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

Vinícius Martins, Pedro Lopes Verardo Ramos, Breno Braga Neves, Maria Vitoria Lima, Johny Arriel, João Victor Godinho, Joanne Ribeiro, Alessandro Garcia, Juliana Alves Pereira
Pontifical Catholic University of Rio de Janeiro (PUC-Rio), Rio de Janeiro, Brazil

ABSTRACT

Detection and refactoring of smelly code are crucial activities, along with software maintenance and evolution. Code smells are indicators of poor design and implementation choices, which are expected to affect the developers' program comprehension. The use of eye-tracking technology provides an interesting means of analyzing the impact of code smells on program comprehension. However, there is limited existing effort in this direction. This paper reports a study in which we have used an eye tracker to investigate how the presence of smells influences developers' program comprehension. We observed that the smell data class leads to a lower cognitive effort, while long methods and feature envies imposed a considerably higher cognitive effort. That explains why refactoring of feature envies and long methods smells has been much more common across the projects. We also have complementary eye-tracking indicators to reveal other smell aspects harming program comprehension. These findings reinforce the importance of enhancing IDE features to reduce the developer's burden when engaging in cognitive processes of largely coupled code.

KEYWORDS

eye tracker, code smell, software quality, program comprehension

ACM Reference Format:
Vinícius Martins, Pedro Lopes Verardo Ramos, Breno Braga Neves, Maria Vitoria Lima, Johny Arriel,, João Victor Godinho, Joanne Ribeiro, Alessandro Garcia, Juliana Alves Pereira. 2024. Eyes on Code Smells: Analyzing Developers' Responses During Code Snippet Analysis. In *Proceedings of Brazilian Symposium on Software Engineering (SBES'24).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

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 [14]. Thus, addressing code smells is important for companies to avoid rework, increased costs, and decreased productivity in software development projects [12]. This highlights the need to thoroughly understand the impact of code smells on program comprehension.

SBES'24, September 30 – October 04, 2024, Curitiba, PR
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

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 [25]. 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 [32].

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 [32]. Sharafi et al. [32] 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, there is currently 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 employed 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 to understand how the presence of code smells influenced developers' program analysis.

To capture and analyze data, we used the iTrace Eclipse Plugin [29], iTrace-Toolkit [6] 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 for improving the development of more effective tools and practices for 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

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

studies have reported that the refactoring of *feature envies* and *long methods* has been much more common across projects [7, 10, 21, 22].

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 subjects 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 source code. By integrating eye-tracking technology into development environments, tools can identify the cognitive patterns associated with developers and 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 by inspiring further research and the development of more effective tools for identifying and refactoring code smells.

2 BACKGROUND

In this section, we introduce the basic concepts of code smells and motivate the use of the eye tracker in the field. Furthermore, we present related work.

2.1 Code Smells

The term *"code smell"* was used by Riel [28] and Beck et al. [9], to highlight symptoms that can lead to poor design and result in problems in maintaining the software. Soon after, Fowler et al. [14] defined code smells formally, and Mäntylä et al. [19] came up with another definition for code smells as a term that refers to a somewhat subjective indicator of poor design or coding style in specific parts of the code. In this work, we focus on three specific types of code smells: *long method*, *feature envy*, and *data class*. According to Mäntylä et al. [18] the following are the definitions for these three types of code smells:

- *Long method:* A method that is too long, making it difficult to understand, change, or extend.
- *Data class:* A class that primarily serves as a container for data fields, but does not encapsulate any additional logic.
- *Feature envy:* A method that shows more interest in other class(es) than the one it is currently located.

We selected these three smell types as (1) they are quite different from each other in terms of structural problems they represent, (2) these different structural problems may stimulate the experiment participant in a wide of different ways, (3) choosing more smell types would make our experiment too complex, and (4) complex experiments tend to make the subjects feel very tired or stressed, which would unavoidably interfere in the results. In future research, studies can replicate our experiments using other smell types.

