Evaluating the Code Comprehension of Novices with Eye Tracking

José Aldo Silva da Costa[1], **Rohit Gheyi**[1]

[1]Universidade Federal de Campina Grande (UFCG)
R. Aprígio Veloso, 882 – 58.109-970 – Campina Grande – PB – Brasil

josealdo@copin.ufcg.edu.br, rohit@dsc.ufcg.edu.br

Abstract. *Code comprehension is essential for software evolution, but it can be hindered by structural problems in the code. Despite its importance, researchers often rely on static code metrics to investigate code comprehension, without considering dynamic aspects such as the visual effort involved. Therefore, we aim to fill this gap by examining code comprehension using eye-tracking metrics. We conducted three controlled eye-tracking studies with 128 novices in three distinct programming languages, C, Python, and Java using constructions with different granularity levels, namely atoms of confusion, refactorings, and* `#ifdef` *annotations. With respect to atoms of confusion, we found that the clarified version of the code reduced the time and number of attempts by 38.6% and 28%, respectively. The use of the Extract Method refactoring significantly reduced the time of two tasks by 70% to 78.8%, while increasing the accuracy of three tasks by 20% to 34.4%. The use of* `#ifdef` *annotations presented reductions in the modified region by 46.9% in time, 44.7% in the fixation duration, 48.4% in the fixation count, and 60.5% in the regressions count. In addition, we also identified patterns in our data by observing several visual transitions going forward and backward between specific code elements which allowed us to infer confusion in the code. These results provide educators, practitioners, and researchers with valuable information on certain transformations and their potential to ease or hinder code comprehension. The study also highlights the potential of visual metrics to reveal the impact of transformations that cannot be captured by static code metrics. Our results also pave the way for the development of more advanced tools that track the eye movements of developers and assist them with helpful suggestions.*

Resumo. *A compreensão de código é essencial para a evolução do software, mas pode ser prejudicada por problemas estruturais no código. Apesar de sua importância, pesquisadores muitas vezes dependem de métricas de código estático para investigar a compreensão do código, sem considerar aspectos dinâmicos, como o esforço visual envolvido. Portanto, nosso objetivo é preencher essa lacuna examinando a compreensão do código usando métricas de rastreamento ocular. Realizamos três estudos controlados de rastreamento ocular com 128 novatos em três linguagens de programação distintas, C, Python e Java, utilizando construções com diferentes níveis de granularidade, ou seja, átomos de confusão, refatorações e anotações* `#ifdef`*. Em relação aos átomos de confusão, descobrimos que a versão esclarecida do código reduziu o tempo e o número de tentativas em 38,6% e 28%, respectivamente. O uso da refatoração*

Extract Method reduziu significativamente o tempo de duas tarefas em 70% a 78,8%, enquanto aumentou a precisão de três tarefas em 20% a 34,4%. O uso de anotações `#ifdef` *apresentou reduções na região modificada de 46,9% no tempo, 44,7% na duração da fixação, 48,4% no número de fixações e 60,5% no número de regressões. Além disso, também identificamos padrões em nossos dados observando várias transições visuais avançando e retrocedendo entre elementos de código específicos, o que nos permitiu inferir confusão no código. Esses resultados fornecem informações valiosas para educadores, profissionais e pesquisadores sobre determinadas transformações e seu potencial para facilitar ou dificultar a compreensão do código. O estudo também destaca o potencial de métricas visuais para revelar o impacto de transformações que não podem ser capturadas por métricas de código estático. Nossos resultados também abrem caminho para o desenvolvimento de ferramentas mais avançadas que rastreiam os movimentos oculares dos desenvolvedores e os auxiliam com sugestões úteis.*

1. Introduction

Code comprehension is crucial for software maintenance and evolution processes, but can be hindered by structural problems in the code. To address these structural problems, developers often use behavior-preserving code transformations [Mongiovi et al. 2018, Soares et al. 2013] aiming to make it easier to understand, maintain, and evolve. However, we still lack empirical evidence on how behavior-preserving code changes affect the understanding of novice programmers, especially in terms of visual effort.

Developers often read code written by others to understand its behavior, but tiny patterns in the code can cause misunderstandings, leading to misjudging its behavior. These patterns, known as atoms of confusion [Gopstein et al. 2017], can obfuscate the code and confuse developers. Atoms of confusion are present when functionally equivalent alternatives result in better performance. However, coarse-grained approaches may not suffice in capturing the effect of fine-grained code elements, such as atoms.

Extract and Inline Method are two commonly used refactorings to improve code understanding. However, there is a lack of empirical evidence on the extent of their actual impact. Although Fowler [Fowler 1999] suggests that short methods with descriptive names lead to clearer code, static code metrics reveal an increase in the number of code *smells*–patterns in the code that indicate deeper issues [Cedrim et al. 2017]. We lack a better understanding of what circumstances developers should opt for inlining or extracting a method. To understand it better, we need to evaluate other dimensions such as the dynamic perspective of code comprehension using eye-tracking metrics.

