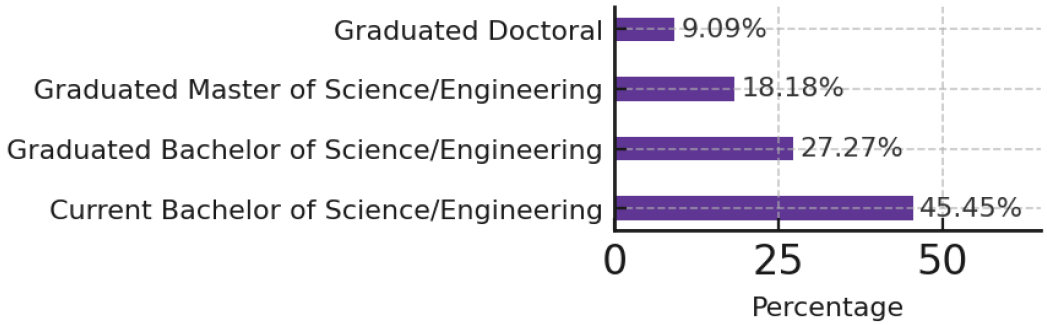

Figure 3.2: Participants' familiarity with the concept of smells.

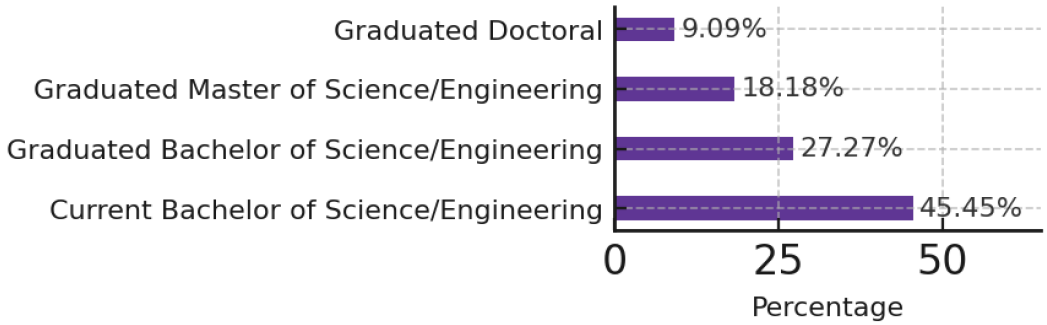

Figure 3.3: Participants' degree.

3.3.2 RQ1: Fixation Time

The process involves a detailed analysis aimed at understanding the relationship between developers' attention (AFD) and their cognitive load (FC) while they analyze code snippets, particularly those with potential code smells. We assume that more time and more fixations correlate with greater difficulty or complexity.

Figure 3.4 shows the frequency of different levels of cognitive load for $AFD * FC$ in milliseconds per smell type. The code smell *Data Class* has the median lower than the boxplots of the other code smells and code snippets without code smells (*None*). It is also observed that the third quartile is lower than the second quartile of the other code smells.

Notice that for *Long Method*, the opposite of the smell *Data Class* occurs, with the median, third quartile and upper limit above the other boxplots. Both

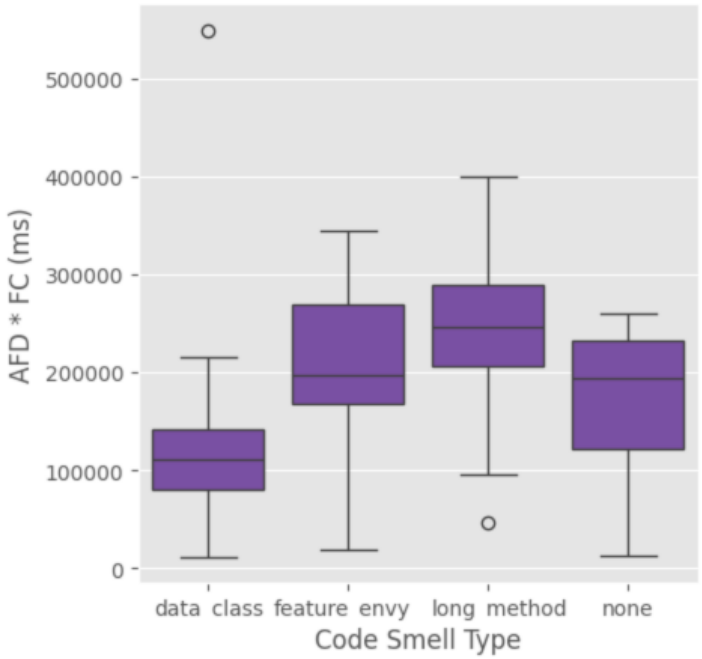

Figure 3.4: Boxplot of AFD * FC per smell type.

interquartile ranges are also smaller when compared to the other boxplots. This indicates that the code smell *Data Class* may not be as impactful as the other code smell types, suggesting that these may be less problematic in relation to the complexity of the code, while the smell *Long Method* presents greater complexity during its analysis. The smaller interquartile range also indicates that the values are more consistent and less variable than the other two boxplots (*Feature Envy* and *None*) presented. This consistency may imply that both code smells are more predictable and potentially more manageable compared to *Feature Envy*.

The stacked bar chart in Figure 3.5 presents the cognitive load ($AFD * FC$) for each code snippet. With a higher count for *Data Class* at the beginning of the plot, we see that this smell type presents a lower cognitive effort when compared to the other code smells. Whereas the *Long Method* smells led to a higher effort.

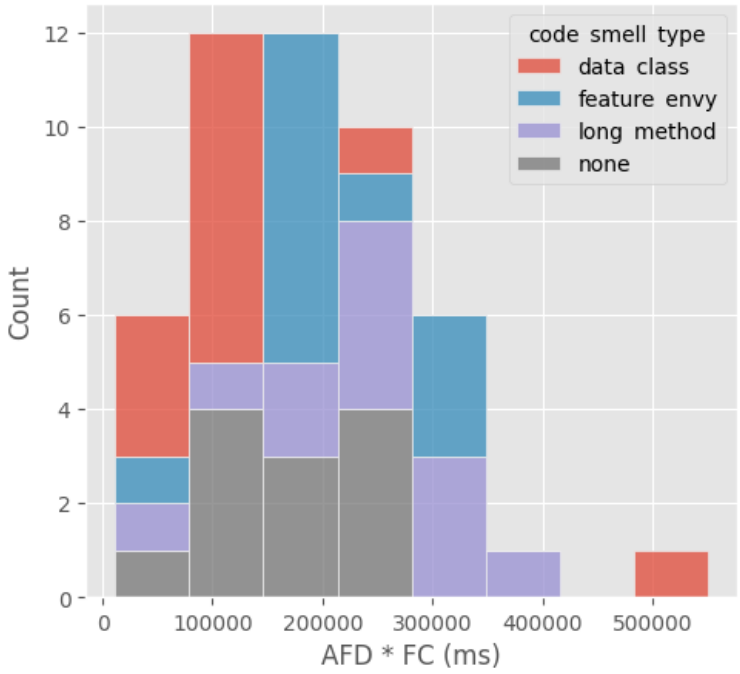

Figure 3.5: Stacked bar of AFD * FC per smell type.

To demonstrate our results, we used normalized qualitative data about the perceived complexity of the code snippet by the participants and quantitative fixation data from the eye tracker. Our results indicate that the code snippets considered more complex by the developers required more fixations, resulting in higher cognitive effort [48]. This is particularly evident in the case of the *Long Method* smells. Additionally, we noted that the *Data Class* smell presents a lower cognitive effort compared to other code smells, as reflected by lower fixation values.

Our data revealed a consensus among the participants' explanations when there was no code smell or when a specific type of code smell, such as the *Data Class* smell, was present. This consensus is further reflected in the lower perceived complexity for *None* and *Data Class*. For *Long Method*, which has higher perceived complexity, participants' explanations were more varied and less consistent.

3.3.3 RQ2: Most Examined Code Sections

To answer this research question, we need to understand the granularity of the syntactic categories involved. `block`, `class`, or `unit` are very broad categories, making them cover significant parts of the code concerning their scope and organizational structure. On the other side, there are much more granular categories such as `name`, `if`, `decl`, and `call`. They are related to the specific elements of the code inside each statement like variables, conditionals operations, declarative operations, and function calls.

One of the types of data that we can collect from the database generated using eye tracker data is `syntactic_category`. This column stores an arrowed list that presents syntactic categories and shows where in the code the fixation was performed contextually. Thus, we identified 10 arrows lists that presented the highest AFD*FC in minutes for all developers (Figure 3.6). From these 10 arrowed lists, we counted all syntactic categories that are present within the arrowed lists to identify which ones have the greatest representation.

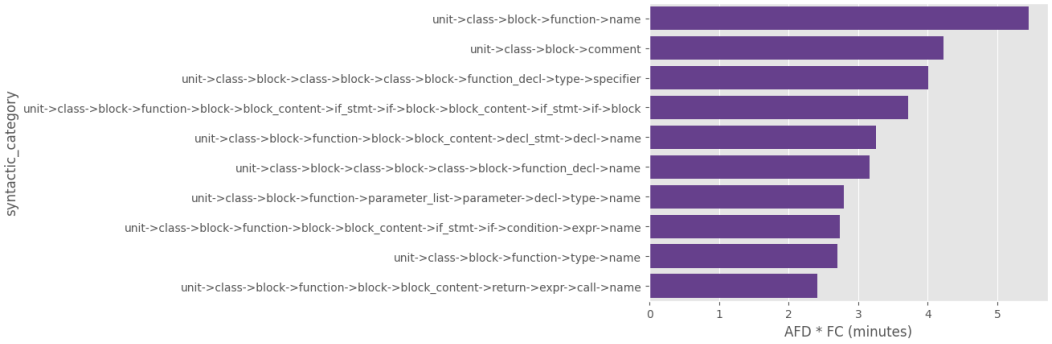

Figure 3.6: TOP 10 arrowed list for the hightest AFD*FC (minutes) for all developers.

Figure 3.7 shows the syntactic categories that appeared the most in the 10 arrowed lists with high AFD*FC. We observe that developers pay fair attention to both, i.e., the general and the specific categories of the code while inspecting code snippets. The categories that are more general, like `block`, `class`, and `unit`, are the first three; they allow one focusing on understanding the architectural organization and program layout. Surprisingly, the granular category `name` comes in with an equivalent count of "function" despite its low scope. This shows that developers invest much time in understanding the names within the code. This highlight demonstrates the importance of good and clear nomenclature even for tinier elements, for the sake of supporting code readability and maintainability. Other categories such as `function_decl` and `if` indicate a deeper analysis into functions and control flow structures, likely due to their complexity and potential impact on program execution.

3.3.4 RQ3: Fixation Patterns

When analyzing Table 3.1, we noticed that there is no huge difference between right and wrong answers regarding the identification or not of code smells.

Figure 3.8 distinguishes the stacked bar of normalized FC count for right (purple) and wrong (blue) answers for all code snippets. It also presents a higher count for the right answers at the lowest values of normalized FC; the wrong answers present a distribution more centralized. This may indicate that developers who correctly identified code smells did so with fewer fixations, possibly indicating

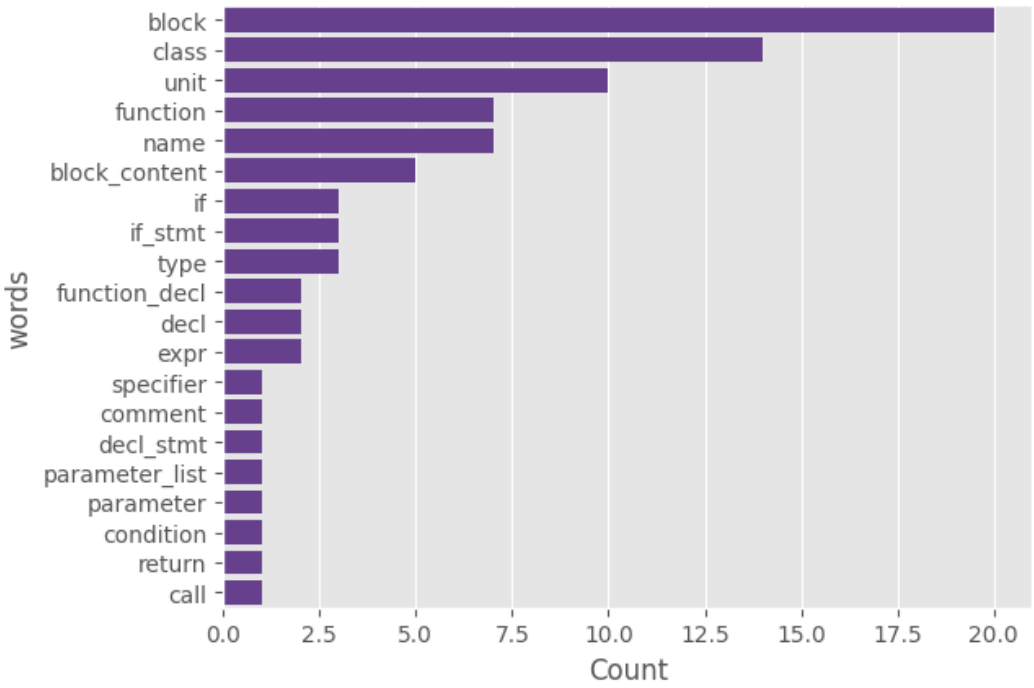

Figure 3.7: Count of the most frequent syntactic categories in the 10 arrowed list of Figure 3.6.

Table 3.1: Total Count and Percentage of Right and Wrong Answers for All Code Snippets.

Answer Type	Count	Percentage (%)
Right	51	43.22
Wrong	67	56.78

a greater level of knowledge and familiarity with the code smell, which can be interpreted as a greater efficiency in the analysis process.