2.2 Eye Tracker

Eye-tracking technology provides a comprehensive understanding of how individuals interact with visual elements. By collecting data on eye movements, this technology reveals how people navigate reading material [27] and respond to visual prompts [11]. It is valuable for understanding the cognitive processes involved in comprehension and problem-solving. Since cognitive functions guide gaze direction, analyzing eye movements offers valuable insights into the cognitive efforts employed during various software engineering activities. Thus, eye tracking is a powerful tool for studying cognitive interactions with different stimuli, as eye gaze patterns can reveal much about an individual's thought processes.

The relation between eye gaze and cognitive processing is grounded in two key assumptions: the immediacy assumption and the eye-mind assumption [16]. The immediacy assumption suggests that the interpretation of stimuli begins as soon as they are seen. The eye-mind assumption posits that individuals focus their attention only on the part of the stimulus currently being processed [16]. These assumptions form the foundation for understanding how eye gaze reflects cognitive processes. Thus, eye gaze data indicate both the target of an individual's attention and the effort and duration spent understanding the stimulus.

In the context of software development, an eye tracker becomes a powerful tool for assessing a developer's attention and cognitive processes. By capturing precise eye movements and gaze patterns, researchers and developers can gain insights into how individuals interact with visual stimuli on a screen. Analyzing which elements attract the most attention helps developers understand the visual hierarchy within a codebase, enabling them to prioritize essential components and optimize code readability.

When working with an eye tracker, several metrics can be analyzed, including saccades, pupil diameter, constriction, areas of interest, and fixation [25]. For this study, we will focus on developers' fixations. Fixations indicate areas of the stimulus where visual attention is concentrated, leading to cognitive processing. Most fixations last between 100 and 600 milliseconds, but this can vary significantly depending on context and other relevant factors [25].

3 RELATED WORK

The application of eye trackers in Software Engineering studies is varied, *e.g.* for program comprehension and code review.

In program comprehension, there is a study conducted by Peitek et al. [26] that focuses on the relationship between programming experience and programming efficacy, encompassing both code comprehension correctness and speed. The experiment combined the use of electroencephalography (EEG) and eye-tracking technologies to observe 38 participants as they comprehended Java code snippets. Their finding was that developers who displayed higher efficacy tended to read the source code in a more targeted manner and experienced a lower cognitive load.

In code review, Alcocer et al. [2] employed an eye tracker to monitor the interaction of software developers with unified views in GitHub for bug detection. The findings suggest that this view may facilitate more efficient code analysis and potentially lead to the discovery of more bugs. The study also observed that participants primarily focused on conditionals, class/instance variables, and changes in the code. Huang et al. [15] conducted a study to understand the differences in how men and women conduct code reviews. With the support of eye-tracking technology, they examined the differences in code review practices between genders. Their findings reveal distinct differences in how men and women perform code reviews highlighting the need for specialized tools.

Several studies [13] use visualization techniques such as heatmaps and scan paths to investigate the behavior of developers when they are coding a software system. Other works are directly related to this research, such as those by Shaffer et al. [29] who developed the iTrace tool used in the study, capable of capturing the movement of developers' eyes while performing tasks in different development environments. In conjunction with iTrace, Behler et al. [6] developed a complementary tool called iTrace-Toolkit, which processes the raw data, maps, and generates fixes, saving the data in a database which helps and facilitates data analysis.

Although previous studies have explored various aspects of software development and code comprehension, our work specifically investigates how developers analyze code snippets with and without code smells. By understanding the cognitive load imposed by code smells, we can better support developers in their efforts to write cleaner, more maintainable code.

4 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 steps (see Figure 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.

4.1 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. The eye tracker was calibrated before the second phase of the experiment with the participant (see Section 4.6). 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.

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.