Code with annotations such as `#ifdef` may also affect code comprehension. There are two types of annotations: disciplined and undisciplined [Medeiros et al. 2015, Liebig et al. 2011]. Disciplined annotations enclose only complete statements, while undisciplined annotations may only enclose partial statements, such as only the opening bracket. Both achieve the same goal, but disciplined annotations align with the code's syntactic structure, while undisciplined annotations do not. Although there are a few other studies in the literature, there is no consensus yet on whether undisciplined annotations should be refactored to become disciplined in practice [Malaquias et al. 2017, Schulze et al. 2013]. However, previous studies have limitations, either relying solely on

developers' opinions or on a narrow set of conventional metrics. Fine-grained transformations may not show observable differences using conventional metrics alone. We need additional indicators of developer behavior when comprehending annotated code, which may reveal important insights into the benefits and drawbacks of the annotation discipline.

To better understand the nuances of code comprehension and contribute with a dynamic additional perspective, we conducted three controlled eye-tracking studies with 128 novices in three distinct programming languages, namely C, Python, and Java using constructions with different granularity levels, namely atoms of confusion, methods, and `#ifdef` annotations.

Regarding the atoms of confusion, we found that the clarified version of the code reduced the time and number of attempts by 38.6% and 28%, respectively. The use of the Extract Method refactoring significantly reduced the time of two tasks by 70% to 78.8%, while increasing the accuracy of three tasks by 20% to 34.4%. The use of `#ifdef` annotations presented reductions in the modified region by 46.9% in time, 44.7% in the fixation duration, 48.4% in the fixation count, and 60.5% in the regressions count. In addition, we also identified patterns in our data by observing several visual transitions going forward and backward between specific code elements which allowed us to infer confusion in the code, such as when adding the particle `not` before the variable.

We gained deeper insights by analyzing visual effort, time, and attempts. We investigated: 1) how much time the subjects spent in the refactored code regions, i.e., line of code that contains an atom of confusion, an extraction of a method, a method call, a method inlining, or `#ifdef` annotations; 2) to what extent transformations can impact visual metrics; and 3) how transformations impact the code reading with the visual regressions. The main contributions of this work are:

- A framework for conducting eye tracking controlled experiments, which can be used to evaluate the effectiveness of behavior-preserving code changes and their impact on code comprehension and visual effort;
- A controlled experiment using eye tracking with novices in Python to evaluate the impact of clarifying atoms of confusion on code comprehension [da Costa et al. 2023];
- A controlled experiment using eye tracking with novices in Java to evaluate the impact of the Extract Method on code comprehension;
- A controlled experiment using eye tracking with novices in the C to evaluate three refactorings that discipline `#ifdef` annotations in C programs [da Costa et al. 2021];
- A controlled experiment using eye tracking with novices and practitioners in C and C++ to evaluate the impact of clarifying atoms of confusion on code comprehension [de Oliveira et al. 2020].

For the **researchers**, our results show the potential of visual metrics to reveal an impact of code transformations that cannot be captured by static code metrics. While under the perspective of code metrics, the Extract Method refactoring can have a negative impact according to a previous study, in our study, we found significant reductions in the visual fixation duration in the code when the novices solve the tasks *Number of Digits* and *Next Prime*, reaching 78.9%. **Educators** should avoid using obfuscated code with atoms such as *Multiple Variable Assignment*, *True or False Evaluation*, and *Operator*

Precedence, as these constructs impact novices' abilities to understand the code in Python. **Practitioners and language designers**' community should be more careful when using constructions to simplify the language that could impair the novices' code comprehension.

2. Problem

In this section, we explain the research gaps in three scenarios that deal with code transformations aiming to improve code comprehension. For instance, atoms of confusion are prevalent in open-source projects in C language [Medeiros et al. 2019] but also occur in other programming languages. In Figure 1(a), we illustrate a *Conditional Expression* found in the *SwiftShader* project for Python language and adapted to a complete code snippet. Iterating over a list of elements, in Line 4, `num` receives the value of `elem` if `elem` is equal to three; otherwise, `num` receives one. If the implications of the study of Gopstein et al. [Gopstein et al. 2017] for C language are sustainable for Python as well, the *Conditional Expression* in Line 4 impairs the code understanding because the assignment depends on the value of a variable.

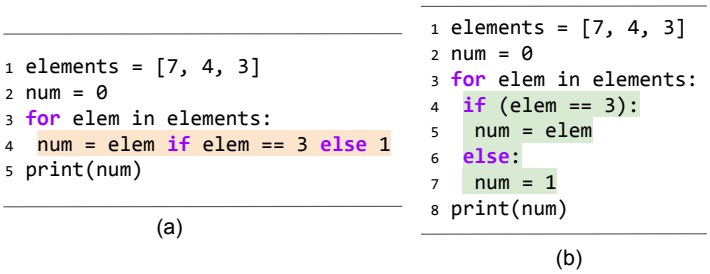


Figura 1. Code adapted from *SwiftShader* with (a) obfuscated code containing the atom *Conditional Expression*, and (b) the clarified version of the code.

Medeiros et al. [Medeiros et al. 2019] proposed a clarified solution that breaks the line containing the atom into four lines of code as presented in Figure 1(b). They also investigated the subjective perceptions of experienced developers regarding the atom in C language, finding that the code with the atom did not negatively influence the developers' understanding. Additionally, developers accepted pull requests with both obfuscated and clarified code versions. We need additional empirical evidence to comprehend better the impact of atoms of confusion on code comprehension. Coarse-grained approaches may not suffice in capturing the effect of fine-grained code elements, such as atoms. Eye tracking has demonstrated potential in assessing the impact of small-grained code changes on comprehension by recording human subjects' eye movements and analyzing their visual attention [da Costa et al. 2021, Sharafi et al. 2015, Busjahn et al. 2011].