Increasing the granularity of the analysis, we plot boxplots for AFD * FC in milliseconds for each group of code snippets with or without code smells. Figure 3.9 shows the bloxplots for the right answers, and Figure 3.10 shows the bloxplots for the wrong answers for the same scale. First, comparing the medians, we notice that the code smell *Feature Envy* stands out with a median with a higher value for wrong answers, while the other boxplots do not have that much difference. Analyzing the distribution of the data, we noticed a different tendency for *Long Method* and *None* when compared to *Feature Envy*. While *Long Method* and *None* have longer interquartile ranges and larger upper whiskers for correct answers in relation to incorrect answers, *Feature Envy* has the opposite trend, having a very narrow interquartile range and small upper and lower whiskers for correct answers in relation to wrong answers.

Thus, for *Feature Envy*, developers who correctly identified the smell demon-

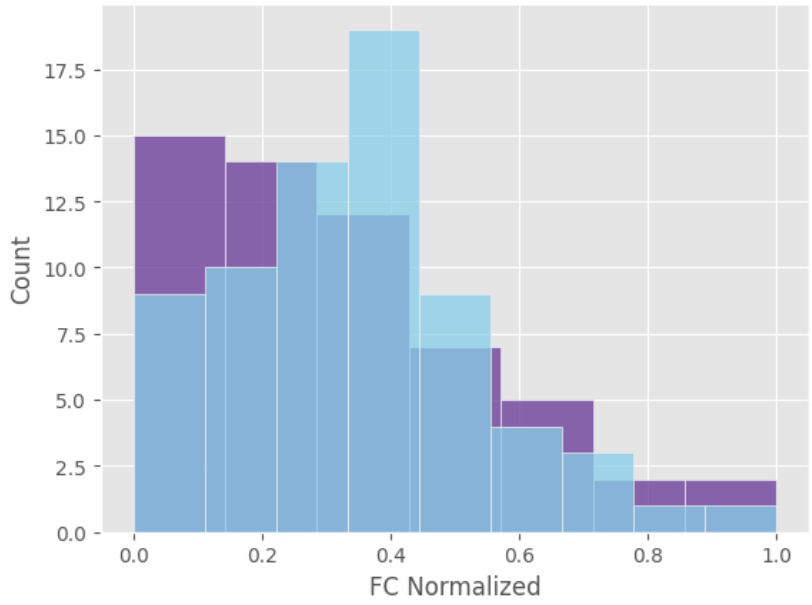

Figure 3.8: Stacked bar of normalized FC count for right (purple) and wrong (blue) answers for all code snippets.

strated greater efficiency and consistency in identifying this specific type of code smell compared to those who did not. For the *Long Method* smell, the results showed that a more detailed analysis accurately identified it, which is expected since this smell is defined by the excessive length of the code.

3.4 Discussion

The implications of the findings in the results section are significant for code review practice and the development of software tools for code smell detection. The recognition that *Long Method* smells require more cognitive resources suggests that new developers may need targeted training to better recognize and manage these smells. Furthermore, attention to general and specific syntactic categories in code review indicates that tools and checklists should be designed to guide developers in inspecting macro and micro elements of code.

Future research could explore the development of custom training modules for identifying code smells, focusing on smells that are more cognitively demanding. The data collected also provides information that can assist in future research focused on individual differences between developers – such as, level of experience in code refactoring, tools they use and length of experience – could generate more insights into how to support developers in the review process of code.

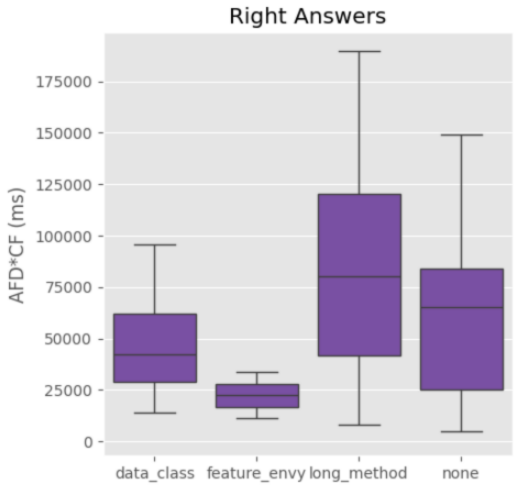

Figure 3.9: Boxplot of AFD * FC by smell type for right answers.

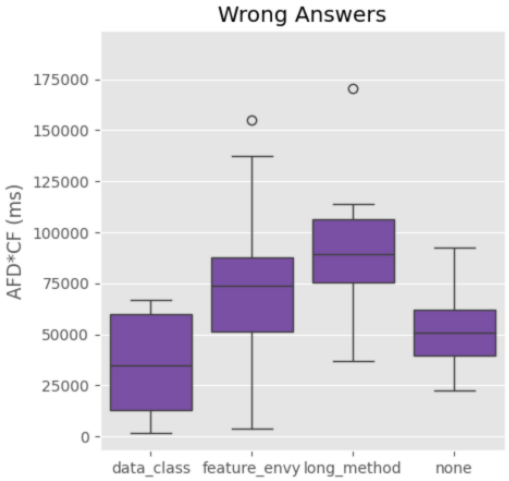

Figure 3.10: Boxplot of AFD * FC by smell type for wrong answers.

3.5 Threats to Validity

Another threat to the validity of the study is the duration of the experiment. The participants needed to remain in their position so that the data captured by the eye tracker was reliable, but they had the freedom to make natural movements to look at the screen. With a duration of up to 1 hour and 30 minutes, some participants may no longer be as comfortable as they were at the beginning, which could compromise the data collected due to loss of eye tracker calibration. It was also observed that some participants were more restless than others, which could also affect the data collected.

As external threats to the validity, we have the number of participants who carry out the study and the selection of short code snippets. The number of participants may be limited due to the fact that the study is carried out in a specific location, requiring participants to travel to the location, which may not be feasible for all potential interested participants. Furthermore, the duration of the

study, which can vary from 30 minutes to 1 hour and 30 minutes, and the number of code smells to be analyzed, asking participants for detailed explanations about their analysis, may deter some participants, potentially impacting the diversity of the sample. The choice of Java as the language for the study, despite its current lower popularity, was a deliberate decision, but it may also limit the generalization of the results to other more current programming languages. The use of smaller code snippets was necessary to maintain a reasonable duration of the experiment, considering that the evaluation of all three code smells, and their four severities already takes a significant amount of time. Expanding to larger code snippets could make the experiment impractical in terms of duration and inaccurate data.

We chose to work only with metrics related to fixations, however other metrics such as pupil diameter and eye blinks can also provide valuable data for analyzing cognitive effort. The choice not to use these metrics was because it might be influenced by external factors, such as ambient lighting and emotional states, making its interpretation more complex and less accurate. In future work, we aim to explore more deep these metrics, as they could provide additional insights into the cognitive effort of developers during code analysis. Moreover, we also plan to conduct a more in-depth examination of the qualitative data collected during the study with a larger number of participants to further develop our conclusions.

3.6 Conclusion

This study explores the physiological responses of developers when analyzing code snippets with and without the presence of code smells. By analyzing eye tracker data, we identified specific categories of developer focus and their responses during code smell analysis. The implications of the findings are significant for code review practice and the development of software tools for code smell detection. The recognition that *Long Method* smells require more cognitive resources suggests that new developers may need targeted training to better recognize and manage these smells. Also, attention to general and specific syntactic categories in code review indicates that tools and checklists should be designed to guide developers in inspecting macro and micro elements of code.

Overall, our findings contribute to the broader field of human-computer interaction by demonstrating the value of eye-tracker data in understanding developers' cognitive processes imposed by code smells. Participants highlighted the utility of specific features such as fixation duration, which was a key metric in our analysis. Additionally, participants mentioned saccade length, pupil dilation, and gaze location. Participants also offered rationales for their perceptions, suggesting that these additional metrics could provide a more comprehensive view of their cognitive processes and difficulties encountered during code analysis.

The results obtained so far are promising and suggest that the continuation of this research, whether by better exploring the data and responses from developers

who participated in the experiment, can offer valuable contributions to the area of Software Engineering. Future research could explore the development of custom training modules for identifying code smells, focusing on smells that are more cognitively demanding. The data collected also provides information that can assist in future research focused on individual differences between developers – such as, level of experience in code refactoring, tools they use and length of experience – to generate more insights into how to support developers in the code review process.

3.7 Summary

In this chapter, we investigated how code smells impact developers' cognitive effort and reading behavior using eye-tracking technology. Cognitive effort were analyzed using eye-tracking metrics (AFD and FC), which indicate the level of attention and cognitive load required to process specific code elements. Reading behaviors were explored by identifying AOIs, revealing which syntactic elements of the code demanded the most attention. Our findings indicate that code smells such as *Long Method* and *Feature Envy* impose a significantly higher cognitive load than *Data Class*. This reinforces the initial hypothesis that more complex structural issues require additional cognitive resources, which in turn may explain why recent studies have found that refactoring *Long method* and *Feature envy* smells is much more common across projects [1, 43, 44, 45]. We also investigated which syntactic categories developers tend to focus on when code smells are present. Our findings indicate that they spend a significant amount of time understanding the names used in the code. This observation highlights the importance of clear and effective naming, especially for functions and control flow structures, given their complexity and potential impact on program execution.

The study presented in this chapter has some limitations. While the sample size is representative, it could be expanded to improve the generalizability of the findings, and statistical results were not presented to further support the outcomes. Additionally, we did not analyze the developers' responses collected during the experiment. Furthermore, only fixation metrics were considered in this study, leaving out other potentially valuable metrics such as pupil diameter and eye blinks. In the next chapter, to address part of these limitations, we expand the dataset by including more responses by developers and introducing new research questions to explore developers' explanations and their cognitive effort. This complementary study also aims to deepen our understanding of the findings presented in this chapter and explore the mental models and reading patterns developers exhibit when analyzing code snippets with or without smells.

4
Reading between the Smells: Eye-Tracking Developer Responses to Code Smells

Understanding developers' perceptions and cognitive processes when analyzing smelly code is essential for improving code comprehension and maintenance practices. While quantitative studies provide valuable insights into cognitive effort through eye-tracking metrics, they often lack an in-depth exploration of developers' thinking reasoning processes. To address this limitation, it is crucial to combine quantitative and qualitative data, enabling a more comprehensive understanding of how developers perceive and interpret code snippets with or without code smells.

In the previous chapter, we conducted an initial investigation into the cognitive effort imposed by three common code smells – *Long Method*, *Feature Envy*, and *Data Class* – by analyzing developers' eye-tracking data. The results demonstrated that different code smells demand varying levels of cognitive resources, with *Long Method* and *Feature Envy* imposing higher cognitive loads compared to *Data Class*. However, the previous study primarily focused on quantitative eye-tracking metrics, such as fixation duration and areas of interest (AOIs), without exploring developers' qualitative reasoning and perceptions about code smells.

To address this limitation, this chapter presents a study submited to the *Transactions on Software Engineering and Methodology (TOSEM)* to explore both quantitative eye-tracking data and qualitative responses from developers, offering a more holistic understanding of the cognitive and perceptual challenges posed by code smells. Thus, we aim to explore how developers perceive and classify code smells based on their self-reported comprehension difficulty, cognitive load, and reading strategies. By combining these approaches, we provide deeper insights into how different types of code smells influence developers' cognitive processes and decision-making.

In summary, this chapter expands upon the previous findings in Chapter 3 by introducing several key improvements: *(i)* a larger dataset, allowing for a broader and more generalizable analysis of developers' interactions with code smells; *(ii)* new research questions, focusing on developers' mental models and reading behaviors, to better capture the nuances of comprehension strategies; and *(iii)* a mixed-methods approach, which triangulates quantitative and qualitative data to gain richer insights into the impact of code smells.

While Section 4.2 contain some overlapping foundational content with Chapter 3, it is recommended to read these sections in full for the best understanding of

this study. They contain improvements, including expanded literature coverage, enhanced methodological details for eye-tracking analysis, and new contextual elements from our qualitative coding framework.

Our results confirm and extend the observations from the previous study, revealing intricate relationships between eye-tracking metrics and developers' qualitative responses. The findings emphasize that the interplay between visual attention and self-reported comprehension difficulty provides valuable information for improving code quality tools and comprehension strategies. These findings contribute directly to addressing the research problem outlined in Section 1.1, particularly in relation to understanding how cognitive effort and reading behavior are influenced by different types of code smells.

4.1 Introduction

Code smells, often described as symptoms of poor design or implementation choices, are known to hinder program comprehension and degrade code quality, ultimately impacting software maintainability, software evolution and productivity [2]. In particular, complex and error-prone code segments with smells such as Long Method, with excessive lines of code and multiple responsibilities, or Feature Envy, with methods that show more interest in other classes than their own, may tend to demand more cognitive effort from developers [53], making it challenging to interpret and modify code effectively.

Although extensive research has been conducted on identifying code smells and understanding their influence on software quality, empirical investigation is still needed into how these smells affect developers' cognitive and behavioral patterns during code comprehension tasks. Despite advances in detection techniques and classification systems [54], there are still gaps in understanding the real-time cognitive impacts these smells impose on developers. We lack comprehensive knowledge about how different categories of smells might affect different aspects of program comprehension, such as mental models or reading patterns. The relationship between specific code smell types and their effects on cognitive load, developer attention, and comprehension efficiency remains largely unexplored.