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

There are several code smells datasets available, such as the one published by Palomba et al. [23], 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 [3] and severity scale [4]. 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 [17], which was manually labeled with the support of 26 developers from industry who participated in reviewing the code snippets. All developers are involved in activities related to

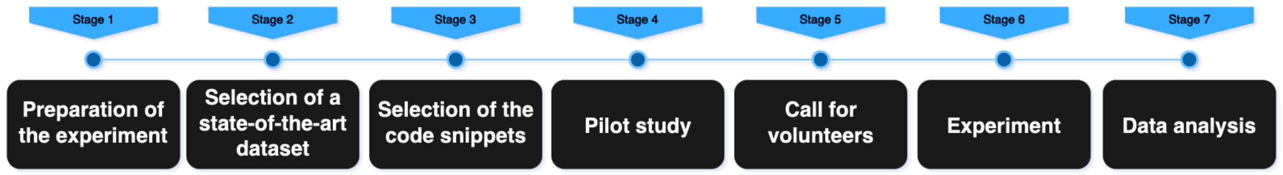

Figure 1: Overview of our research methodology.

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 [17] and access the dataset at https://zenodo.org/records/3666840.

4.3 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) Professional experience in software development greater than 3 years, (IC2) Professional experience in the software industry greater than 3 years, and (IC3) Professional 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, and (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 4.4).

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

4.4 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. [31]: *(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.

4.5 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 [24], asking each participant to refer our survey to colleagues with similar experiences and interest in joining.

4.6 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 collected. All responses were supplemented by audio and video recordings. The survey is available in our supplementary material [20].

4.6.1 *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.

4.6.2 *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, he 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 [20] 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.

4.6.3 *Third Phase: Collecting participant background data.*

In the third phase, the participant responded to a subset of questions derived from the original MLCQ article. 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, to understand their development experience. 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. Additionally, we solicited the participants' opinions on the utility of the data gathered by the eye tracker in understanding their assessments.

4.7 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 [29]. 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[6], 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 [31], which indicates where developers are focusing their attention in the code and is derived from time. We chose the I-VT algorithm [6], to calculate fixation duration, recommended for eye trackers with a refresh rate higher than 200Hz aligning with our equipment (see Section 4.8).

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 [30].

4.7.1 *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 RQ 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 [5, 8, 33]. 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 question aims to discover the regions of the code that receive the most attention from developers. Understanding these Areas of Interest (AOIs) 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 [6]. 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 question aims to explore the relationship between developers' fixation patterns and their ability to correctly identify code smells. The goal is to determine whether there is a significant difference in fixation patterns between developers who successfully identified code smells and those who did not. First, we verified the right and wrong answers. Then, we compared the fixation count between these answers. 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.

4.8 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 [1]. 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 [20].

5 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.

5.1 Participants Contextualization

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

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

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 databases, it was checked whether the data collected via eye tracker were consistent, analyzing the amount of data generated within a database. 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.

Figure 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 shows that more than 50% of the participants have at least a bachelor's degree in Science or Engineering.

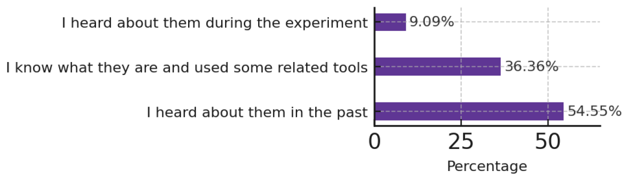

Figure 2: Participants' familiarity with the concept of smells

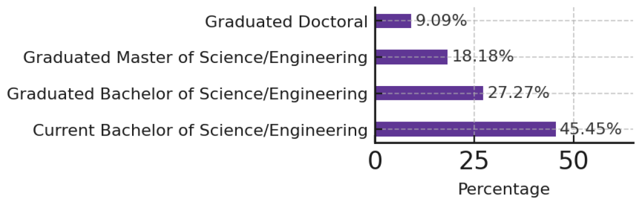

Figure 3: Participants' degree

5.2 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 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. It is also observed that the third quartile is lower than the second quartile of the