We simplified a sequence of fixations performed by two subjects in Figure 2. Each red circle represents a fixation that varies in size according to duration. The sequence and direction of fixations are depicted in chronological order with a number inside. In the obfuscated version (Figure 2(a)), the subject makes eight fixations with six within the line of the atom (Line 4). In the clarified version (Figure 2(b)), the subject makes five fixations, with four of them within the atom region (Lines 4–7). Thus, the subject fixates more times and for a longer time in obfuscated version. In addition, the subject regresses visually in the code more times in obfuscated version. In obfuscated version, she goes

back three times in code, twice vertically examining the list, and one time horizontally to possibly inspect a variable. In the clarified version, the subject goes back only once to the list, making a vertical regression between lines. By examining their behavior at this small-grained level, we can new insights.

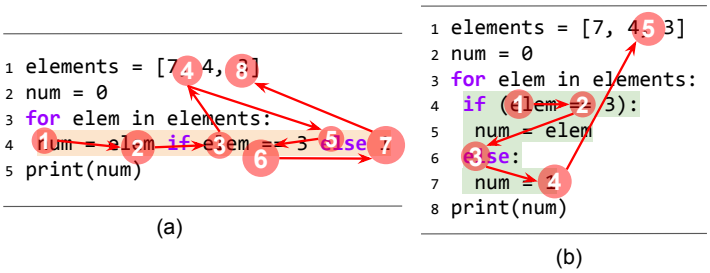


Figura 2. Code with eye gaze patterns for (a) obfuscated code containing the atom *Conditional Expression*, and (b) the clarified version of the code.

We also need more evidence to understand how classical code transformations affect our ability to comprehend the code. Extract and Inline Method are two commonly used refactorings to improve code understanding. However, there is a lack of empirical evidence on the extent of their actual impact. Although Fowler [Fowler 1999] suggests that short methods with descriptive names lead to clearer code, the Extract Method can increase the number of code smells instead [Cedrim et al. 2017]. We lack a better understanding of what circumstances developers should opt for inlining or extracting a method. In another scenario, comprehending code with `#ifdef` directives can be difficult despite the relevance and prevalence of conditional compilation in practice. Code with either disciplined or undisciplined annotations may affect program comprehension. However, empirical knowledge on the influence of the annotation discipline is still scarce.

Overall, there is no consensus on whether developers should use disciplined annotations. Previous studies are either strictly based on developers' opinions or on a limited set of conventional metrics related to code comprehension, such as time and accuracy. There are not always observable differences in applying fine-grained refactorings using conventional metrics, and the use of `#ifdef` directives is often employed in a fine-grained program context (i.e., attached to one or a few statements). Opinions and conventional measures may not reveal important nuances in the comprehension of disciplined versus undisciplined annotated code, which may also help to better explain the benefits and drawbacks of annotation discipline. Therefore, there is a need to perform additional controlled experiments that also enable the analysis of complementary indicators about what the developer is doing while trying to comprehend annotated code.

Research Problem: The current research on the impact of code transformations on code comprehension lacks conclusive results and details on possible qualitative and quantitative correlations. Our research problem is considered in the scope of using an eye tracking camera to evaluate constructions with different granularity levels, namely atoms of confusion, refactorings, and #ifdef annotations with 128 novices in three distinct programming languages, C, Python, and Java.

3. Study Goals

In this work, we compare programs versions before and after being transformed for the purpose of understanding how code transformations can be associated with improvements with respect to time, number of answer attempts and visual effort from the point of view of novices in the different programming languages in the context of tasks adapted from introductory programming courses. We address five research questions (RQs):

RQ_1: To what extent does the code transformation affect task completion time? Following prior studies [Gopstein et al. 2017, de Oliveira et al. 2020], to answer this question, we measure how much time the subject spends in the whole program to specify the correct output, in addition to the time in specific areas of the code.

RQ_2: To what extent does the code transformation affect the number of attempts? To answer this question, we measure the number of attempts made by the subject until specifying the correct output of the program.

RQ_3: To what extent does the code transformation affect fixation duration? Longer fixations have been linked to increased attention to the stimuli [Busjahn et al. 2011]. To answer this question, we measure the duration of each fixation in the programs.

RQ_4: To what extent does the code transformation affect fixation count? More fixations have been linked to increased attention to complex code [Sharafi et al. 2015]. To answer this question, we count the number of fixations in the programs.

RQ_5: To what extent does the code transformation affect regressions count? Developers in imperative programming languages may read code left-to-right, top-to-bottom, like natural language. To answer the question, we counted regressive eye movements in the opposite direction of the writing system. For a fair comparison, both versions had loops that iterated over the same number of elements.

4. Method

We conducted an eye-tracking experiment investigating how six atoms of confusion impact code comprehension for 32 students with programming knowledge but limited experience with Python. We compare programs containing six atoms of confusion with functionally equivalent clarified versions to observe how and to what extent they influence the subjects' performance regarding time, number of attempts, and visual effort. Previous studies have used similar metrics to investigate code comprehension [Malaquias et al. 2017, Schulze et al. 2013, Busjahn et al. 2011]. We analyze these metrics in the whole code and the main Area of Interest (AOI). The AOI defines the code region containing the atom or its corresponding clarifying version. In this study, we selected the six atoms that occur in real projects: *Multiple Variable Assignment*, *True or False Evaluation*, *Conditional Expression*, *Operator Precedence*, *Implicit Predicate*, and *Augmented Operator*. Each program containing one of these obfuscating atoms has a functionally equivalent clarified version to be compared. We use the Latin Square design to assign the programs and minimize learning effects. We made all the supplementary material publicly available in our replication package [da Costa et al. 2023].