To explore this gap, this study investigates how the presence of code smells impacts developers' perception of comprehension difficulty through qualitative analysis of free-form developer responses and quantitative analysis of cognitive load and reading behaviors data by leveraging eye-tracking technology. Eye tracking offers unique advantages in software engineering research by providing real-time insights into developers' cognitive processes that would remain hidden with traditional methods alone. Unlike self-reported measures, which can be subject to recall bias, eye tracking reveals unconscious attention patterns and mental effort expended when encountering different code structures [32]. With the eye tracker, we performed a detailed analysis of eye movements, fixation patterns, and visual attention alloca-

tion, providing valuable data on the cognitive processes involved in code review tasks [32]. In addition, we conducted a Grounded Theory analysis of developers' free-form explanations through open and axial coding [4], identifying 14 distinct categories of comprehension challenges. Through the analysis of both quantitative eye tracking data and qualitative responses, we aim to capture how developers respond to code with and without smells, and different types of smells, highlighting the aspects of code that demand greater focus or induce cognitive strain. By analyzing the free-form developer responses about the comprehension challenges and metrics such as fixation count (FC) and average fixation duration (AFD), we provide a nuanced understanding of the mental models and reading patterns that developers form when confronted with smelly code.

The goal of this study is to provide a deeper understanding of how developers perceive, engage in cognitive effort, and mentally process code with varying levels of *'smelliness'* through mental models and reading patterns. To the best of our knowledge, this is the first study to systematically investigate different types of code smells and their unique effects on comprehension, supported by eye-tracking technology.

We provide significant empirical contributions regarding the cognitive and behavioral effects of three specific types of code smells - Long Method, Feature Envy, and Data Class - on developers' code comprehension. Specifically, we demonstrate that Long Method and Feature Envy substantially increase cognitive load and are frequently associated with developers' self-reported comprehension difficulties, particularly those related to code length, logical complexity, and confusing naming conventions. Conversely, Data Class exhibits a comparatively lower cognitive burden. By leveraging eye-tracking technology, we identify predominant bottom-up mental models, as well as top-to-bottom and skimming reading patterns, revealing specific strategies developers adopt when analyzing smelly code. These results offer actionable insights to improve tool support and refactoring practices, particularly benefiting novice developers.

4.2 Study Design

We build this study upon on our previous research [53]. The experiment was carried out with 27 participants, significantly expanding the previous study's sample size and allowing us to address new research questions. To structure our research, we applied the *Goal-Question-Metrics* (GQM) approach [55] as follows:

– *Goal.* Our goal is to analyze Java code snippets containing code smells for the purpose of understanding how they affect developers' behavior; with respect to comprehension difficulty, cognitive effort, and reading patterns; in the context of code review tasks performed by developers.

- *Question.* How do different types of code smell impact developers' visual engagement and code comprehension?
- *Metrics.* We collected both qualitative and quantitative metrics. Qualitative metrics were gathered through surveys, capturing developers' self-reported perceptions regarding code comprehension difficulty, code functionality understanding, and identification of code smells with their reasoning. Quantitative metrics were obtained via eye-tracking, focusing on fixation-based data such as Fixation Count (FC) and Average Fixation Duration (AFD). These quantitative metrics were used to analyze cognitive load, mental models, and reading patterns employed by developers. A description with more detail of these metrics is detailed in Section 4.2.8.

4.2.1 Research Questions

To structure our investigation, we formulate three research questions (RQs) based on the Goal-Question-Metrics framework presented earlier:

> **RQ$_1$**: How does the presence of code smells affect developers' perception of code comprehension difficulty?

Code smells, like overly large classes, can vary in both presence (whether they exist in a given code snippet) and severity (the degree to which they affect code quality). By examining developers' perceptions of difficulty in understanding code with different types of smells, this question aims to reveal whether and to what extent these smells make code harder to understand, which could ultimately affect productivity, code quality, and maintainability.

> **RQ$_2$**: Does the presence of code smells influence the cognitive effort required for developers to analyze code snippets?

To investigate the cognitive effort that code smells impose on developers, we analyze average fixation duration (AFD) and fixation count (FC) as key metrics. AFD reveals how long developers focus on specific elements within the code, while FC indicates the frequency of visual attention shifts. By examining variations in these metrics in response to different types of code smells, we aim to determine whether certain smells intensify the cognitive load on developers. It enables us to identify particular coding patterns that make code review more mentally demanding.

> **RQ$_3$**: How does the presence of code smells influence developers' reading behavior when analyzing code snippets?

This question investigates whether developers' reading patterns – such as their eye movements, focus points, and scanning order – are affected by the presence of code smells. Inspired by the work developed by Abid et al. [56], we identify the mental models and reading patterns of developers who classified code snippets with and without code smells. By analyzing reading patterns, such as the time spent on certain lines, areas of the code that receive the most attention, and the frequency of backtracking, we can understand how code smells impact the natural flow of code review. Uncovering these behavioral patterns can reveal whether certain smells cause developers to revisit code sections, or spend extra time in (or out) the scope of the code snippets to interpret complex areas.

To address these research questions, we designed the experiment to systematically capture both qualitative and quantitative data, and divided our study methodology into seven steps, which are described in the following sections.

4.2.2 Preparation of the Experiment

Our experiment followed all required ethical guidelines and was submitted to the research ethics committee (CEP) through the Brazil Platform under the Certificate of Presentation for Ethical Consideration (CAAE) number 74286223.4.0000.5235. The personal data of the participants, including names and images, were kept strictly confidential, and all participants were required to sign a *"Free and Informed Consent Form"* (TCLE).

The experiment was conducted in an isolated room at PUC-Rio to minimize distractions, with clear instructions and proper eye tracker calibration provided to all participants beforehand. Each session lasted approximately 1 hour, with variations depending on participants' experience and familiarity with the task. To accommodate this, participants were encouraged to proceed at their own pace, ensuring comfort and focus. This approach minimized the impact of session duration variability on data quality. Clear instructions were provided to all participants, and the eye tracker was carefully calibrated before the analysis to ensure consistent data capture.

Moreover, information was provided before the experiment for developers to familiarize themselves with the materials (eye tracker, keyboard, mouse and IDE where the developers analyzed the code snippets). We emphasize to the participant that everyone has their own way of approaching and solving coding problems. The interest lies in observing their distinct perspectives, and they should not concern themselves with the correctness of their answers or the time taken to analyze each code file, which further mitigated performance pressure.

Participants also had a brief introductory discussion with the researchers about the study. This discussion was designed to reduce participant anxiety and ensure a neutral emotional state, thereby minimizing potential biases. This preparatory step

was crucial to minimize any external influences on the data collected through the eye tracker.

4.2.3 Selection of a State-of-the-Art Dataset

We carefully chose the MLCQ (Madeyski Lewowski Code Quest) dataset for the experiment, which was manually labeled by 26 professional developers from the software industry [10]. This dataset provides 4,770 code samples from 792 open-source and industry-relevant projects, totaling 14,739 reviews. The programming language of the dataset is Java, which was chosen because of its extensive popularity [57] and widespread use for the development of code smells datasets [58]. All these dataset characteristics increase the practical applicability of the findings. The dataset also offers detailed background information about the reviewers, allowing a deeper analysis of their skills and experience.

The dataset contains four labeled code smells: *Feature Envy* and *Long Method* at the method level; and *Blob* and *Data Class* at the class level. They labeled code smells according to four severity degrees based on their knowledge and experience [10]:

- *Critical*: indicates a code smell that can have a severe impact on maintainability and readability.
- *Major*: indicates a code smell that can have a high impact on maintainability and readability.
- *Minor*: indicates a code smell that can slightly impact maintainability and readability.
- *None*: indicates no presence of the smell in the code snippet.

We must emphasize that the developers were free to label the severity of the code smells on this four-level scale without any specific training or imposed thresholds, leaving the interpretation up to the individual developers based on their professional experience and judgment.

We selected the MLCQ dataset because of its unique manual labeling process, conducted by experienced industry developers, which stands in contrast to other datasets that rely on automated labeling techniques [16, 28, 46, 59]. The smells in the MLCQ dataset represent distinct types of design issues that could engage participants in different ways, providing valuable insights into their comprehension processes. Additionally, MLCQ provides a distinct advantage over other manually labeled datasets by including severity labels for code smells as well as detailed background information about the reviewers. This context allows for a more nuanced analysis of code quality issues, informed by the professional expertise and severity assessments provided by the dataset. For a comprehensive description of the dataset's development process, the reader should refer to [10]. The dataset is publicly accessible at `https://zenodo.org/records/3666840`.

4.2.4 Selection of the Code Snippets

We selected a total of 13 code snippets for this study. To familiarize participants with the presentation format, one of these code snippets was used as a warm-up exercise, while the remaining 12 snippets were used for data collection. These 12 snippets were carefully chosen to represent three specific types of code smells (Long Method, Feature Envy and Data class). These three types of smells (out of the four) were chosen due to the following combination of theoretical rationale and practical considerations:

1. (1) Long Method: Addresses issues of code length and complexity, requiring developers to navigate and understand larger code blocks;
2. (2) Feature Envy: Involves inter-class dependencies, highlighting challenges related to the lack of cohesion and complex class interactions;
3. (3) Data Class: Relates to data encapsulation, focusing on artificial simplicity and the absence of data-related behavior in the same class.

The diversity of the selected smells allowed us to study a broad spectrum of cognitive and behavioral responses, enriching the insights derived from the analysis. Initially, we considered including a fourth code smell category, Blob, but removed it to avoid overwhelming participants with a total of 16 snippets, as our pilot study (Section 4.2.5) highlighted potential fatigue and undesirably exceeded the limits of time constraints.

For each smell types analyzed, we included one snippet from each severity level: None, Minor, Major, and Critical; with the None category representing "clean" code. Thus, our study included 3 clean code (None) and 9 snippets contained code smells. This study design allowed us to analyze differences in cognitive and behavioral responses.

To ensure the quality and relevance of the selected code snippets, we applied specific inclusion and exclusion criteria based on the MLCQ dataset's reviewer backgrounds. For the inclusion criteria (IC), we selected code snippets labeled by reviewers who met the following qualifications: (IC1) Professional experience in software development of more than 3 years; (IC2) Professional experience in the software industry of more than 3 years; (IC3) Familiarity with the Java language. These criteria ensured that the selected snippets were diverse and evaluated by professionals with substantial expertise in software development and Java programming, increasing the reliability of the manual labels associated with each snippet and the relevance of the data collected for our study.

Additionally, we applied the following exclusion criteria (EC) to refine the selection of code snippets: (EC1) Exclusion of developers who did not respond to any questions related to their background; (EC2) Exclusion of code snippets longer

than 44 lines. Although the iTrace Eclipse Plugin allows developers to scroll through the full file within the IDE, we opted to limit the length of the snippets to balance readability and workload: (1) to improve readability on standard screen resolutions, minimizing the need for frequent scrolling while still allowing participants to view the complete code file as needed; and (2) to reduce participant fatigue, as longer snippets extended analysis time and compromised data quality. These considerations were confirmed during a pilot study, where participants spent over two hours to complete the task set when reviewing longer snippets. This finding guided our decision to limit the code snippet length to ensure a balance between data quality and participant workload.

Before beginning the experiment, two researchers - a Ph.D. and a master's student - independently reviewed the selected snippets for clarity and relevance, resolving disagreements with a third reviewer. This diverse review team ensured that each snippet was suitable for data collection while benefiting from different levels of academic and practical expertise.

4.2.5 Conducting the Pilot Study

We designed a survey to collect developers' background information and their perceptions while reviewing a code snippet. We piloted this survey with two practitioners to estimate its length and clarity. The pilot study was carried out based on the guidelines provided by Sharafi et al. [32]: *(i)* we ensured that the eye tracker and room were set up correctly; *(ii)* we verified that the recording process properly acquired and saved data to disk; *(iii)* we checked the quality of the recorded data to ensure that the lighting conditions were appropriate for capturing eye movements; *(vi)* we observed how the participant reacted to the setup and tasks; *(v)* we recorded the time taken by the participant to complete the study; and *(vi)* we analyzed the data to evaluate the results and prevent any data loss.

The first participant took 90 minutes to analyze 16 code snippets and noted fatigue, discomfort with the chair, and errors found in the forms to be filled out. This feedback led to minor adjustments. Due to the extensive duration of the pilot, we decided to focus on 3 code smells and select 13 code snippets. As a result, the duration of the second pilot was reduced to 60 minutes, including the time allocated for completing a questionnaire on the researcher's background, which was conducted after the code snippet analysis.

4.2.6 Call for Participants

We selected participants under two constraints: *(i)* they must have contact with Java or other similar syntax programming language, and *(ii)* they must have been in contact with the topic of code smells. With these minimum requirements,

we seek to ensure that the participants could achieve a minimum understanding of the code snippet. They would also have a comprehension of the importance of software quality assurance practices. Recruitment spanned universities, companies, and a snowballing strategy [47], where participants referred qualified peers from their professional networks. We use the snowballing strategy [47], asking each participant to refer our survey to colleagues with similar experiences and interest in joining. The final sample included developers with programming experience ranging from a few months to over a decade. Detailed participant data, including demographics and experience, is provided in the supplementary material [9].

4.2.7 Execution of the Experiment

The survey was carried out via Google Forms, containing both multiple-choice and free-text questions. It begins by outlining the survey's purpose and research goals, emphasizing the confidentiality of participants' responses. The survey is divided into three phases. The first phase presents the types of code smells and the corresponding definitions. The second phase involves analyzing the code snippets using the eye tracker. In the third phase, participants are asked about their feelings during and after the analysis and their perceptions regarding the use of biosensors. Additionally, to better understand their background, participants' knowledge of the software engineering area is collected. The data collected in the third phase are little explored in this study and can be explored in future researches. The survey and all collected data are available in our supplementary material [9]. We detail each of the three phases as follows.