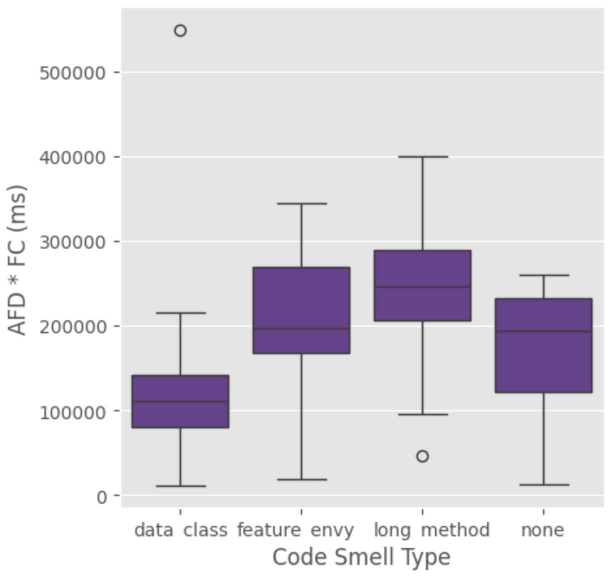

Figure 4: Boxplot of AFD * FC per Smell Type.

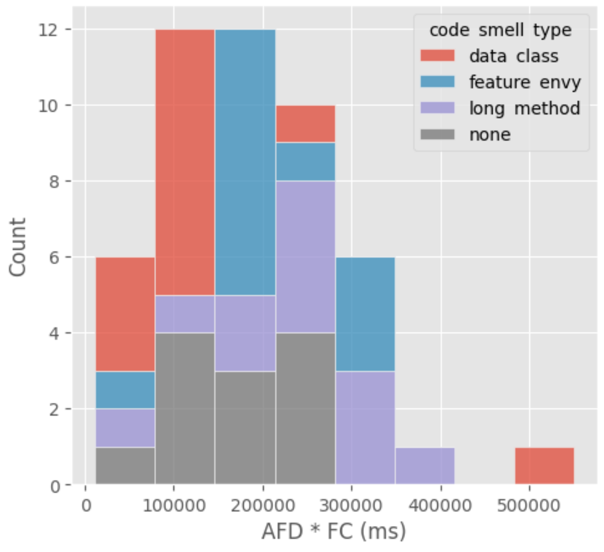

Figure 5: Stacked bar of AFD * FC per Smell Type.

other code smells present in the study. 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 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 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.

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 [20]. 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 *long method* with higher perceived complexity, participants' explanations were more varied and less consistent.

5.3 RQ2: Most Examined Code Sections

To answer this research question, we need to understand the granularity 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 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 7, shows the syntactic categories that appeared the most in the 10 arrowed lists with the most 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,

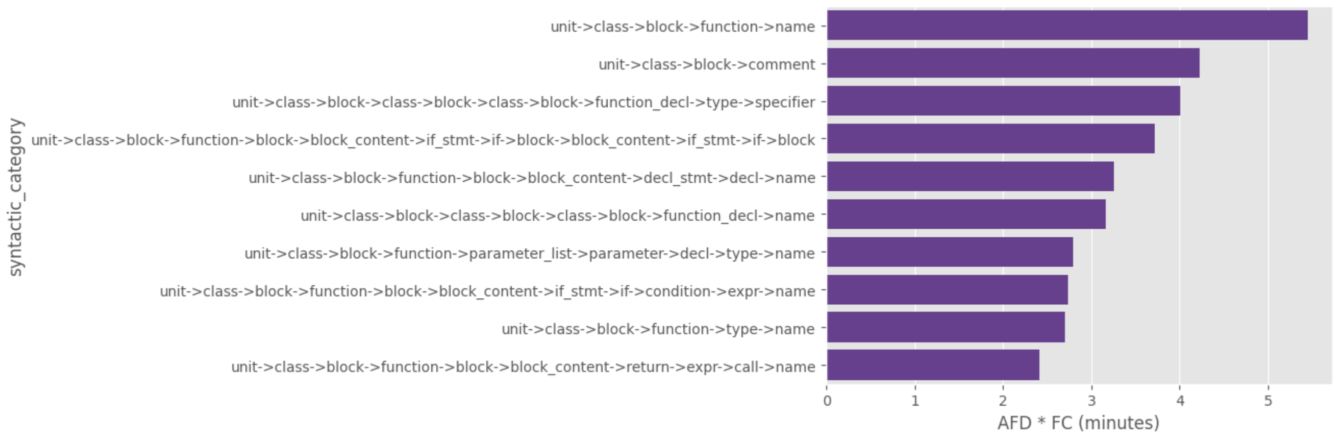

Figure 6: TOP 10 arrowed list for the hightest AFD*FC (minutes) for all developers.

likely due to their complexity and potential impact on program execution.