To investigate the impact of Extract Method refactoring, we conduct a controlled experiment with 32 novices in Java and measure their objective performance with time, number of attempts, and visual effort in the entire code and the AOI. We interview

the novices regarding their perceptions of the difficulties of the programs and analyze the qualitative data using the method of grounded theory. We select eight tasks from introductory programming courses: *Sum Numbers*, *Calculate Next Prime*, *Return Highest Grade*, *Calculate Factorial*, *Count Multiples of Three*, *Calculate Area of Square*, *Check If Even*, and *Count Number of Digits*. For each task, we compare two functionally equivalent versions, one with a method inlined and the other with the method extracted.

To investigate the impact of `#ifdef` annotations, we conducted a controlled experiment with 64 subjects majoritarily novices. We consider all the subjects who know how to program but have little experience in C programming language "novices". We observe how disciplined annotations influence their comprehension with six tasks in terms of time, accuracy, and visual effort. We selected the three refactorings most preferred by developers to discipline annotations according to Medeiros et al. [Medeiros et al. 2018]. The three refactorings differ in various ways: Refactoring 1 ⟨wrapping function call⟩ duplicates a token in a function call to wrap only entire statements with preprocessor directives. Refactoring 2 ⟨undisciplined `if` conditions⟩ resolves undisciplined directives surrounding boolean expressions by defining a fresh variable to maintain the statement's conditions. Refactoring 3 ⟨alternative `if` statements⟩ uses an alternative `if` statement, also defining a fresh variable to keep the statement's condition.

5. Results

In the context of novices dealing with undisciplined `#ifdef` annotations, we observed that the developers have a reduced visual effort. One refactoring added one extra variable and two extra lines of code, which is only a small impact on the LOCs, but it presented reductions in the modified region by 46.9% in the time, 44.7% in the fixation duration, 48.4% in the fixation count, and 60.5% in the regressions count. These results have the potential to provide more insights and deepen the discussion on the advantages or disadvantages of disciplining annotations.

Our results revealed an impact of atoms to a considerable extent. The clarified version of the code containing the *Operator Precedence* reduced the time in the AOI by 38.6%. In the visual metrics, the number of regressions was reduced to the extent of 50%. On the other hand, the clarified version of the code containing the atom *Multiple Variable Assignment* increased the number of regressions reaching the extent of 60%. Thus, even in small and simple programs, we observed a considerable impact of the obfuscated and clarified on the code comprehension. In addition, atoms of confusion also revealed an impact on the focus of attention of subjects and the regions that received most of the eye attention were the regions with atoms [de Oliveira et al. 2020].

Eye tracking allows us to see the impact of adding the parenthesis at a fine-grained level. We isolated the subjects who submitted more than one answer to solve and compared the eye movements between `FirstOperand` → `SecondOperand` and `SecondOperand` → `ThirdOperand`. In the obfuscated version, `FirstOperand` → `SecondOperand` has the `and` operator which has precedence but no parenthesis. We learned that they go back and forth between `FirstOperand`, `SecondOperand`, and `ThirdOperand` with a similar effort. However, we observed that they make `ThirdOperand` → `FirstOperand` nine times and `FirstOperand` → `ThirdOperand` three times, which might be indicative of confusion. The pattern

`ThirdOperand` → `FirstOperand` can indicate wrong precedence involving the `or` operator between the regions (See Figure 3).

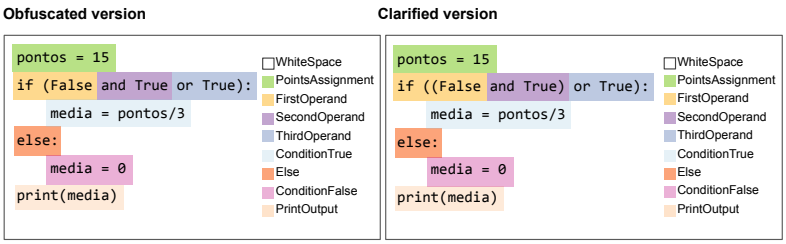
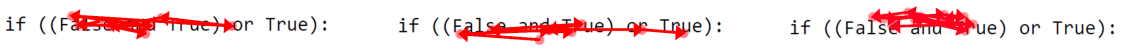


Figura 3. Set of regions inside the code version with *Operator Precedence* atom and in the code with the clarified code version.

We performed a deeper analysis of the gaze transitions to understand the thought process of the subjects. For the clarified version of the *Operator Precedence*, the subjects exhibited transitions going forward and backward concentrating on the expression inside the parentheses. We observed the same transitions for other subjects. Consider the transitions exhibited by the following subjects in Figure 4.



(a) Transitions of Subject 1 **(b)** Transitions of Subject 5 **(c)** Transitions of Subject 13

Figura 4. Sequence of transitions of a subject on the clarified code version with *Operator Precedence*.