4.2.7.1 First Phase: Introduction to Code Smells

In the first phase, the developer is introduced to the concept of code smell and seven types of code smells, namely *Long Method*, *Data Class*, *Duplicate Code*, *Data Clumps*, *Feature Envy*, *Refused Bequest*, and *Message Chains*. We include a wider variety of code smells to ensure no bias in the responses. Thus, the participants could also look for these smells in the code snippets of the experiment. These measures were also adopted to establish a knowledge base, as many are aware of code smells but do not know how to name them correctly. Thus, we ensured that all participants had the same view of what is considered to be each smell type.

4.2.7.2 Second Phase: Analysis of Code Snippets

In this phase, the developers performed the analysis of the code snippets. The first code snippet was an example so that the participant could clarify any doubts with the researcher. In subsequent sections, was carried out the analysis without any

interference, aiming for ensuring the experimental integrity of the data. We presented the code snippets in the same order for all participants. To mitigate the risk of potential order effects, such as increased learning or mental fatigue, we diversified the types of code smells and their severities, throughout the experiment. We also added attention-check questions throughout the experiment to ensure participants remained engaged and attentive during the experiment. Consistent response patterns across metrics, such as time taken and fixation values, indicated that neither learning effects nor mental fatigue significantly influenced the results.

During the analysis of each code snippet, participants were asked to *(i)* describe how the code snippet works, *(ii)* whether it was difficult or not to understand and explain why, *(iii)* if it has any code smell and, if so (one or more), what is its severity and why this severity was chosen, and, lastly, *(iv)* how the participant felt (1 for very uncomfortable to 5 for very comfortable) when analyzing the code snippet.

4.2.7.3 Third Phase: Collecting Participant Background Data

In the third phase, the participant responded to a subset of questions derived from the original MLCQ article [10]. We explored aspects such as the duration of their programming career, their experience with software development, the programming languages they are familiar with, and their knowledge of the concept of code smells, among others. This data is intended to allow future research to establish a link between both studies. The background information was collected at the end of the experiment, rather than at the beginning, to avoid any impact of participant fatigue on the eye-tracking data. Collecting background data at the end ensured that participants could perform the main experimental tasks without prior cognitive fatigue, thus preserving the reliability of physiological measures. Additionally, we ask participants' opinions on the utility of the data gathered by the eye tracker in understanding their assessments.

4.2.8 Data Analysis

The data analysis for this study combines quantitative and qualitative approaches to address each of the RQs established in Section 4.2.1. The developers' responses, gathered through a structured Google Form (see Section 4.2.7.2), provide valuable qualitative insights into their subjective experiences and perceived comprehension difficulties when analyzing code snippets. This information is crucial for answering RQ1, as it captures developers' impressions of code complexity and difficulty, especially concern the presence of code smells. Additionally, objective data collected through an eye tracker – such as fixation duration, fixation count, and scan paths – enables us to address RQ2 and RQ3, focusing on the cognitive and be-

havioral aspects of code review. For these questions, we use the statistical analysis techniques that will be present in detail in each research question.

It is important to note that developers could classify code snippets with any code smell type (Section 4.2.7.1), including those beyond the three main types we focused on (Long Method, Feature Envy, and Data Class). Consequently, when a participant identified a code smell that did not fall within the types we focused on, we categorized it under "Others" for data analysis purposes. This categorization allowed us to maintain a clear focus on our primary set of smell types while acknowledging additional code smells that developers perceived as present and impactful. In the remaining of this section, we outline our methodology for analyzing both qualitative and quantitative data, systematically detailing the steps taken to address each research question.

4.2.8.1 Qualitative Coding of Developer Responses

To address RQ1, which investigates the relationship between developers' perceived difficulty in code comprehension and the presence of code smells, we conducted a qualitative coding analysis of developers' open-ended responses. We analyze the answer to how the code snippet works, whether and why it was difficult or not to understand, and whether there is a code smell.

We employed a qualitative coding process guided by principles of the Grounded Theory approach, specifically following the processes of open and axial coding [4]. While our analysis was inspired by Grounded Theory, we did not implement the full methodology, such as theoretical sampling or the iterative development of theory. Instead, we applied key Grounded Theory techniques, such as thematically coding open-ended responses, categorizing participants' explanations for comprehension difficulty, and analyzing the frequency of these factors in relation to specific code smells. Three experienced Java developers contributed to the open coding phase.

Specifically, to analyze the level of understanding of each participant based on their responses to the following question: *Please, describe how the code snippet works*, we identified four fine-grained categories of understanding levels of the code's functionality, ranging from *no understanding* to *full comprehension*. Table 4.1 summarizes the four levels. Concerning the question: *Why was it difficult or not to understand?*, we identified 14 categories based on participants' responses. Table 4.2 summarizes these categories, including their descriptions and quote samples.

After completing the open coding, we progressed to axial coding. Axial coding helped us on establishing relationships between the codes, allowing us to form an integrated structure of categories and subcategories that reflect the key factors influencing code comprehension. This iterative coding process allowed us to identify recurring themes in developers' responses, highlighting patterns in how code smells impact comprehension difficulty and providing insights into the specific aspects of code that developers find challenging.

Table 4.1: Levels of Understanding of Code Functionality

Level of Understanding	**Description**
No Understanding	The individual did not demonstrate understanding of the basic functions of the code, its syntax, or logic, or stated that they did not understand it.
Basic	The individual understands the basic function of the code, its objective, and what it does at a high level, but does not describe how this is implemented.
Intermediate	The individual described what the functions or methods do, demonstrating knowledge of the basic flow of execution (loops, conditionals) but did not describe the code in detail for more complex scenarios.
Full	The individual describes the functionality of the code in depth, including aspects of performance, optimization, and maintainability.

Table 4.2: Categories and Quote Samples for Difficulty in Understanding

Category	**Description**	**Quote Sample**
Application Domain Knowledge	Mentions the necessary understanding of the domain or specific area in which the software is being developed.	[...] code is difficult to understand, especially with little knowledge of the application domain. [...]
Class or Method Cohesion and Responsibility	Mentions when a class or method is doing too many things at once or when a responsibility is poorly distributed.	[...] since it has more functionality than just writing content to a file.
Code Length	Mentions about the size of the method or class.	code is extremely verbose [...]
Coding Style Inconsistency	Refers to inconsistencies in coding style, such as different conventions or patterns within the same code segment.	[...] lines are squished together without following standard indentation [...]
Complex Control Structure	Mentions comments on control structures (if-else, switches, loops, etc.) that are difficult to follow.	[...] has multiple specific conditions
Confusing Names	Mentions confusing names or those that do not follow clear conventions, such as variable, method, or bad class names.	variables are poorly named
Use of Encapsulation	Comments on the use of encapsulation, making it difficult to understand the behavior of the code.	contains too much encapsulation [...]
Complex Dependencies and Inter-class Interactions	Difficulty understanding dependencies or relationships between classes, and chain of calls.	[...] the use of so many methods that are not part of the class left me confused.
Excessive Variables or Parameters	Mentions code with "too many variables" or "excessive parameters", indicating that the code is overloaded.	The 'filename' variable requires several things to be initialized [...]
Lack or Absence of Adequate Comments	Mentions the absence of comments or comments that are confusing or outdated.	[...] the comments were not very enlightening
Language Syntax	Mentions a lack of knowledge about the more complex rules and structures of the language.	lack of experience with the language and unfamiliarity with methods used makes me a bit confused
Logical Complexity	Mentions related to internal logic being difficult to follow or understand, such as the presence of multiple nested loops, complex conditionals, or code branching.	[...] following a more complex path than necessary for writing to a file
Use of Polymorphism	Comments on the use of polymorphism, making it difficult to understand the behavior of the code.	Java syntax allows methods with the same name but different signatures
Use of Recursion	Comments on the use of recursion, making it difficult to understand the behavior of the code.	[...] potential infinite recursion [...]

4.2.8.2 Cognitive Effort

To answer RQ2, which investigates developers' cognitive effort while analyzing code snippets with and without code smells, we analyzed two primary metrics: Average Fixation Duration (AFD) and Fixation Count (FC) [32]. By multiplying these metrics (AFD * FC), we derived an estimate of cognitive load, which reflects the level of mental effort developers expend while reviewing code.

To ensure the accuracy of our cognitive load data, we conducted a thorough preprocessing and outlier treatment on all fixation and AFD * FC data points. Guided by Songwon Seo's flowchart for outlier detection [60], we first assessed the asymmetry of the fixation data distribution by applying the medcouple skewness statistic, which identifies potential skewness and the need for adjusted thresholds. Following this, we used an adjusted boxplot to detect and mark outliers, a method that allows for refined thresholds based on the skewness level identified. Once this initial analysis was complete, we applied the same outlier detection procedure to the combined AFD * FC data, refining the measurement of cognitive load. Finally, all identified outliers were winsorized [61], a method that replaces extreme values with a set threshold to reduce the influence of anomalous data points without distorting the dataset's overall structure. This approach allowed us to identify which code snippets classified as code smells imposed a higher cognitive load on developers, offering insights into specific code characteristics that demand greater mental effort.

We conducted statistical analyses to compare cognitive load between code snippets with and without code smells. First, we applied Levene's test to assess variance equality between groups. Based on these results, we employed the Mann-Whitney U test to analyze differences in cognitive metrics between snippets with and without smells. Additionally, we conducted ANOVA to examine differences across all code smell types and clean code. When ANOVA indicated significant differences, we performed Tukey's HSD post-hoc test to identify which specific pairs of smell types exhibited meaningful differences in cognitive load.

4.2.8.3 Mental Models and Reading Patterns

To answer RQ3, we were inspired by the framework proposed by Abid et al. [56], which explores the mental models and reading patterns developers use during program comprehension tasks. This approach allowed us to analyze how developers process and interpret code.

For examining mental models, we analyzed developers' gaze movements across chunks. Chunks are defined as continuous segments of code representing a cohesive unit or functionality within a method or function. These chunks are normally at most 10 LOC and allow us to segment developers' reading behavior, providing insights into how they navigate through different parts of the code and identify key areas of focus.

Each chunk was manually defined to ensure it represented a logical block of code, facilitating the analysis of developers' transitions between different code sections. An illustration of these chunks can be found in our supplementary material [9].

We focused on two mental models for program comprehension: the top-down and bottom-up models [62]. Given that our study involved code snippets embedded within larger Java files, we categorized gaze patterns based on whether they occurred inside or outside the code snippet. This distinction created four categories of mental model interactions:

- *Top-Down Inside.* Movements between chunks within the target code snippet, indicative of an exploratory reading strategy.
- *Bottom-Up Inside.* Fixations within the same chunk of the target code snippet, reflecting a detailed and focused reading approach.
- *Top-Down Outside.* Movements to areas outside the target code snippet, potentially to gather additional contextual information.
- *Bottom-Up Outside.* Fixations outside the target code snippet, indicating an examination of related code sections.

It is important to highlight that the framework proposed by Abid et al. [56] provides a simplified method for associating reading patterns with cognitive models, distinguishing between top-down and bottom-up approaches. However, this model does not capture the full complexity of cognitive processes during program comprehension. For instance, while chunk-to-chunk transitions are classified as top-down behaviors, they may also reflect exploratory navigation rather than hypothesis testing. Similarly, within-chunk fixation patterns do not necessarily indicate a sequential and detailed bottom-up examination. These limitations shows the exploratory nature of this framework in our study and emphasize the need for caution when interpreting the results.

For examining reading patterns, we adopt three approaches [63]:

- *Top-to-Bottom vs. Bottom-to-Top.* Reflects the directionality of reading. Top-to-bottom reading flows naturally from the beginning to the end of the code, line by line. Bottom-to-top reading, moves in reverse, as the developer reads upwards through the code lines.
- *Skimming vs. Thorough.* Indicates reading depth. Skimming is a quick scan, where the developer briefly glances over lines, focusing selectively on certain cues or sections. Thorough reading involves careful inspection of each line within a segment. We used a 1,000ms fixation threshold: fixations shorter than 1,000ms indicate skimming, while longer fixations suggest thorough reading.
- *Disorderly vs. Sectionally.* Reflects reading organization. Disorderly reading is marked by non-linear jumps across code sections, suggesting a less structured approach. Sectionally reading involves systematically inspecting contiguous

blocks of code. For this study, sections were defined as groups of three consecutive lines.

To address RQ3, we applied the same data processing and cleaning approach used for RQ2. This included steps for identifying and handling outliers, normalizing data points, and ensuring data reliability. Thus, it facilitated meaningful comparisons between cognitive effort and reading patterns across different code snippets.

4.2.9 Data Collection and Availability

To collect data from participants, the Tobii TX300 [52] eye tracker was used along with iTrace-core software [6]. In addition, we use Eclipse IDE[1] to present the code snippet with iTrace Eclipse plugin [6]. We also used the iTrace ToolKit [42], a complementary tool to process raw data. This toolkit was crucial in creating a database and calculating fixations[2]. Google Forms was used to collect responses from developers who participated in the experiment and information about their background. All files we used for the elaboration of the study and to display the graphics and data tables present in this paper can be accessed at our GitHub repository [9].

4.3 Results

The dataset reveals a heterogeneous sample that ranges from individuals with secondary-level education to those holding doctoral degrees, and from novices with minimal coding experience to professionals boasting over two decades of expertise in roles such as junior developer, software engineer and project manager. Also, while most participants demonstrate familiarity with code review practices and with associated tools (including SonarQube, PyLint, ESLint, and various static analysis software) used to identify code smells, only a minority consistently engage in formal, systematic code review processes.

In this section, we discuss the results of the research questions, focusing on how the presence of code smells influence developers' comprehension difficulty, cognitive effort, and reading behavior.