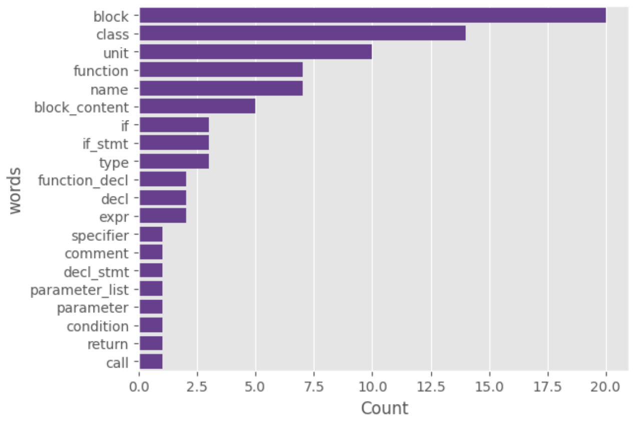

Figure 7: Count of the most frequent syntactic categories in the 10 arrowed list.

5.4 RQ3: Fixation Patterns

When analyzing Table 1, we noticed that there is no huge difference between right and wrong answers regarding the identification or not of code smells and their type.

Table 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

Figure 8 distinguishes the stacked bar of normalized FC count for right (purple) and wrong (blue) answers for all code snippets.

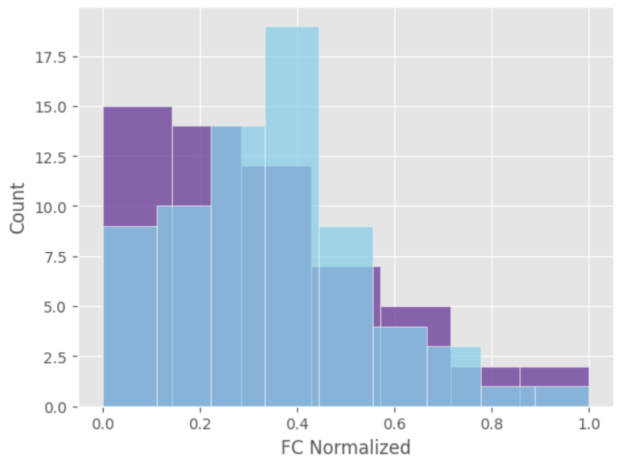

Figure 8: Stacked bar of normalized FC count for right (purple) and wrong (blue) answers for all code snippets.

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 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 9 shows the bloxplots for the right answers and Figure 10 shows 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

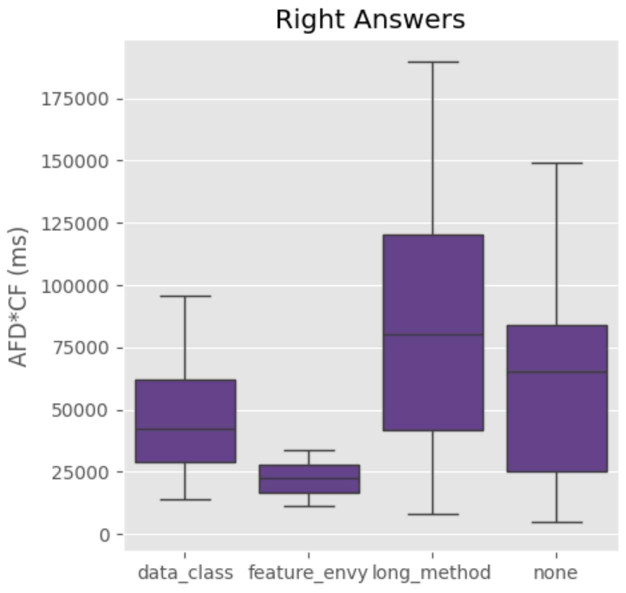

Figure 9: Boxplot of AFD * FC by smell type for right answers.