In the gaze, we observed transitions going forward and backward between 'False and True' which may indicate that the subject understands which sub-expression should be evaluated first. Subject 5 mentioned that it was easy to solve because of the parenthesis. On the other hand, for the obfuscated version of the *Operator Precedence*, we found transitions that indicate confusion. Consider the gaze transitions exhibited by the following subjects in Figure 5.

(a) Transitions of Subject 2 **(b)** Transitions of Subject 2 **(c)** Transitions of Subject 10

Figura 5. Sequence of transitions of a subject on the obfuscated code version with *Operator Precedence*.

In the gaze, we observed transitions going forward and backward repeatedly between the logical operators '`and`' and '`or`', which may indicate that the subjects were not certain about which operator should come first. Subject 2 mentioned having difficulties with the precedence, made an error in the first submission, and reported that only later on she realized that the '`and`' had precedence over '`or`'. Consider the transitions exhibited by the subjects in Figure 6.

(a) Transitions of Subject 4

(b) Transitions of Subject 4

(c) Transitions of Subject 8

Figura 6. Sequence of transitions of a subject on the clarified code version with *Operator Precedence*.

We observed transitions going forward and backward between '`True or True`' which may indicate that the subject had doubts about which expression should be evaluated first. Indeed, Subject 4 mentioned having difficulties with the '`and`' and '`or`' operator. The lack of parenthesis may lead to more transitions going forward and backward between the logical operators and lead to making wrong associations. Unlike the obfuscated version, in the clarified version we did not observe such transitions going forward and backward eye movements between '`True) or True`'.

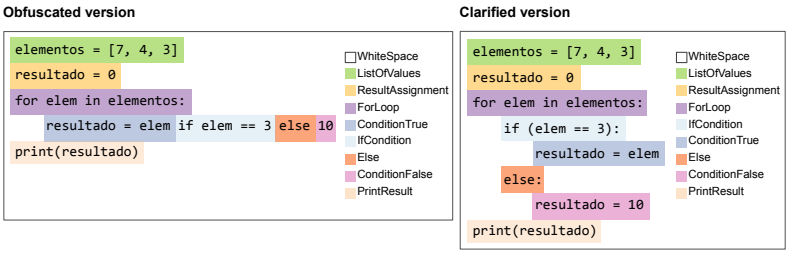


Figura 7. Set of regions inside the code version with *Conditional Expression* atom and in the code with the clarified version of code.

For the obfuscated version of the atom *Conditional Expression* depicted in Figure 7, we observed several transitions going forward and backward in the center of the expression. Several subjects mentioned being confused about the condition line. For instance, one subject mentioned that the ternary `if` was confusing and needed two submissions to solve the task while spending more than twice the average time in the AOI than the subjects in the clarified version. The true condition before the condition being tested can make the subjects go back more times to observe it. For instance, consider the transitions exhibited by the subjects in Figure 8.

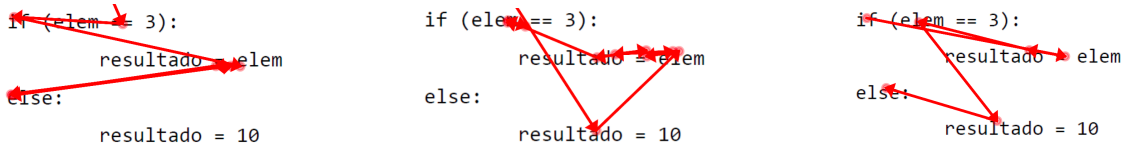
(a) Transitions of Subject 4

(b) Transitions of Subject 2

(c) Transitions of Subject 12

Figura 8. Sequence of transitions of subjects on the clarified code version with *Conditional Expression*.

In the gaze transitions, the subjects often go back in the code to the true condition. Subject 4 regresses three times the average number of regressions in the AOI in the clarified version, which can indicate confusion. We also observed similar transitions going forward and backward for other subjects. On the other hand, in the clarified code, we observed longer transitions between true and false conditions. For instance, consider the transitions exhibited by the subjects in Figure 9.



(a) Transitions of Subject 3

(b) Transitions of Subject 5

(c) Transitions of Subject 9

Figura 9. Sequence of transitions of subjects on the clarified code version with *Conditional Expression*.

The subjects make transitions between statements when the condition is `true` and `false`. However, unlike the obfuscated code, the transitions are not concentrated nor go back and forth so often. Instead, they are in the vertical between lines of code. Subject 5 commented about the structure of the code which she found easy. She solved the tasks in the first submission with half of the time and half of the number of regressions in the AOI compared to the average time and number of regressions in the obfuscated version. Breaking one long line of the condition expression into more lines modified the structure of the code but also seemed to improve the performance of the subjects.

We approach comprehension by triangulating time, attempts, and visual effort. By adding eye tracking, we can infer potential areas with bottlenecks in the code tasks. For instance, when the novices spend more time in the code written in a certain style and make more attempts, we can triangulate this information with the areas of interest in the code to observe where they most fixated, for how long, and where they usually go back in the code. These observations, supported by the interviews, can help us identify the code areas in which the novices faced difficulties, find reading patterns, and give us useful insights into how certain patterns can affect the developers in the comprehension of the code. These insights could provide a meaningful strategy to integrate an eye tracker into a computer so that we can track the eyes of the developers while they observe the code in an Integrated Development Environment (IDE). This strategy can provide the developer with immediate feedback on achieving more productivity.