4.3.1 Impact of Code Smells on Developers' Perceived Comprehension Difficulty

To answer RQ1, we characterized: *(i)* the levels of understanding of code functionality (see Table 4.1); and *(ii)* the reasons described by developers why the code snippet was difficult or not to understand (see Table 4.2). Table 4.3 summarizes

[1]https://www.eclipse.org/ide/

[2]We used the I-VT algorithm [42], recommended for eye trackers with a refresh rate higher than 200Hz aligning with our equipment.

the analysis performed on developers' perceptions of difficulty in understanding code with different types of smells. The 1st column lists each code comprehension difficulty category according to Table 4.2. The 2nd column shows the total number of times each difficulty category was mentioned by participants, regardless of the number of code smells identified. The 3rd to 7th columns represent mentions of specific code smells, *Data Class* (DC), *Long Method* (LM), *Feature Envy* (FE), as well as *Others* and *None*. The 8th column aggregates only the mentions of specific code smells (DC+LM+FE), excluding *None* and *Others*. Finally, the 9th column gives the total number of mentions, including all categories (smells and non-smells). A single response could include multiple mentions of different code smells, which explains why the total in the 9th column may differ from the value in the 2nd column.

Table 4.3: Distribution of Code Smells grouped by Comprehension Difficulty Categories and Self-admitted Difficulty

Self-admitted Difficulty (Yes)								
Category	**#cited**	**DC**	**LM**	**FE**	**Others**	**None**	**Total Smells Only**	**Total All**
Application Domain Knowledge	2	1	0	0	0	1	1	2
Class or Method Cohesion and Responsibility	9	2	5	2	3	2	9	14
Code Length	17	0	15	5	0	1	20	21
Coding Style Inconsistency	6	0	0	1	1	4	1	6
Complex Control Structure	3	0	3	0	0	0	3	3
Confusing Names	15	2	3	3	4	6	8	18
Use of Encapsulation	2	1	1	1	0	0	3	3
Complex Dependencies and Inter-class Interactions	7	0	2	2	1	3	4	8
Excessive Variables or Parameters	5	1	3	2	1	1	6	8
Lack or Absence of Adequate Comments	11	0	3	3	3	3	6	12
Language Syntax	10	0	1	0	3	6	1	10
Logical Complexity	20	2	6	4	1	10	12	23
Use of Polymorphism	2	0	1	0	1	0	1	2
Use of Recursion	2	0	1	0	1	0	1	2
Total	**111**	**9**	**44**	**23**	**19**	**37**	**76**	**132**
Self-admitted Difficulty (No)								
Category	**#cited**	**DC**	**LM**	**FE**	**Others**	**None**	**Total Smells Only**	**Total All**
Application Domain Knowledge	0	0	0	0	0	0	0	0
Class or Method Cohesion and Responsibility	0	0	0	0	0	0	0	0
Code Length	4	0	1	0	0	2	1	3
Coding Style Inconsistency	0	0	0	0	0	0	0	0
Complex Control Structure	1	0	1	0	0	0	1	1
Confusing Names	1	1	0	0	0	0	1	1
Use of Encapsulation	2	0	0	0	0	0	0	0
Complex Dependencies and Inter-class Interactions	0	0	0	0	0	0	0	0
Excessive Variables or Parameters	0	0	0	0	0	0	0	0
Lack or Absence of Adequate Comments	4	2	1	1	0	2	4	6
Language Syntax	1	0	0	0	0	1	0	1
Logical Complexity	7	2	2	1	2	2	5	9
Use of Polymorphism	0	0	0	0	0	0	0	0
Use of Recursion	1	0	0	1	0	0	1	1
Total	**21**	**5**	**5**	**3**	**2**	**7**	**13**	**22**

4.3.1.1 The Most Cited Categories of Self-admitted Code Comprehension Difficulty

By analyzing Table 4.3, we observe that among developers who self-admitted difficulty (yes), the top-3 most common comprehension difficulty categories are as follows: *Logical Complexity* was the most cited, with 20 instances, 12 DC+LM+FE and 23 in total (code snippet with or without code smell). The *Code Length* was the second most mentioned category, with 17 instances, 20 DC+LM+FE and 21 in total. Finally, the *Confusing Names* also appears prominently with 15 instances, 8 DC+LM+FE and 18 in total. The last one, suggest that unclear naming conventions represent a common struggle.

Among developers who did not self-admitted difficulties to understand the code snippet, *Logical Complexity* remained the most cited (7 instances, 5 DC+LM+FE, 9 total), followed by Code Length (4 instances, 1 DC+LM+FE, 3 total), and Lack of Adequate Comments (4 instances, 4 DC+LM+FE, 6 total). This suggests that even those who did not explicitly report difficulties still encountered comprehension challenges. *Code Length* was cited 4 times (1 LM and 3 in total) and *Lack or Absence of Adequate Comments* category appears as a non-difficulty issue mentioned 4 times (4 DC+LM+FE and 6 in total). This suggests that some developers may perceive this as a less critical problem. Even though there is a discrepant amount between the developers who self-admitted difficulty and the developers who did not, *Logical Complexity* and *Code Length* are present in both, which demonstrates that the developer may come across the presented categories. For *Logical Complexity* a developer who self-admitted difficulty said *"actions and checks contain a lot of information making it difficult to understand each one"* and the developer who did not self-admitted difficulty said *"Despite having several conditionals, the logic is relatively simple"*.

Table 4.3 highlights notable differences between groups. *Confusing Names* was cited 15 times (18 total) by those who self-admitted difficulty, compared to just once in the other group, indicating that developers who struggle with code smells are more likely to recognize naming issues. *Lack of Adequate Comments* was mentioned 11 times (12 total) in the self-admitted difficulty (Yes) group but only 4 times (6 total) in the other group. This indicate that comments play a crucial role in comprehension for developers facing challenges, *i.e.* who admit difficulty may view comments as decisive: they are either more important or problematic. *Code Length* was cited 17 times (21 total) by those reporting difficulty, versus only 4 times in the other group, reinforcing that long code fragments contribute to cognitive overload.

Some categories showed minimal differences or no mentions in the group that did not report difficulties, as the case of *Application Domain Knowledge*, *Class or Method Cohesion and Responsibility*, *Coding Style Inconsistency*, and *Complex Dependencies and Inter-class Interactions*, indicating these were not perceived as major obstacle. Similarly, *Use of Polymorphism* and *Use of Recursion* categories

were rarely cited in this group.

> Developers who self-admitted difficulty reported more instances of code smells, especially those smells related to *logical complexity*, *code length* and *naming*. *Confusing Names* was a major challenge for developers who self-admitted difficulty, but was rarely mentioned by those who did not, suggesting that struggling developers are more sensitive to unclear naming conventions.

4.3.1.2 Distribution of Code Smells Grouped by Comprehension Difficulty Categories and Self-admitted Difficulty

Referring back to Table 4.3, we can observe that LM is the most frequent code smell across categories in the (Yes) group, especially *Code Length* (15), *Logical Complexity* (6), and *Class or Method Cohesion and Responsibility* (5). The FE smell also appears frequently in the *Code Length* (5), *Logical Complexity* (4), and *Confusing Names* (3) categories.

DC appears most prominently in *Confusing Names* (2), *Logical Complexity* (2), and *Class or Method Cohesion and Responsibility* (2) categories, showing a more distributed pattern compared to other smell types. The *Others* category is most frequently associated with *Confusing Names* (4), followed by *Class or Method Cohesion and Responsibility* (3), *Lack or Absence of Adequate Comments* (3), and *Language Syntax* (3).

We observe that *Code Length* has a strong association with both LM (15) and FE (5), which suggests that extensive code is often perceived as problematic due to these specific smell types. Similarly, *Logical Complexity* shows significant occurrences across LM (6), FE (4), and even in code without smells (10), indicating that complexity challenges comprehension regardless of the presence of specific code smells.

Confusing Names appears as a difficulty category across all smell types (DC: 2, LM: 3, FE: 3, *Others*: 4), as well as in code marked as having no smells (6). This consistent presence suggests that naming conventions pose a universal challenge to code comprehension that transcends specific code smell categories. In the case of LM, extended implementations could result in generic or inconsistent naming as developers attempt to encapsulate multiple responsibilities within a single method. For FE, the reliance on external class data or behavior could lead to ambiguous names that fail to clearly convey the method's intent, reflecting its focus on cross-class interactions rather than cohesive functionality.

Despite the widespread impact of *Confusing Names* and *Comments* on code comprehension, state-of-the-art tools such as Organic [64], Designate [65], and PMD [25] do not explicitly consider naming and comment issues in their detection approach (mainly for DC, LM and FE). These tools primarily focus on structural aspects of the code, such as method length, class cohesion, and coupling, while

overlooking the cognitive challenges posed by inconsistent or ambiguous naming conventions. Our findings highlight the need for future research and tool enhancements that incorporate naming-related factors into automated analysis, improving support for developers in identifying and mitigating comprehension difficulties.

> LM appears frequently among developers with difficulties related to *Code Length*, *Logical Complexity*, and *Class or Method Cohesion and Responsibility*. FE appears frequently with *Code Length*, *Logical Complexity*, and *Confusing Names*. This suggests that focusing on these categories of difficulties to detect such smells may yield significant benefits, as they may compound cognitive challenges, such as increase comprehension times and mental strain.

4.3.1.3 Code Smells Frequency Grouped by Level of Comprehension and Self-admitted Difficulty

Table 4.4 shows the distribution of the code smells characterized by developers' level of comprehension and their self-admitted difficulty. The 1st column indicates the level of comprehension, ranging from *No Understanding* to *Full* (see Table 4.1). The 2nd column specifies the smell type identified. The 3nd column represents the number of instances where a developer self-admitted difficulty and identified the code smell. Finally, the last column indicates the number of instances where the developer did not self-admitted difficulty and identified the code smell.

Table 4.4: Code Smells Frequency grouped by Level of Comprehension and Self-admitted Difficulty

Level of Comprehension	Code Smell Type	Self-admitted Difficulty (Yes)	Self-admitted Difficulty (No)
No Understanding	None	13	2
	DC	5	5
	LM	20	3
	FE	8	2
	Others	5	0
Basic	None	5	2
	DC	2	0
	LM	17	1
	FE	10	0
	Others	6	1
Intermediate	None	12	3
	DC	0	0
	LM	6	1
	FE	4	1
	Others	5	0
Full	None	0	0
	DC	0	0
	LM	1	0
	FE	1	0
	Others	4	1

We can observe that LM is the most frequently reported code smell across all levels of comprehension, especially in the *No Understanding* (with 20 instances

in the difficulty group), and *Basic* (with 17 instances in the difficulty group). This might suggest that LM is a key factor in the comprehension difficulties to developers who do not understand the code or with a basic understanding. A similar behavior applies to the FE smell. DC appears less frequently when compared to all types, which suggests that DC may be less perceptible or considered less influential to code comprehension.

We also can observe that developers with *No Understanding* and *Basic* level of comprehension reported more difficulties across almost all code smells compared to those who marked (No) difficulty. This might indicate a strong correlation between code smells and perceived difficulty at these levels. Regarding the *Intermediate* level of comprehension there is a slightly lower frequency of self-admitted difficulty in comparison to the *Basic* level, suggesting an improved comprehension that may mitigate some issues caused by code smells. Finally, in the *Full* comprehension level, we can observe that few smells are noted, and the difficulty (yes) is minimal, indicating that code comprehension may reduce the perceived impact of code smells.

None is also present in all levels of comprehension, except for level *Full*. While the *No Understanding* (with 13 instances in the difficulty group) level suggest a lack of knowledge of the developers, the *Intermediate* (with 12 instances in the difficulty group) level suggest that even when the code snippet do not have smells in their opinion, they can have trouble to understand the code.

> LM stands out, especially at lower comprehension levels, 20 at *No Understanding* and 17 at *Basic* levels with self-admitted difficulty. FE is similar but with a lower frequency. Overall, self-admitted difficulty decreases across all smell types when the comprehension level increases from *Basic* to *Full*, suggesting experienced developers can better navigate smelly code. Removing LM and FE requires increased support for the affected code comprehension, particularly for novice developers or team members unfamiliar with the codebase. DC may be less perceptible or considered less influential to code comprehension.

4.3.2 Impact of Code Smells on Developers' Cognitive Load

To answer RQ2, we analyzed the influence of code smells on the cognitive effort required for developers to analyze code snippets. Initially, we conducted Levene's test to assess the equality of variances between code snippets with and without code smells. The null hypothesis (H0) for Levene's test states that *there is no variances difference between these two groups*. With a test statistic of 5.109 and a p-value of 0.0245, we reject the null hypothesis, indicating that the variances are statistically different between the groups. This result allowed us to proceed with a Mann-Whitney U test. The null hypothesis (H0) for the test states that *there is no significant difference between AFC*FC of these groups*. The test yielded a test

statistic of 15290.0 and a p-value of 0.00018, leading us to reject the null hypothesis. This finding demonstrated a significant difference in cognitive load between code snippets containing smells and those without smells, indicating that the presence of code smells generally increased cognitive effort. This finding is supported by a follow-up ANOVA test comparing different smell types and no smell, which yielded an F-statistic of 5.83 and a p-value of 0.00072. This result confirms significant differences in cognitive load across code smell types and/or clean code, prompting further analysis to identify which types of smells contributed most to higher cognitive effort.