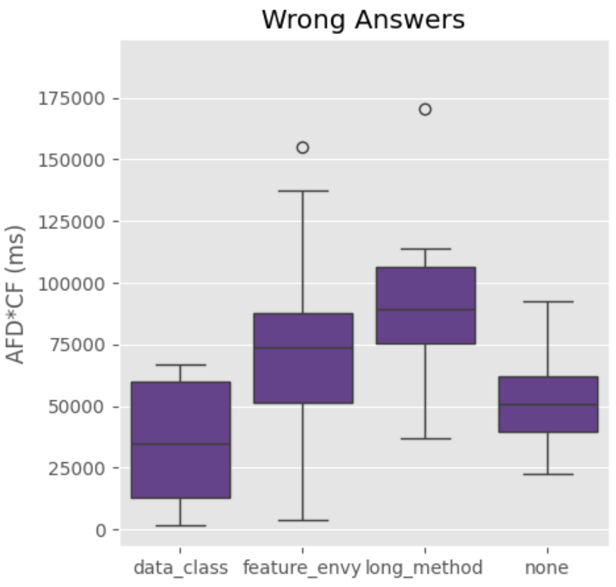

Figure 10: Boxplot of AFD * FC by smell type for wrong answers.

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 got it right demonstrate greater efficiency and consistency in identifying this specific type of smell when compared to those who got it wrong. This may be linked to the fact that this code smell is more related to other classes. If the developer fails to notice it initially, may indicate that the method is not related to another class and continued to try to identify some other type of code smell.

For the *long method*, the result demonstrated that a more detailed analysis presented better results in correctly identifying the smell, which is understandable considering that this smell is characterized by being a code smell that is very long.

6 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.

7 THREATS TO VALIDITY

In the present study, there are threats to validity. Threats to internal validity include individual participant variations and the length of the experiment. The different programming skills, experience, and individual physiological conditions of developers can influence their responses to biosensors. Measures such as having knowledge of the Java language, a presentation of the concept and the types of code smells before analysis, and the implementation of a protocol so that the participant remains calm during the experiment were carried out with the aim of mitigating such threats. Another point 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.

For external threats, 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 even more 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.

8 CONCLUSION

This study explores the physiological responses of developers when analyzing code snippets with and without the presence of code smells using eye tracker. By analyzing the fixations generated by eye tracking, it was possible to identify specific categories of developer focus and their responses during the analysis help to better understand these focal points.

This work contributes to the understanding of how structural problems in code can affect developers, indicating the path towards the development of more effective tools and techniques in identifying and treating code smells and, as a consequence, bringing a better quality of life to developers. 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.

ACKNOWLEDGMENTS

This research was partially funded by the State Institute of Engineering and Architecture (IEEA), linked to the Infrastructure Secretariat of the State of Rio de Janeiro, through Contract 001/2021 with funding from the Government of the State of Rio de Janeiro.