Concerning the Extract Method version, we observed reductions in the time of two tasks, which reached the extent of 78.8%. For three tasks, the subjects attempt 34.4% less to solve the tasks. Moreover, they solved them without going back 84.6% less often in the code. However, negative effects were also observed for some tasks, reaching an increase of 200% in the visual metrics.

In Figure 10, we built two graphs to depict the distribution of the regressions for two subjects who examine a program to determine the *Highest Grade*. We chose two subjects whose results for time, attempts, and visual metrics were consistent with the

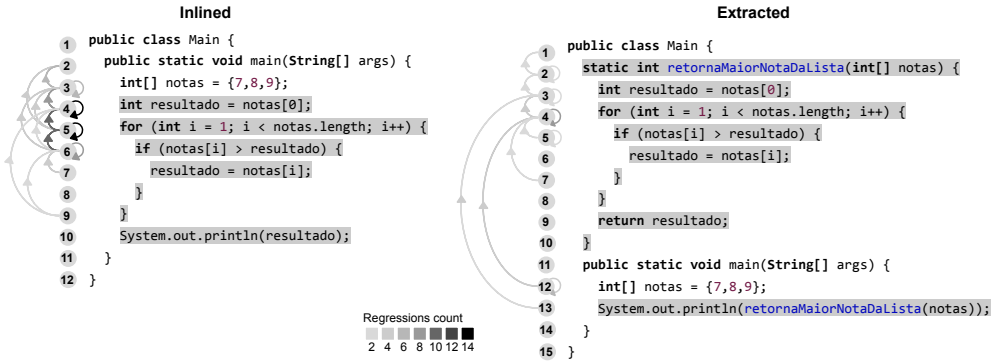


Figura 10. Visual regressions for the inlined and extracted method versions to determine the *Highest Grade*.

others. One subject solved the inlined version, and the other solved the extracted method version. The graph shows visual regressions and the nodes represent lines of code, and the grayscale of edges represents how often the regression occurred.

Comparing the examples, we observe that in the inlined version, the subject goes back more times, especially in the lines where the variables were assigned values and in the loop followed by the decision control. In the extracted version, we observed regressions between the call of the method and the method. Those regressions were expected because the method was located before the call in the code. However, the subject makes fewer regressions while examining the method's body.

Based on the lessons learned from the controlled experiments, we developed a framework for conducting eye tracking studies in the context of behavior-preserving code changes on the code comprehension of novice programmers, such as depicted in Figure 11. We learned a set of lessons about the design of programs, such as the size and style, the use of specific eye tracking metrics, and parameters for fixations detection, among others. In addition, we contribute with visualizations for the visual effort regarding transitions between code regions and horizontal/vertical regressions in the code.

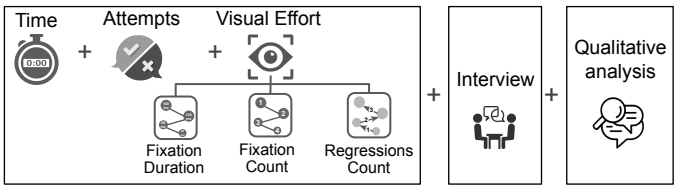
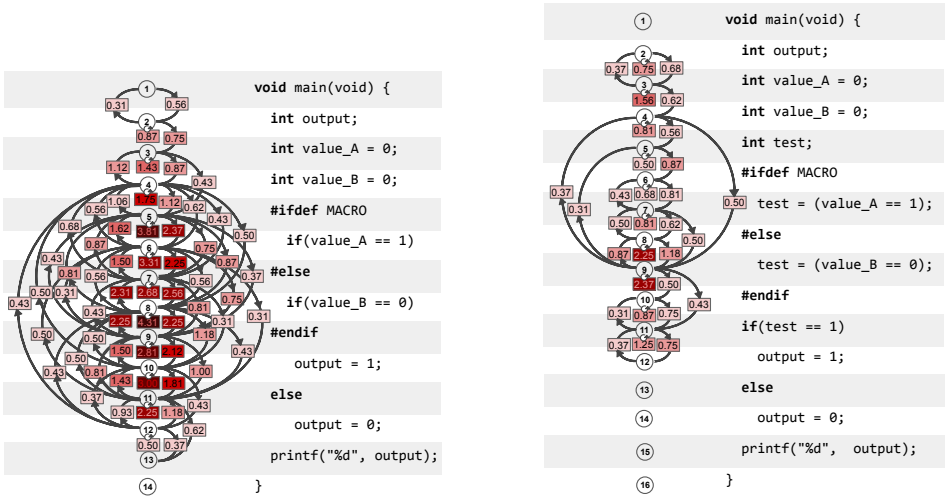


Figura 11. Framework overview for conducting eye tracking studies on code comprehension of novice programmers.

To derive insights and better analyze our data, we developed new graphic visualizations to compare code reading behavior. We map the chronological sequence of gaze transitions between the code lines to visualize and better understand the dynamics of eye movements before and after applying the code transformation. A graph can help us visualize such dynamics with a node representing a line of code and an edge representing a transition, such as in Figure 12 for the `#ifdef` directives. In it, the edge weight represents how frequently the subjects made the same saccades from one specific line to another while performing the same task. The intensity of the colors varies according to



(a) Undisciplined annotations **(b)** Disciplined annotations

Figura 12. Visualization of visual transitions in a graph on a program.

the weight. The darker the red color, the heavier the weight, which implies the more frequently they made the same transition.