Table 4.5: Multiple Comparison of Means - Tukey HSD, FWER=0.05

group1	**group2**	**meandiff**	**p-adj**	**lower**	**upper**	**reject**
DC	FE	22947.33	0.702	-26107.38	72002.03	False
DC	**LM**	**46419.84**	**0.037**	**1798.79**	**91040.89**	**True**
DC	None	2215.44	0.999	-35441.38	39872.25	False
DC	**Others**	**59478.78**	**0.003**	**14451.13**	**104506.44**	**True**
FE	LM	23472.51	0.631	-22663.65	69608.66	False
FE	None	-20731.89	0.601	-60172.26	18708.47	False
FE	Others	36531.45	0.200	-9998.07	83060.97	False
LM	**None**	**-44204.40**	**0.004**	**-77971.50**	**-10437.31**	**True**
LM	Others	13058.94	0.912	-28770.12	54888.01	False
None	**Others**	**57263.35**	**<0.001**	**22960.75**	**91565.94**	**True**

To identify which specific pairs of groups exhibit significant differences in cognitive load, we applied Tukey's HSD test (see Table 4.5). The results revealed that the LM smell type led to a significantly greater cognitive load compared to both the None category ($p = 0.004$) and the DC type ($p = 0.037$). This suggests that LM smells demand more cognitive effort, likely due to the extended or complex logic often present in lengthy methods. The None category showed significantly lower cognitive load compared to Others ($p < 0.001$), highlighting that code without smells is easier for developers to process than code with ambiguous or less specific structural indicators. Furthermore, the Others group imposed a higher cognitive load than DC ($p = 0.003$), indicating that smells categorized as Others also contribute to increased cognitive demands, possibly due to their varied and potentially ambiguous structures. This finding also motivates further research studies involving the comparison of more code smell types. For FE, there was no evidence when compared with others smells.

Although this study was conducted with the goal of understanding developers' perceptions, we also performed statistical tests to assess whether the developers' classifications were correct or incorrect. This analysis allows us to examine whether cognitive effort differs between those who correctly identified the presence of code smells and those who did not. To this end, we evaluated the variance between the two groups and, based on the results, applied either the Mann-Whitney U test or

the t-test to verify the null hypothesis (H0), using the same null hypothesis applied in the previous analysis.

The results of these statistical tests are presented in Table 4.6. In all cases, the null hypotheses were not rejected, suggesting that the cognitive effort required to analyze the code does not significantly differ between developers who correctly identified the code smells and those who did not. This finding indicates that the way developers perceive the presence of code smells in the code is what primarily influences their cognitive effort, regardless of the accuracy of their classification.

Table 4.6: Test Results for Cognitive Effort - Correct X Incorrect

Smell	**Test**	**statistic**	**p-value**	**reject**
LM	Mann-Whitney U	270.0	0.588	False
DC	t-test	-1.3482	0.186	False
FE	t-test	-0.3693	0.714	False
None	t-test	-0.8658	0.387	False

LM smells substantially increase the cognitive effort required for analysis, whereas DC smells do not impose a similar burden. Also, developers' perception of code smells plays a key role in cognitive effort, regardless of classification accuracy.

4.3.3 Impact of Code Smells on Developers' Reading Behavior

To answer RQ3, we analyze how the presence of code smells influences developers' reading behavior when analyzing code snippets. Thus, we evaluated the mental models and reading patterns learned by developers when classifying different smell types and their absence in the code snippets. Additionally, we applied the Wilcoxon test to validate the results. The null hypothesis (H0) states that *for each smell category (LM, DC, FE, Others, and None), there is no significant difference in the proportion of time spent in each of the following:* (4.3.3.1) *mental model (bottom-up vs. top-down);* (4.3.3.2) *reading direction (top-to-bottom vs. bottom-to-top);* (4.3.3.3) *organizational reading pattern (sectionally vs. disorderly);* (4.3.3.4) *depth of reading (thorough vs. skimming).*

4.3.3.1 Mental Models: Bottom-up vs. Top-down

Table 4.7 presents the detailed results of the Wilcoxon signed-rank test comparing the time spent using Bottom-up versus Top-down mental models for each perceived code smell category. The table shows the smell, the hypothesis, the

metric used for comparison, the mental model, the sample size (n), the test statistic (T) for each model, the Z-score (Z), and the p-value (p).

The results indicated that the H0 was rejected for LM, DC, FE, and Others, suggesting that developers when analyzing smelly code go through the bottom-up mental model. This indicates that developers may need detailed, line-by-line comprehension due to the increased complexity or ambiguity associated with these smells. For the None category, where no code smells were identified by developers, the hypothesis was not rejected, indicating no significant preference between bottom-up and top-down approaches in the absence of code smells.

Table 4.7: Comparison of Mental Models (Bottom-up vs Top-down) for Different Code Smell Types

Smell	**H**	**Metric**	**Model**	**n**	**T**	**Z**	**p**
LM	H0	Fix.	Bottom-up	49	3618.0	-8.49	<0.001
			Top-down	49	1233.0		
DC	H0	Fix.	Bottom-up	37	2000.0	-6.99	<0.001
			Top-down	37	775.0		
FE	H0	Fix.	Bottom-up	33	1603.0	-6.65	<0.001
			Top-down	33	608.0		
None	H0	Fix.	Bottom-up	148	31263.0	-13.73	<0.001
			Top-down	148	12693.0		
Others	H0	Fix.	Bottom-up	47	3329.0	-8.31	<0.001
			Top-down	47	1136.0		

4.3.3.2 Reading Patterns: Top-to-bottom vs. Bottom-to-top

Table 4.8 provides the detailed statistical results (Wilcoxon test: n, T, Z, p) for the comparisons of the three reading pattern pairs based on the developers' overall perception of the code smell type.

For all categories, H0 was rejected. This suggests that the presence of code smells and even the no presence affects the directional pattern of reading, with developers more frequently adopting a top-to-bottom approach. This likely reflects an attempt to gain an initial high-level understanding before diving into more detailed sections of the code. In other words, developers tend to follow the natural execution flow of the code, regardless of smell presence, indicating that the sequential nature of program comprehension remains consistent even when facing potentially problematic code structures.

Table 4.8: Comparison of Reading Patterns (Top-to-bottom vs Bottom-to-top, Sectionally vs Disordely, Thorough vs Skimming) for Different Code Smell Types by Developers Perception

Smell Type	Metric	Reading Pattern	n	T	Z	p
LM	Fix.	Top-to-bottom Bottom-to-top	49 49	1224 0	-6.030	<0.001
	Fix.	Sectionally Disordely	49 49	486 739	-1.258	0.212
	Fix.	Thorough Skimming	49 49	0 1225	-6.092	<0.001
DC	Fix.	Top-to-bottom Bottom-to-top	37 37	703 0	-5.302	<0.001
	Fix.	Sectionally Disordely	37 37	153 549	-2.969	0.003
	Fix.	Thorough Skimming	37 37	0 703	-5.302	<0.001
FE	Fix.	Top-to-bottom Bottom-to-top	33 33	561 0	-5.011	<0.001
	Fix.	Sectionally Disordely	33 33	260 301	-0.366	0.724
	Fix.	Thorough Skimming	33 33	1 560	-4.994	<0.001
None	Fix.	Top-to-bottom Bottom-to-top	148 148	11025 0	-10.517	<0.001
	Fix.	Sectionally Disordely	148 148	4743 6283	-1.473	0.140
	Fix.	Thorough Skimming	148 148	0 11026	-10.553	<0.001
Others	Fix.	Top-to-bottom Bottom-to-top	47 47	1128 0	-5.968	<0.001
	Fix.	Sectionally Disordely	47 47	761 367	-2.084	0.036
	Fix.	Thorough Skimming	47 47	0 11026	-5.968	<0.001

4.3.3.3 Reading Patterns: Sectionally vs. Disorderly

H0 was rejected for DC and Others, indicating that developers adopt a more sectionally organized approach for these types. For DC, this may be due to its data-driven structured nature, as it often contains straightforward data containers with minimal logic, allowing developers to follow a sequential, organized reading pattern without frequent cross-referencing. This approach is clearly different from smells like FE that may require more "disorderly" navigation, due to the nature of this smell that show more interest in other classes than their own, forcing developers to jump back and forth between multiple code locations to understand the relationships and dependencies. For LM, FE, and None, the hypothesis was not rejected, suggesting no consistent preference between sectionally and disorderly reading for these smells.

4.3.3.4 Reading Patterns: Thorough vs. Skimming

H0 was rejected for all code smell types, indicating a significant difference between thorough and skimming patterns. The rank sums show a clear preference for skimming across all code smells, suggesting that developers more frequently engaged in a quick scanning approach, selectively focusing on certain cues or sections rather than inspecting each line in detail. For example, when encountering an LM smell, developers tends to spend brief periods scanning through the method structure before diving deeper into specific areas of interest. This behavior suggests an initial approach where developers first identify key structural elements and potential problem areas before committing to a more detailed analysis.

4.3.3.5 Reading Patterns: Correct and Incorrect Classifications

We also analyzed the reading patterns when developers correctly and incorrectly classified the code snippet as having a smell, aiming to understand whether there are differences in the analyzed results when assessed solely based on their responses, without considering the dataset classification.

The test results can be found in Table 4.9. This table presents the Wilcoxon test statistics (n, T, Z, p) for each reading pattern pair side-by-side for instances where developers provided Correct Answers versus Incorrect Answers. In Table 4.10, we summarize the outcomes (Reject/Not Reject H0) of the Wilcoxon tests for each reading pattern pair across three perspectives: overall developer perception (Developers Perception), instances associated with developers' correct classifications (Correct), and instances associated with incorrect classifications (Incorrect). The main difference observed is in the *Disordely vs. Sectionally* reading pattern for the DC smell. While the null hypothesis was rejected in the overall analysis, indicating a more structured reading approach, it was not rejected when considering only incorrect classifications. This finding suggests that there is no significant evidence to indicate a consistent preference for an organized reading approach among developers who misclassified the smell. For the other reading patterns, the results remained consistent with the general perception.

> Developers who classify code snippets with LM, DC, and FE smells use a bottom-up mental model. DC encourages a sectionally organized reading pattern and developers predominantly use skimming across all smells, preferring a quick overview over a detailed inspection.

Table 4.9: Comparison of Reading Patterns (Top-to-bottom vs Bottom-to-top, Sectionally vs Disordely, Thorough vs Skimming) for Different Code Smell Types by Developers' Answer Correctness

Smell Type	**Reading Pattern**	**Correct Answers**				**Incorrect Answers**			
		n	**T**	**Z**	**p**	**n**	**T**	**Z**	**p**
LM	Top-to-bottom	35	629.0	-5.09	<0.001	14	105.0	-3.29	<0.001
	Bottom-to-top	35	0.0			14	0.0		
	Sectionally	35	227.0	-1.44	0.1535	14	54.0	-0.09	0.95
	Disordely	35	403.0			14	51.0		
	Thorough	35	630.0	-5.16	<0.001	14	105.0	-3.29	<0.001
	Skimming	35	0.0			14	0.0		
DC	Top-to-bottom	28	406.0	-4.62	<0.001	9	45.0	-2.66	0.0039
	Bottom-to-top	28	0.0			9	0.0		
	Sectionally	28	329.0	-2.86	0.0043	9	16.0	-0.77	0.49
	Disordely	28	76.0			9	29.0		
	Thorough	28	406.0	-4.62	<0.001	9	45.0	-2.66	0.0039
	Skimming	28	0.0			9	0.0		
FE	Top-to-bottom	6	21.0	-2.20	0.03125	27	378.0	-4.54	<0.001
	Bottom-to-top	6	0.0			27	0.0		
	Sectionally	6	14.0	-0.73	0.5625	27	185.0	-0.09	0.93
	Disordely	6	7.0			27	193.0		
	Thorough	6	21.0	-2.20	0.03125	27	377.0	-4.51	<0.001
	Skimming	6	0.0			27	1.0		
None	Top-to-bottom	49	1225.0	-6.09	<0.001	99	4949.0	-8.59	<0.001
	Bottom-to-top	49	0.0			99	0.0		
	Sectionally	49	612.5	0.0	1.0	99	1967.0	-1.77	0.076
	Disordely	49	612.5			99	2983.0		
	Thorough	49	1225.0	-6.09	<0.001	99	4950.0	-8.63	<0.001
	Skimming	49	0.0			99	0.0		

4.4 Threats to Validity

Our study focused on exploring participants' perceptions and cognitive responses to code snippets, rather than validating their feedback against the annotations provided in the MLCQ dataset. This reliance on participants' subjective identification of code smells introduces a potential threat to validity, as their perceptions (as expected) may not fully align with the predefined annotations. Future studies could address this by systematically comparing participants' feedback with the MLCQ dataset annotations, providing a deeper understanding of discrepancies between the perceived against the perception of more experinced engineers (who annotated the dataset).

There are many possible reasons to explain these possible discrepancies. Thus, the different programming skills, experience, and individual physiological conditions of developers can influence their responses to biosensors. To mitigate these threats, several measures were implemented, including ensuring participants had knowledge of the Java language, presenting the concept and types of smells prior to the analysis, and following a protocol designed to keep participants calm during the experiment.

Another threat to the validity of the study is the duration of the experiment,

Table 4.10: Test Hypothesis Compared for Reading Patterns

Smell Type	**Reading Pattern**	**Developers Perception**	**Correct**	**Incorrect**
LM	Top-to-bottom vs. Bottom-to-top	Reject	Reject	Reject
	Disordely vs. Sectionally	Not Reject	Not Reject	Not Reject
	Skimming vs. Thorough	Reject	Reject	Reject
DC	Top-to-bottom vs. Bottom-to-top	Reject	Reject	Reject
	Disordely vs. Sectionally	Reject	Reject	Not Reject
	Skimming vs. Thorough	Reject	Reject	Reject
FE	Top-to-bottom vs. Bottom-to-top	Reject	Reject	Reject
	Disordely vs. Sectionally	Not Reject	Not Reject	Not Reject
	Skimming vs. Thorough	Reject	Reject	Reject
None	Top-to-bottom vs. Bottom-to-top	Reject	Reject	Reject
	Disordely vs. Sectionally	Not Reject	Not Reject	Not Reject
	Skimming vs. Thorough	Reject	Reject	Reject

which can vary from 1 hour to 1 hour and 30 minutes. The participants needed to remain in their position so that the data captured by the eye tracker was reliable.