REFERENCES

[1] [n. d.]. Tobii TX300 Eye Tracker. https://www.spectratech.gr/Web/Tobii/pdf/TX300.pdf. https://www.spectratech.gr/Web/Tobii/pdf/TX300.pdf Accessed: January 17, 2024.
[2] Juan Pablo Sandoval Alcocer, Alejandra Cossio-Chavalier, Tiara Rojas-Stambuk, and Leonel Merino. 2023. An Eye-Tracking Study on the Use of Split/Unified Code Change Views for Bug Detection. *IEEE Access* 11 (2023), 136195–136205. https://doi.org/10.1109/ACCESS.2023.3336859
[3] F. Arcelli Fontana, M.V. Mäntylä, M. Zanoni, et al. 2016. Comparing and experimenting machine learning techniques for code smell detection. *Empirical Software Engineering* 21 (2016), 1143–1191. https://doi.org/10.1007/s10664-015-9378-4
[4] Francesca Arcelli Fontana and Marco Zanoni. 2017. Code smell severity classification using machine learning techniques. *Knowledge-Based Systems* 128 (2017), 43–58. Università degli Studi di Milano-Bicocca, Milan, Italy.
[5] R Bednarik. 2012. Expertise-dependent visual attention strategies develop over time during debugging with multiple code representations. *International Journal of Human-Computer Studies* 70, 2 (2012), 143–155. https://doi.org/10.1016/j.ijhcs.2011.09.003
[6] Joshua Behler, Praxis Weston, Drew T. Guarnera, Bonita Sharif, and Jonathan I. Maletic. 2023. 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)*. 46–50. https://doi.org/10.1109/ICSE-Companion58688.2023.00022
[7] Ana Carla Bibiano, Anderson Uchôa, Wesley K.G. Assunção, Daniel Tenório, Thelma E. Colanzi, Silvia Regina Vergilio, and Alessandro Garcia. 2023. Composite refactoring: Representations, characteristics and effects on software projects. *Information and Software Technology* 156 (2023), 107134. https://doi.org/10.1016/j.infsof.2022.107134
[8] D Binkley, M Davis, D Lawrie, JI Maletic, C Morrell, and B Sharif. 2013. The impact of identifier style on effort and comprehension. *Empirical Software Engineering* 18, 2 (2013), 219–276. https://doi.org/10.1007/s10664-012-9201-4
[9] William J. Brown. 1998. *AntiPatterns: refactoring software architectures and projects in crisis.* Wiley, New York.
[10] Diego Cedrim, Alessandro Garcia, Melina Mongiovi, Rohit Gheyi, Leonardo Sousa, Rafael de Mello, Baldoino Fonseca, Márcio Ribeiro, and Alexander Chávez. 2017. 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* (Paderborn, Germany) *(ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 465–475. https://doi.org/10.1145/3106237.3106259
[11] Martha Crosby, Jean Scholtz, and Susan Wiedenbeck. 2002. The roles beacons play in comprehension for novice and expert programmers. (07 2002).
[12] Dipta Das, Abdullah Al Maruf, Rofiqul Islam, Noah Lambaria, Samuel Kim, Amr S. Abdelfattah, Tomas Cerny, Karel Frajtak, Miroslav Bures, and Pavel Tisnovsky. 2022. Technical debt resulting from architectural degradation and code smells: a systematic mapping study. *SIGAPP Appl. Comput. Rev.* 21, 4 (jan 2022), 20–36. https://doi.org/10.1145/3512753.3512755
[13] Daniel Kyle Davis and Feng Zhu. 2022. Analysis of software developers' coding behavior: A survey of visualization analysis techniques using eye trackers. *Computers in Human Behavior Reports* 7 (2022), 100213.
[14] Martin Fowler and Kent Beck. 1999. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley Longman Publishing Co., Inc., USA.
[15] Yu Huang, Kevin Leach, Zohreh Sharafi, Nicholas McKay, Tyler Santander, and Westley Weimer. 2020. Biases and differences in code review using medical imaging and eye-tracking: genders, humans, and machines. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 456–468. https://doi.org/10.1145/3368089.3409681
[16] Marcel A. Just and Patricia A. Carpenter. 1980. A theory of reading: From eye fixations to comprehension. *Psychological Review* 87, 4 (1980), 329–354. https://doi.org/10.1037/0033-295X.87.4.329
[17] Lech Madeyski and Tomasz Lewowski. 2020. MLCQ: Industry-Relevant Code Smell Data Set. In *Proceedings of the Evaluation and Assessment in Software Engineering (EASE '20)*. Association for Computing Machinery, New York, NY, USA, 342–347. https://doi.org/10.1145/3383219.3383264
[18] Mika Mäntylä. 2003. *Bad Smells in Software - A Taxonomy and an Empirical Study.* Ph. D. Dissertation. Helsinki University of Technology.
[19] Mika V. Mäntylä and Casper Lassenius. 2006. Subjective evaluation of software evolvability using code smells: An empirical study. *Empirical Software Engineering* 11 (2006), 395–431. https://doi.org/10.1007/s10664-006-9002-8
[20] Vinícius Martins, Pedro Lopes Verardo Ramos, Breno Braga Neves, Maria Vitoria Lima, Johny Arriel, João Victor Godinho, Joanne Ribeiro, Alessandro Garcia, and Juliana Alves Pereira. 2024. Eyes on Code Smells: Analyzing Developers' Responses During Code Snippet Analysis. https://github.com/aisepucrio/EoCS. Accessed: 2024-07-25.
[21] Daniel Oliveira, Wesley K. G. Assunção, Alessandro Garcia, Ana Carla Bibiano, Márcio Ribeiro, Rohit Gheyi, and Baldoino Fonseca. 2023. The untold story of code refactoring customizations in practice. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 108–120. https://doi.org/10.1109/ICSE48619.2023.00021
[22] Matheus Paixão, Anderson Uchôa, Ana Carla Bibiano, Daniel Oliveira, Alessandro Garcia, Jens Krinke, and Emilio Arvonio. 2020. 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* (Seoul, Republic of Korea) *(MSR '20)*. Association for Computing Machinery, New York, NY, USA, 125–136. https://doi.org/10.1145/3379597.3387475
[23] Fabio Palomba, Dario Di Nucci, Michele Tufano, Gabriele Bavota, Rocco Oliveto, Denys Poshyvanyk, and Andrea De Lucia. 2015. Landfill: An Open Dataset of Code Smells with Public Evaluation. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories.* 482–485. https://doi.org/10.1109/MSR.2015.69
[24] Charlie Parker, Sam Scott, and Alistair Geddes. 2019. *Snowball Sampling.* SAGE Publications, Inc., London. https://doi.org/10.4135/9781526421036831710 Accessed on January 16, 2024.