6. Related Work

Gopstein et al. [Gopstein et al. 2017] introduced the term "atom of confusion" as the smallest code pattern that can cause misunderstanding in programmers. They found that code snippets containing atoms of confusion are harder to understand than clarified versions. We conducted an experiment using Python and eye-tracking to observe the impact of obfuscated code with atoms of confusion on novice programmers' comprehension. Our approach used more meaningful names for variables and was closer to practical scenarios.

Medeiros et al. [Medeiros et al. 2019] studied the prevalence of atoms of confusion in C open-source projects using a mixed research approach involving experienced developers. Yeh et al. [Yeh et al. 2021] used an EEG device to measure the cognitive load of the developers as they attempted to predict the output of C code snippets. They aimed to observe whether particular patterns within the code snippet induced higher levels of cognitive load. They found that particular patterns indeed affect the developers' cognitive processes. We focused on eye movements instead of brain activity. We explored in more depth the visual effort regarding atoms in Python language.

Langhout and Aniche [Langhout and Aniche 2021] replicated the work of Gopstein et al. [Gopstein et al. 2017], however, in the Java programming language. They found that atoms of confusion can cause confusion among novice developers. We also investigated the potential of atoms to influence the code comprehension of novices negatively. However, we did so in Python language and from the perspective of the eye tracking measures.

Cedrim et al. [Cedrim et al. 2017] used code metrics such as Lines of Code, Coupling Between Objects, and Cyclomatic Complexity to evaluate several refactorings. They found that the Extract Method increases the number of smells in the code, suggesting that

the Extract Method refactoring might have a negative impact on code quality. They also found that the number of smells remained the same after applying Inline Method, making it neutral. The three most common reasons were to reuse code, introduce an alternative method signature, and improve readability. We also investigated the Extract Method, however, through controlled experiments focusing on the extent of its impact on the code comprehension of novices with eye tracking. We also explored their subjective perceptions to understand better the factors that affect their understanding.

Medeiros et al. [Medeiros et al. 2018] proposed a catalog of refactorings to convert undisciplined annotations to disciplined ones. For certain refactorings, developers showed a preference for the disciplined version, while for others, both disciplined and undisciplined versions had similar preferences. However, there is no consensus on whether undisciplined annotations should become disciplined ones in practice [Malaquias et al. 2017, Schulze et al. 2013]. For instance, while one study found that undisciplined annotations are more time-consuming and error-prone [Malaquias et al. 2017], another found no differences regarding task completion time and accuracy [Schulze et al. 2013].

Fenske et al. [Fenske et al. 2020] conducted a controlled study involving both an experiment and questionnaires with 521 experienced developers to understand the impact of refactoring C preprocessor directives. They evaluated coarse-grained transformations converting from undisciplined to disciplined annotations instead of evaluating a single fine-grained transformation. We used an eye tracker to analyze how disciplining annotations impact code comprehension using a different perspective not considered in other studies. We observed differences for two of the three refactorings evaluated.

7. Conclusions and Contributions

We report on three controlled experiments with an eye tracking camera to evaluate the impact of code transformations on the code comprehension of 128 novices in three distinct programming languages, C, Python, and Java using constructions with different granularity levels, namely atoms of confusion, refactorings, and #ifdef annotations.

With an eye tracker, we revealed an impact of code transformations that could not be captured by static code metrics: 1) the time spent by subjects specifically in the region containing the atom of confusion, method extraction, or `#ifdef` annotation; 2) The number of fixations in the code and/or in the modified area; 3) The duration of fixations in the modified area; 4) The number of eye movement regressions in the code or specifically in the region containing the atom, method extraction, or `#ifdef` annotation.

These visual metrics allowed us to understand better the impact of clarified atoms, extracted methods, and disciplined #ifdef annotations on the visual effort – previous works have measured the impact on time and accuracy, but none of them could measure to what extent the atoms impacted the fixation duration, fixations count, and regressions count. We also better understood how and measured the extent of the impact of clarified atoms, extracted methods, and disciplined #ifdef annotations on the code reading. These investigations resulted in published articles in journals and international collaborations.

For the research community, our study setup exploring the visual effort dimension contributes to nuances not observed by previous works. For instance, in the analysis of

the visual data for code containing the *Multiple Variable Assignment*, we perceived that the use of multiple assignments within the same line impacted the way the subjects read the code. The code with *Multiple Variable Assignment* allowed the subjects to read the assignments in a more direct manner, with 60% fewer regressions in the AOI. When the assignments are split between two lines, to make the code clearer, the subjects tended to make more vertical regressions and to keep coming back to those lines, transitioning between those lines and the lines of code that later use them. Hopefully, this will encourage researchers to consider eye tracking as a promising alternative to evaluate atoms of confusion. Other dimensions, such as mapping neural activities with Functional Magnetic Resonance Imaging (fMRI) or tracking all the subjects' activity during the experiments, could possibly reveal other insights and allow us to dive deeper into how this atom impacts difficulty beyond visual effort. This can be a future direction for research.