Another significant threat lies in the potential for over-interpretation of results. Given the controlled nature of the experiment and the limited scope of analyzed snippets (12), generalizing findings to broader programming contexts or larger-scale software systems must be approached cautiously. The study's conclusions should be framed as exploratory, providing insights rather than definitive answers.

The choice of Java as the language for the study may also limit the generalization of the results to other programming languages. Results would be different for smells in programming languages following other programming paradigms. By focusing on Java developers, we aim to explore the extent to which code smells influence the cognitive load and overall comprehension within a popular programming context. The structure of the code snippets would be quite similar if they were written in other object-oriented languages, such as C#.

Furthermore, the subjectivity inherent in participants' identification of code smells introduces variability in the data. This variability is particularly notable when comparing participant-identified smells to oracle annotations, which could affect the interpretation of cognitive and behavioral metrics. Future work should include a systematic comparison between these perspectives to validate and refine the findings.

The use of smaller code snippets was necessary to maintain a reasonable duration of the experiment, considering that the evaluation of all three types of code smells, and their severities already takes a significant amount of time. Expanding to larger code snippets could make the controlled experiment impractical in terms of duration and inaccurate data.

4.5 Conclusion

This study presents significant practical and theoretical implications for software engineering, particularly concerning program comprehension. Practically, our findings reveal the substantial cognitive load imposed by code smells like LM and FE. IDEs should thus prioritize detecting these smells and offer automated recommendations to ease cognitive strain. Also, confusing naming conventions frequently aggravate comprehension issues, highlighting the need for explicit naming standards and targeted developer training.

Observed reading behaviors revealed that developers typically adopt bottom-up mental models and engage predominantly in skimming, reflecting distinct comprehension patterns. This indicates a need for training programs tailored to encourage more structured and thorough reading strategies, especially benefiting novice developers. Organizations can leverage these insights to enhance onboarding and mentoring processes focused on managing cognitive challenges related to code smells.

By combining qualitative insights with quantitative eye-tracking data, we provide a comprehensive understanding of developers' cognitive interactions with smelly code. The methodological approach presented sets a foundation for future cognitive research in software comprehension. Future investigations should further examine developers' subjective perceptions alongside objective cognitive metrics, ultimately contributing to improved software quality and maintainability practices.

4.6 Summary

In this chapter, we expanded our investigation into the impact of code smells on developer comprehension by incorporating a qualitative analysis of developers' responses, in addition to the analysis of eye-tracking data. By employing qualitative coding techniques, we identified key themes and patterns in developers' explanations, providing deeper insights into how code smells influence their comprehension processes. Our findings confirm and extend the results presented in Chapter 3, reinforcing that code smells such as *Long Method* and *Feature Envy* impose higher cognitive effort compared to *Data Class*, which is perceived as less challenging by developers. Furthermore, the qualitative analysis revealed that developers frequently cite factors such as method complexity, dependency relationships, and naming conventions as critical elements affecting their comprehension. These insights offer a more comprehensive understanding of the cognitive challenges posed by code smells.

Despite these contributions, the study has certain limitations, such as the subjectivity inherent in qualitative coding and the potential influence of individual experience levels on the perceived difficulty of code snippets. Future work should consider expanding the dataset and refining the coding framework to improve the generalization of the findings. In the next chapter, we summarize the main

contributions of this dissertation, discussing how our findings contribute to the broader field of software engineering and present the key challenges identified throughout the study and outline opportunities for future research, including the development of improved code comprehension tools and strategies to mitigate the negative impact of code smells on developers' cognitive effort.

5
Conclusion

Code smells are widely recognized as indicators of poor design and implementation choices, significantly impacting developers' ability to comprehend and maintain software systems. While extensive research has been conducted on identifying and refactoring code smells, there remains a critical need to understand how these smells influence developers' cognitive processes and reading behaviors during code comprehension tasks. This dissertation addresses this gap by employing eye-tracking technology to empirically investigate the impact of code smells on developers' reasoning and subjective experiences during comprehension tasks, cognitive effort, visual attention, and reading patterns. The findings provide valuable insights into the cognitive challenges posed by three types of code smells and offer practical implications for improving software development tools and practices.

5.1
Summary of Contributions

This dissertation makes several key contributions to the field of software engineering, particularly in the areas of code quality, program comprehension, and developer tooling support. Below, we summarize these contributions and their significance:

Contribution 1: A Framework for Eye-Tracking Analysis of Code Smells. This dissertation introduces a novel framework for studying the impact of code smells on developers' cognitive processes using eye-tracking technology. By monitoring developers' visual attention, fixation durations, and reading patterns, we were able to quantify the cognitive effort required to analyze code snippets with and without code smells. This framework, detailed in Chapter 3 and improved in Chapter 4, provides a robust methodology for future studies aiming to explore the cognitive dimensions of software development tasks.

Contribution 2: Empirical Evidence on the Cognitive Impact of Code Smells. Through a series of controlled experiments, we collected empirical data on how different types of code smells – such as *Long Method*, *Feature Envy*, and *Data Class* – affect developers' cognitive load and comprehension. The results, presented in Chapter 4, reveal that *Long Method* and *Feature Envy* significantly increase cognitive effort, while *Data Class* imposes a comparatively lower burden. Our findings suggest that refactoring strategies for *Long Method* could prioritize extracting smaller, cohesive methods and enhancing naming clarity to reduce complexity,

as this smell's extended logic was shown to demand prolonged fixations and higher cognitive load. These findings contribute to a deeper understanding of how specific code smells challenge developers and highlight the need for targeted code refactoring strategies.

Contribution 3: Insights into Developers' Reading Patterns and Mental Models. Our analysis of eye-tracking data uncovered distinct reading patterns and mental models adopted by developers when analyzing smelly code. For instance, developers tended to use a bottom-up mental model when encountering *Long Method* and *Feature Envy* smells, focusing on detailed, line-by-line comprehension. In contrast, *Data Class* encouraged a more sectionally organized reading pattern. To support these patterns, an IDE feature could highlight critical code sections, such as method signatures in *Long Method* and dependencies in *Feature Envy* scenarios, with visual cues to guide developers toward key comprehension points, reducing the cognitive effort observed in our study. These insights, discussed in Chapter 4, provide a foundation for designing tools that support developers in navigating complex code structures, improving code readability and maintainability.

5.2 Implications of our Findings

The findings of this dissertation have important implications for researchers, tool developers, and practitioners in the field of software engineering. Below, we discuss these implications.

Implications for Researchers. The studies in this dissertation demonstrate the value of eye-tracking technology in understanding the cognitive processes involved in code comprehension. Researchers can build on this work by exploring additional metrics, such as pupil dilation and saccade length, to gain a more comprehensive view of developers' cognitive effort. Furthermore, the experimental framework developed in this dissertation can be applied to study other code smells, consider additional program languages; as well to study various other aspects of software development, such as debugging and code review.

Implications for Tool Developers. The empirical evidence gathered in this study provides a strong case for integrating cognitive load metrics into development tools. For instance, IDEs could use real-time eye-tracking data to identify code sections that require excessive cognitive effort and suggest refactoring opportunities. Additionally, tools could be developed to train developers in recognizing and addressing code smells, particularly those that impose the highest cognitive burden.

Implications for Practitioners. For software development teams, the findings of this study underscore the importance of writing clean, well-structured code to reduce cognitive load and improve maintainability. Practitioners can use the insights from this research to prioritize refactoring efforts, focusing on code smells that are most detrimental to comprehension. Additionally, the study highlights the importance of clear naming conventions and modular design in enhancing code readability.

5.3 Future Works

In this section, we describe several avenues for future research:

- **Expansion to Other Code Smells:** This study focused on three types of code smells (*Long Method*, *Feature Envy*, and *Data Class*). Future work could explore the cognitive impact of other common smells in industrial projects, such as *Duplicate Code* or *God Class*.
- **Application to Different Programming Languages:** The experiments in this study were conducted using Java code snippets. Future research could investigate whether the findings hold for other programming languages, such as Python, Lua or C++.
- **Integration of Additional Metrics:** Incorporating metrics such as pupil dilation and blink rate could provide a more nuanced understanding of developers' cognitive states during code comprehension tasks.
- **Longitudinal Studies:** Conducting longitudinal studies to observe how developers' cognitive patterns evolve over time could provide insights into the long-term effects of code smells on software maintenance and evolution.
- **Analysis of Participants' Background with other data**: Future studies could investigate how developers' level of experience, educational background, or familiarity with programming paradigms influence their cognitive responses to code smells. Integrating these data with eye-tracking metrics may provide deeper insights into how different developer profiles handle problematic code.
- **Validation in Industrial Settings:** While this study was conducted in a controlled environment, future research could validate the findings in real-world industrial settings to ensure their applicability to large-scale software projects.

Limitations This study has several limitations that should be addressed in future work as was presented in the Chapters 3 and 4. First, the use of small code snippets may not fully capture the complexity of real-world software systems. Second, the reliance on participants' subjective identification of code smells introduces variability in the data. Finally, the study's focus on Java developers may limit the generalizability of the findings to other programming contexts.

5.4 Conclusion

This dissertation advances our understanding of how code smells impact developers' cognitive processes during code comprehension tasks. By leveraging eye-tracking technology, we have provided empirical evidence on the cognitive effort required to analyze smelly code and identified specific reading patterns and mental models adopted by developers. These findings have important implications for the design of development tools and the improvement of software maintenance practices. Future research can build on this work to further explore the cognitive dimensions of software development and develop more effective strategies for managing code quality.

Data availability statement

All scripts and data used in this study are available in the package [9].

Bibliography

[1] OLIVEIRA, D.; ASSUNÇÃO, W. K. G.; GARCIA, A.; BIBIANO, A. C.; RIBEIRO, M.; GHEYI, R. ; FONSECA, B.. **The untold story of code refactoring customizations in practice**. In: 2023 IEEE/ACM 45TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING (ICSE), p. 108–120, 2023.

[2] FOWLER, M.; BECK, K.. **Refactoring: Improving the Design of Existing Code**. Addison-Wesley Longman Publishing Co., Inc., USA, 1999.

[3] MÄNTYLÄ, M. V.; LASSENIUS, C.. **Subjective evaluation of software evolvability using code smells: An empirical study**. Empirical Software Engineering, 11:395–431, 2006.

[4] DAS, D.; MARUF, A. A.; ISLAM, R.; LAMBARIA, N.; KIM, S.; ABDELFATTAH, A. S.; CERNY, T.; FRAJTAK, K.; BURES, M. ; TISNOVSKY, P.. **Technical debt resulting from architectural degradation and code smells: a systematic mapping study**. SIGAPP Appl. Comput. Rev., 21(4):20–36, jan 2022.

[5] SHARAFI, Z.; SOH, Z. ; GUÉHÉNEUC, Y.-G.. **A systematic literature review on the usage of eye-tracking in software engineering**. Information and Software Technology, 67:79–107, 2015.

[6] SHAFFER, T. R.; WISE, J. L.; WALTERS, B. M.; MÜLLER, S. C.; FALCONE, M. ; SHARIF, B.. **Itrace: Enabling eye tracking on software artifacts within the ide to support software engineering tasks**. In: PROCEEDINGS OF THE 2015 10TH JOINT MEETING ON FOUNDATIONS OF SOFTWARE ENGINEERING, ESEC/FSE 2015, p. 954–957, New York, NY, USA, 2015. Association for Computing Machinery.

[7] SHARAFI, Z.; SHAFFER, T.; BONITA, S. ; GUÉHÉNEUC, Y.. **Eye-tracking metrics in software engineering**. In: PROCEEDINGS OF THE 22ND ASIA-PACIFIC SOFTWARE ENGINEERING CONFERENCE, APSEC '15. IEEE CS Press, 2015.

[8] MARTINS, V.; RAMOS, P. L. V.; NEVES, B. B.; LIMA, M. V.; ARRIEL, J.; GODINHO, J. V.; RIBEIRO, J.; GARCIA, A. ; PEREIRA, J. A.. **Eyes on Code Smells: Analyzing Developers' Responses During Code Snippet Analysis**. `https://github.com/aisepucrio/EoCS`, 2024. Accessed: 2024-07-25.

[9] **Reading between the smells: Eye-tracking developer responses to code smells**. `https://github.com/aisepucrio/EoS-emse2025`, 2025.

[10] MADEYSKI, L.; LEWOWSKI, T.. **MLCQ: Industry-relevant code smell data set**. In: PROCEEDINGS OF THE EVALUATION AND ASSESSMENT IN SOFTWARE ENGINEERING (EASE '20), p. 342–347, New York, NY, USA, 2020. Association for Computing Machinery.

[11] SILVA DA COSTA, J. A.; GHEYI, R.. **Evaluating the code comprehension of novices with eye tracking**. In: PROCEEDINGS OF THE XXII BRAZILIAN SYMPOSIUM ON SOFTWARE QUALITY, SBQS '23, p. 332–341, New York, NY, USA, 2023. Association for Computing Machinery.

[12] MARINESCU, R.. **Detection strategies: metrics-based rules for detecting design flaws**. In: 20TH IEEE INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE, 2004. PROCEEDINGS., p. 350–359, 2004.

[13] PEREPLETCHIKOV, M.; RYAN, C.. **A controlled experiment for evaluating the impact of coupling on the maintainability of service-oriented software**. IEEE Transactions on Software Engineering, 37(4):449–465, July 2011.