[25] Joseph R Pauszek. 2023. An introduction to eye tracking in human factors healthcare research and medical device testing. *Human Factors in Healthcare* 3 (2023), 100031.

[26] Norman Peitek, Annabelle Bergum, Maurice Rekrut, Jonas Mucke, Matthias Nadig, Chris Parnin, Janet Siegmund, and Sven Apel. 2022. Correlates of programmer efficacy and their link to experience: a combined EEG and eye-tracking study. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Singapore) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 120–131. https://doi.org/10.1145/3540250.3549084

[27] K. Rayner. 1998. Eye movements in reading and information processing: 20 years of research. *Psychological Bulletin* 124, 3 (1998), 372–422. https://doi.org/10.1037/0033-2909.124.3.372

[28] Arthur J. Riel. 1996. *Object-oriented design heuristics*. Addison-Wesley Pub. Co, Reading, Mass.

[29] Timothy R. Shaffer, Jenna L. Wise, Braden M. Walters, Sebastian C. Müller, Michael Falcone, and Bonita Sharif. 2015. 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* (Bergamo, Italy) *(ESEC/FSE 2015)*. Association for Computing Machinery, New York, NY, USA, 954–957. https://doi.org/10.1145/2786805.2803188

[30] Z Sharafi, T Shaffer, S Bonita, and YG Guéhéneuc. 2015. Eye-tracking metrics in software engineering. In *Proceedings of the 22nd Asia-Pacific Software Engineering Conference (APSEC '15)*. IEEE CS Press.

[31] Zohreh Sharafi, Bonita Sharif, Yann-Gaël Guéhéneuc, Andrew Begel, Roman Bednarik, and Martha Crosby. 2020. A Practical Guide on Conducting Eye Tracking Studies in Software Engineering. *Empirical Softw. Engg.* 25, 5 (sep 2020), 3128–3174. https://doi.org/10.1007/s10664-020-09829-4

[32] Zohreh Sharafi, Zéphyrin Soh, and Yann-Gaël Guéhéneuc. 2015. A systematic literature review on the usage of eye-tracking in software engineering. *Information and Software Technology* 67 (2015), 79–107. https://doi.org/10.1016/j.infsof.2015.06.008

[33] B Sharif, M Falcone, and JI Maletic. 2012. 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)*. ACM, New York, 381–384.

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009