In the gaze transitions of the *Conditional Expression*, the subjects often go back in the code to the true condition. One subject regressed three times the average number of regressions in the AOI in the clarified version, which can indicate confusion. We observed similar transitions going forward and backward for other subjects. On the other hand, in the clarified code, we observed longer transitions between true and false conditions. In the gaze transitions of the *Operator Precedence*, we observed transitions going forward and backward repeatedly between the logical operators '`and`' and '`or`', which may indicate that the subjects were not certain about which operator should come first. One subject mentioned difficulties with the precedence, made an error in the first submission, and reported that only later on she realized that '`and`' had precedence over '`or`'

In future work, we envision the development of a tool that tracks the eye movements of developers and assists them with suggestions to improve the code. They can be implemented in IDEs that can monitor eye movements and identify in real-time when developers are confused. For instance, if a tool detects transitions going forward and backward eye movements between two operators such as '`and`' and '`or`', or between an expression that has no precedence, a tip should arise to add the parentheses. We envision proposing heuristics or building a model whereby a programmer receives an arbitrary source code, and we use eye tracking data to identify which elements were atom candidates or infer confusing regions that impacted negatively the code comprehension.

Referências

Busjahn, T., Schulte, C., and Busjahn, A. (2011). Analysis of Code Reading to Gain More Insight in Program Comprehension. Koli Calling'11, pages 1–9.

Cedrim, D., Garcia, A., Mongiovi, M., Gheyi, R., Sousa, L., de Mello, R., Fonseca, B., Ribeiro, M., and Chávez, A. (2017). Understanding the Impact of Refactoring on Smells: A Longitudinal Study of 23 Software Projects. ESEC/FSE'17, page 465–475.

da Costa, J. A. S., Gheyi, R., Castor, F., de Oliveira, P. R. F., Ribeiro, M., and Fonseca, B. (2023). Seeing Confusion through a New Lens: on the Impact of Atoms of Confusion on Novices' Code Comprehension. *Empirical Software Engineering*, to appear(to appear):to appear.

da Costa, J. A. S., Gheyi, R., Ribeiro, M., Apel, S., Alves, V., Fonseca, B., Medeiros, F., and Garcia, A. (2021). Evaluating Refactorings for Disciplining #ifdef Annotations: An Eye Tracking Study with Novices. *Empirical Software Engineering*, 26(5):1–35.

da Costa et al. (2023). "Supplementary Material for the Thesis (Artifacts)". At `https://github.com/josealdo/material-complementar-tese`.

de Oliveira, B., Ribeiro, M., da Costa, J. A. S., Gheyi, R., Amaral, G., de Mello, R., Oliveira, A., Garcia, A., Bonifácio, R., and Fonseca, B. (2020). Atoms of Confusion: The Eyes Do Not Lie. SBES'20, pages 243–252.

Fenske, W., Krüger, J., Kanyshkova, M., and Schulze, S. (2020). #ifdef Directives and Program Comprehension: The Dilemma between Correctness and Preference. ICSME'20.

Fowler, M. (1999). *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley/Longman.

Gopstein, D., Iannacone, J., Yan, Y., DeLong, L., Zhuang, Y., Yeh, M. K.-C., and Cappos, J. (2017). Understanding Misunderstandings in Source Code. ESEC/FSE'17, pages 129–139.

Langhout, C. and Aniche, M. (2021). Atoms of Confusion in Java. (ICPC'21), pages 25–35. IEEE.

Liebig, J., Kästner, C., and Apel, S. (2011). Analyzing the discipline of preprocessor annotations in 30 million lines of C code. AOSD'11, pages 191–202.

Malaquias, R., Ribeiro, M., Bonifácio, R., Monteiro, E., Medeiros, F., Garcia, A., and Gheyi, R. (2017). The Discipline of Preprocessor-Based Annotations – Does #ifdef TAG n't #endif Matter. ICPC'17, pages 297–307. IEEE.

Medeiros, F., Lima, G., Amaral, G., Apel, S., Kästner, C., Ribeiro, M., and Gheyi, R. (2019). An Investigation of Misunderstanding Code Patterns in C Open-source Software Projects. *Empirical Software Engineering*, 24(4):1693–1726.

Medeiros, F., Ribeiro, M., Gheyi, R., Apel, S., Kästner, C., Ferreira, B., Carvalho, L., and Fonseca, B. (2018). Discipline matters: Refactoring of preprocessor directives in the #ifdef hell. *IEEE Transactions on Software Engineering*, 44(5):453–469.

Medeiros, F., Rodrigues, I., Ribeiro, M., Teixeira, L., and Gheyi, R. (2015). An empirical study on configuration-related issues: investigating undeclared and unused identifiers. In *(GPCE'15)*, pages 35–44.

Mongiovi, M., Gheyi, R., Soares, G., Ribeiro, M., Borba, P., and Teixeira, L. (2018). Detecting overly strong preconditions in refactoring engines. *IEEE Transactions on Software Engineering*, 44(5):429–452.

Schulze, S., Liebig, J., Siegmund, J., and Apel, S. (2013). Does the Discipline of Preprocessor Annotations Matter?: A Controlled Experiment. GPCE '13, pages 65–74.

Sharafi, Z., Shaffer, T., Sharif, B., and Guéhéneuc, Y.-G. (2015). Eye-tracking Metrics in Software Engineering. APSEC'15, pages 96–103. IEEE.

Soares, G., Gheyi, R., and Massoni, T. (2013). Automated behavioral testing of refactoring engines. *IEEE Transactions on Software Engineering*, 39(2):147–162.

Yeh, M. K.-C., Yan, Y., Zhuang, Y., and DeLong, L. A. (2021). Identifying Program Confusion Using Electroencephalogram Measurements. *Behaviour & Information Technology*, pages 1–18.