[14] BAVOTA, G.; QUSEF, A.; OLIVETO, R.; DE LUCIA, A. ; BINKLEY, D.. **An empirical analysis of the distribution of unit test smells and their impact on software maintenance**. In: 2012 28TH IEEE INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE (ICSM), p. 56–65, Sep. 2012.

[15] SANTOS, J. A. M.; ROCHA-JUNIOR, J. B.; PRATES, L. C. L.; NASCIMENTO, R. S. D.; FREITAS, M. F. ; MENDONÇA, M. G. D.. **A systematic review on the code smell effect**. Journal of Systems and Software, 144:450–477, 2018.

[16] PALOMBA, F.; DI NUCCI, D.; TUFANO, M.; BAVOTA, G.; OLIVETO, R.; POSHYVANYK, D. ; DE LUCIA, A.. **Landfill: An open dataset of code smells with public evaluation**. In: 2015 IEEE/ACM 12TH WORKING CONFERENCE ON MINING SOFTWARE REPOSITORIES, p. 482–485, 2015.

[17] XU, W.; ZHANG, X.. **Multi-granularity code smell detection using deep learning method based on abstract syntax tree**. In: PROCEEDINGS OF THE 33RD INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING KNOWLEDGE ENGINEERING (SEKE), p. 503–509, 2021.

[18] MÄNTYLÄ, M.. **Bad Smells in Software - A Taxonomy and an Empirical Study**. PhD thesis, Helsinki University of Technology, 2003.

[19] YAMASHITA, A.; MOONEN, L.. **To what extent can maintenance problems be predicted by code smell detection? – an empirical study**. Information and Software Technology, 55(12):2223–2242, 2013.

[20] PALOMBA, F.; BAVOTA, G.; DI PENTA, M.; OLIVETO, R.; DE LUCIA, A. ; POSHYVANYK, D.. **Detecting bad smells in source code using change history information**. In: PROCEEDINGS OF THE 28TH IEEE/ACM INTERNATIONAL CONFERENCE ON AUTOMATED SOFTWARE ENGINEERING, ASE '13, p. 268–278. IEEE Press, 2013.

[21] SOBRINHO, E. V. D. P.; DE LUCIA, A. ; MAIA, M. D. A.. **A systematic literature review on bad smells–5 w's: Which, when, what, who, where**. IEEE Transactions on Software Engineering, 47(1):17–66, 2021.

[22] MOHA, N.; GUEHENEUC, Y.-G.; DUCHIEN, L. ; LE MEUR, A.-F.. **Decor: A method for the specification and detection of code and design smells**. IEEE Transactions on Software Engineering, 36(1):20–36, 2010.

[23] TSANTALIS, N.; CHAIKALIS, T. ; CHATZIGEORGIOU, A.. **Jdeodorant: Identification and removal of type-checking bad smells**. In: 2008 12TH EUROPEAN CONFERENCE ON SOFTWARE MAINTENANCE AND REENGINEERING, p. 329–331, 2008.

[24] MARINESCU, C.; MARINESCU, R.; MIHANCEA, P.; RATIU, D. ; WETTEL, R.. **iplasma: An integrated platform for quality assessment of object-oriented design.** p. 77–80, 01 2005.

[25] PMD. **PMD Source Code Analyzer**, 2025. Available in `https://github.com/pmd/pmd/tree/main`. Accessed March 12, 2025.

[26] **Checkstyle**. `https://checkstyle.sourceforge.io/checks.html`, 2025. Accessed: March 16, 2025.

[27] **Sonarqube**. `https://www.sonarqube.org/`, 2025. Accessed: March 16, 2025.

[28] ARCELLI FONTANA, F.; MÄNTYLÄ, M.; ZANONI, M. ; OTHERS. **Comparing and experimenting machine learning techniques for code smell detection**. Empirical Software Engineering, 21:1143–1191, 2016.

[29] RAYNER, K.. **Eye movements in reading and information processing: 20 years of research**. Psychological Bulletin, 124(3):372–422, 1998.

[30] CROSBY, M.; SCHOLTZ, J. ; WIEDENBECK, S.. **The roles beacons play in comprehension for novice and expert programmers**. 07 2002.

[31] JUST, M. A.; CARPENTER, P. A.. **A theory of reading: From eye fixations to comprehension**. Psychological Review, 87(4):329–354, 1980.

[32] SHARAFI, Z.; SHARIF, B.; GUÉHÉNEUC, Y.-G.; BEGEL, A.; BEDNARIK, R. ; CROSBY, M.. **A practical guide on conducting eye tracking studies in software engineering**. Empirical Softw. Engg., 25(5):3128–3174, sep 2020.

[33] PAUSZEK, J. R.. **An introduction to eye tracking in human factors healthcare research and medical device testing**. Human Factors in Healthcare, 3:100031, 2023.

[34] FEITELSON, D. G.. **From code complexity metrics to program comprehension**. Commun. ACM, 66(5):52–61, Apr. 2023.

[35] POLITOWSKI, C.; KHOMH, F.; ROMANO, S.; SCANNIELLO, G.; PETRILLO, F.; GUÉHÉNEUC, Y.-G. ; MAIGA, A.. **A large scale empirical study of the impact of spaghetti code and blob anti-patterns on program comprehension**. Information and Software Technology, 122:106278, 2020.

[36] PINTO, G.; DE SOUZA, A.. **Cognitive driven development helps software teams to keep code units under the limit!** J. Syst. Softw., 206(C), Dec. 2023.

[37] MÜLLER, S. C.; FRITZ, T.. **Stuck and frustrated or in flow and happy: sensing developers' emotions and progress**. In: PROCEEDINGS OF THE 37TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING - VOLUME 1, ICSE '15, p. 688–699. IEEE Press, 2015.

[38] LI, W.-C.; HORN, A.; SUN, Z.; ZHANG, J. ; BRAITHWAITE, G.. **Augmented visualization cues on primary flight display facilitating pilot's monitoring performance**. International Journal of Human-Computer Studies, 135:102377, 2020.

[39] ABBAD-ANDALOUSSI, A.; SORG, T. ; WEBER, B.. **Estimating developers' cognitive load at a fine-grained level using eye-tracking measures**. In: 2022 IEEE/ACM 30TH INTERNATIONAL CONFERENCE ON PROGRAM COMPREHENSION (ICPC), p. 111–121, 2022.

[40] ALBAGHLI, R.; BEIDAS, A. ; ATTAR, N.. **Eyes on higher education: Evaluating web usability in kuwaiti private universities using eye-tracking and supr-q metrics**. Journal of Engineering Research, 2024.

[41] MERINO, L.; GHAFARI, M.; ANSLOW, C. ; NIERSTRASZ, O.. **A systematic literature review of software visualization evaluation**. Journal of Systems and Software, 144:165–180, 2018.

[42] BEHLER, J.; WESTON, P.; GUARNERA, D. T.; SHARIF, B. ; MALETIC, J. I.. **itrace-toolkit: A pipeline for analyzing eye-tracking data of software engineering studies**. In: 2023 IEEE/ACM 45TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING: COMPANION PROCEEDINGS (ICSE-COMPANION), p. 46–50, May 2023.

[43] BIBIANO, A. C.; UCHÔA, A.; ASSUNÇÃO, W. K.; TENÓRIO, D.; COLANZI, T. E.; VERGILIO, S. R. ; GARCIA, A.. **Composite refactoring: Representations, characteristics and effects on software projects**. Information and Software Technology, 156:107134, 2023.

[44] PAIXÃO, M.; UCHÔA, A.; BIBIANO, A. C.; OLIVEIRA, D.; GARCIA, A.; KRINKE, J. ; ARVONIO, E.. **Behind the intents: An in-depth empirical study on software refactoring in modern code review**. In: PROCEEDINGS OF THE 17TH INTERNATIONAL CONFERENCE ON MINING SOFTWARE REPOSITORIES, MSR '20, p. 125–136, New York, NY, USA, 2020. Association for Computing Machinery.

[45] CEDRIM, D.; GARCIA, A.; MONGIOVI, M.; GHEYI, R.; SOUSA, L.; DE MELLO, R.; FONSECA, B.; RIBEIRO, M. ; CHÁVEZ, A.. **Understanding the impact of refactoring on smells: a longitudinal study of 23 software projects**. In: PROCEEDINGS OF THE 2017 11TH JOINT MEETING ON FOUNDATIONS OF SOFTWARE ENGINEERING, ESEC/FSE 2017, p. 465–475, New York, NY, USA, 2017. Association for Computing Machinery.

[46] ARCELLI FONTANA, F.; ZANONI, M.. **Code smell severity classification using machine learning techniques**. Knowledge-Based Systems, 128:43–58, 2017. Università degli Studi di Milano-Bicocca, Milan, Italy.

[47] PARKER, C.; SCOTT, S. ; GEDDES, A.. **Snowball Sampling**. SAGE Publications, Inc., London, 2019. Accessed on January 16, 2024.

[48] MARTINS, V.; RAMOS, P. L. V.; NEVES, B. B.; LIMA, M. V.; ARRIEL, J.; GODINHO, J. V.; RIBEIRO, J.; GARCIA, A. ; PEREIRA, J. A.. **Eyes on code smells: Analyzing developers' responses during code snippet analysis**. `https://github.com/aisepucrio/EoCS`, 2024. Accessed: 2024-07-25.

[49] BEDNARIK, R.. **Expertise-dependent visual attention strategies develop over time during debugging with multiple code representations**. International Journal of Human-Computer Studies, 70(2):143–155, 2012.

[50] SHARIF, B.; FALCONE, M. ; MALETIC, J.. **An eye-tracking study on the role of scan time in finding source code defects**. In: PROCEEDINGS OF THE SYMPOSIUM ON EYE TRACKING RESEARCH & APPLICATIONS, ETRA'12, p. 381–384, New York, 2012. ACM.

[51] BINKLEY, D.; DAVIS, M.; LAWRIE, D.; MALETIC, J.; MORRELL, C. ; SHARIF, B.. **The impact of identifier style on effort and comprehension**. Empirical Software Engineering, 18(2):219–276, 2013.

[52] **Tobii tx300 eye tracker**. `https://www.spectratech.gr/Web/Tobii/pdf/TX300.pdf`. Acessado em: 17 de janeiro de 2024.

[53] MARTINS, V.; RAMOS, P.; NEVES, B.; LIMA, M.; ARRIEL, J.; GODINHO, J.; RIBEIRO, J.; GARCIA, A. ; PEREIRA, J.. **Eyes on code smells: Analyzing developers' responses during code snippet analysis**. In: ANAIS DO XXXVIII SIMPóSIO BRASILEIRO DE ENGENHARIA DE SOFTWARE, p. 302–312, Porto Alegre, RS, Brasil, 2024. SBC.

[54] LACERDA, G.; PETRILLO, F.; PIMENTA, M. ; GUÉHÉNEUC, Y. G.. **Code smells and refactoring: A tertiary systematic review of challenges and observations**. Journal of Systems and Software, 167:110610, 2020.

[55] VAN SOLINGEN, R.; BERGHOUT, E.. **The goal/question/metric method: a practical guide for quality improvement of software development**. 1999.

[56] ABID, N. J.; MALETIC, J. I. ; SHARIF, B.. **Using developer eye movements to externalize the mental model used in code summarization tasks**. In: PROCEEDINGS OF THE 11TH ACM SYMPOSIUM ON EYE TRACKING RESEARCH & APPLICATIONS, ETRA '19, New York, NY, USA, 2019. Association for Computing Machinery.

[57] CASS, S.. **The top programming languages 2024**. IEEE Spectrum, Aug. 2024. Accessed: 2025-01-07.

[58] ZAKERI-NASRABADI, M.; PARSA, S.; ESMAILI, E. ; PALOMBA, F.. **A systematic literature review on the code smells datasets and validation mechanisms**. ACM Comput. Surv., 55(13s), July 2023.

[59] SANTANA, A.; FIGUEIREDO, E. ; PEREIRA, J. A.. **Unraveling the impact of code smell agglomerations on code stability**. In: INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE AND EVOLUTION (ICSME), p. 461–473, 2024.

[60] SEO, S.. **A review and comparison of methods for detecting outliers in univariate data sets**. Master's thesis, University of Pittsburgh, August 2006.

[61] BIXLER, R.; D'MELLO, S.. **Automatic gaze-based user-independent detection of mind wandering during computerized reading**. User Modeling and User-Adapted Interaction, 26:33–68, 2016.

[62] VON MAYRHAUSER, A.; VANS, A. M.. **Program comprehension during software maintenance and evolution**. Computer, 28(8):44–55, Aug. 1995.

[63] RODEGHERO, P.; MCMILLAN, C.. **An empirical study on the patterns of eye movement during summarization tasks**. In: PROCEEDINGS OF THE 2015 ACM/IEEE INTERNATIONAL SYMPOSIUM ON EMPIRICAL SOFTWARE ENGINEERING AND MEASUREMENT (ESEM), volumen 00, p. 1–10. ACM/IEEE, 2015.

[64] CEDRIM, D.; SOUSA, L.. **Organic**, 2017. Available in `https://github.com/diegocedrim/organic`. Accessed February 5, 2025.

[65] SHARMA, T.; MISHRA, P. ; TIWARI, R.. **Designite: A software design quality assessment tool**. In: INTERNATIONAL WORKSHOP ON BRINGING ARCHITECTURAL DESIGN THINKING INTO DEVELOPERS' DAILY ACTIVITIES, p. 1–4, 2016.

[4] CORBIN, J.; STRAUSS, A.. **Basics of qualitative research: Techniques and procedures for developing grounded theory**. Thousand Oaks, 3:1–400, 2008.