
**UNIVERSIDADE FEDERAL DE CAMPINA GRANDE**

**CENTRO DE ENGENHARIA ELÉTRICA E INFORMÁTICA**

**PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO**

**JOSÉ ALDO SILVA DA COSTA**

# EVALUATING THE IMPACT OF REFACTORINGS ON THE CODE COMPREHENSION OF NOVICES WITH EYE TRACKING

**CAMPINA GRANDE - PB**

**2023**

**JOSÉ ALDO SILVA DA COSTA**

# EVALUATING THE IMPACT OF REFACTORINGS ON THE CODE COMPREHENSION OF NOVICES WITH EYE TRACKING

Tese apresentada ao Programa do Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Campina Grande - Campus I, pertencente à linha de pesquisa ENGENHARIA DE SOFTWARE e área de concentração CIÊNCIA DA COMPUTAÇÃO, como requisito para obtenção do grau de Doutor em Ciência da Computação.

Orientadores: PROF. DR. ROHIT GHEYI (ORIENTADOR)

**CAMPINA GRANDE - PB**

**2023**

C837e Costa, José Aldo Silva da.
Evaluating the impact of refactorings on the code comprehension of novices with eye tracking / José Aldo Silva da Costa. - Campina Grande, 2023.
184 f. : il. color.

Tese (Doutorado em Ciência da Computação) - Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática, 2022.
"Orientação: Prof. Dr. Rohit Gheyi."
Referências.

1. Refatoramentos. 2. Compreensão de Código. 3. Eye Tracking. I. Gheyi, Rohit. II. Título.

CDU 004.41(043)

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECÁRIA ITAPUANA SOARES DIAS GONÇALVES CRB-15/93

MINISTÉRIO DA EDUCAÇÃO
**UNIVERSIDADE FEDERAL DE CAMPINA GRANDE**
POS-GRADUACAO CIENCIAS DA COMPUTACAO
Rua Aprigio Veloso, 882, - Bairro Universitario, Campina Grande/PB, CEP 58429-900

## FOLHA DE ASSINATURA PARA TESES E DISSERTAÇÕES

**JOSÉ ALDO SILVA DA COSTA**

EVALUATING THE IMPACT OF REFACTORINGS ON THE CODE COMPREHENSION OF NOVICES WITH EYE TRACKING

Tese apresentada ao Programa de Pós-Graduação em Ciência da Computação como pré-requisito para obtenção do título de Doutor em Ciência da Computação.

Aprovada em: 13/12/2022

Prof. Dr. ROHIT GHEYI, Orientador, UFCG

Prof. Dr. TIAGO LIMA MASSONI, Examinador Interno, UFCG

Prof. Dr. HYGGO OLIVEIRA DE ALMEIDA, Examinador Interno, UFCG

Prof. Dr. VANDER RAMOS ALVES, Examinador Externo, UnB

Prof. Dr. FERNANDO JOSÉ CASTOR DE LIMA FILHO, Examinador Externo, UFPE

Documento assinado eletronicamente por **ROHIT GHEYI**, **PROFESSOR(A) DO MAGISTERIO SUPERIOR**, em 13/12/2022, às 13:56, conforme horário oficial de Brasília, com fundamento no art. 8º, caput, da Portaria SEI nº 002, de 25 de outubro de 2018.

Documento assinado eletronicamente por **HYGGO OLIVEIRA DE ALMEIDA**, **PROFESSOR(A) DO MAGISTERIO SUPERIOR**, em 13/12/2022, às 14:45, conforme horário oficial de Brasília, com fundamento no art. 8º, caput, da Portaria SEI nº 002, de 25 de outubro de 2018.

Documento assinado eletronicamente por **VANDER RAMOS ALVES**, **Usuário Externo**, em 13/12/2022, às 20:52, conforme horário oficial de Brasília, com fundamento no art. 8º, caput, da Portaria SEI nº 002, de 25 de outubro de 2018.

Documento assinado eletronicamente por **TIAGO LIMA MASSONI**, **COORDENADOR(A) ADMINISTRATIVO(A)**, em 14/12/2022, às 10:11, conforme horário oficial de Brasília, com fundamento no art. 8º, caput, da Portaria SEI nº 002, de 25 de outubro de 2018.

Documento assinado eletronicamente por **Fernando José Castor de Lima Filho**, **Usuário Externo**, em 15/12/2022, às 13:30, conforme horário oficial de Brasília, com fundamento no art. 8º, caput, da Portaria SEI nº 002, de 25 de outubro de 2018.

A autenticidade deste documento pode ser conferida no site https://sei.ufcg.edu.br/autenticidade, informando o código verificador **2970824** e o código CRC **26A4A5F2**.

# Resumo

A compreensão do código é fundamental para a manutenção e evolução do software, porém, pode ser prejudicada por problemas estruturais no código. Para resolver os problemas estruturais no código e torná-lo mais fácil de ler e entender, os desenvolvedores costumam usar refatoramentos. Um refatoramento é uma técnica disciplinada de reestruturação do código que visa melhorar sua estrutura interna preservando seu comportamento. No entanto, o estado-da-arte sobre o entendimento do impacto de refatoramentos na compreensão do código necessita de resultados conclusivos e detalhes quantitativos/qualitativos sobre como e o porquê de possíveis correlações. Enquanto alguns estudos encontraram um impacto divergente de refatoramentos na compreensão do código, outro descobriu que certos refatoramentos levaram à introdução de mais problemas estruturais. Com o objetivo de investigar o impacto das refatoramentos na compreensão do código, realizamos três estudos controlados com rastreio ocular: o primeiro sobre o impacto de átomos de confusão clarificados com 32 novatos em Python, o segundo sobre o impacto do refatoramento Extrair Método com 32 novatos em Java, e o terceiro sobre o impacto das anotações `#ifdef` com 64 majoritariamente novatos na linguagem C. Além de usar vários critérios como tempo, número de tentativas e opiniões, medimos o esforço visual dos sujeitos com rastreamento ocular por meio da duração da fixação, contagem de fixações e contagem de regressões. Em nossos resultados, a versão de código clarificada de um dos átomos reduziu o tempo em 38,6% e o número de tentativas de resposta em 28%. Além disso, observamos 47,3% menos regressões horizontais na região do átomo facilitando sua leitura. O uso do refatoramento Extrair Método apresentou uma redução significativa no tempo de duas tarefas, que variou de 70% a 78,8%. Observamos um aumento na acurácia de três tarefas, que variou de 20% a 34,4%. Os sujeitos resolveram essas tarefas com o Extrair Método voltando visualmente no código com 74,4% a 84,6 % menos frequência comparado ao Inline Método. No contexto das anotações `#ifdef`, um dos refatoramentos adiciona uma variável extra e duas linhas extras de código, o que é apenas um pequeno impacto na métrica Linhas de Código (LOC), mas apresentou reduções na região modificada em 46,9% no tempo, 44,7% na duração da fixação, 48,4% na contagem de fixação e 60,5% na contagem de regressões. Esses resultados contribuem para

conscientizar educadores sobre certos refatoramentos e seu potencial para facilitar ou dificultar a compreensão de código de novatos em Python, Java e C. Praticantes e designers de linguagem de programação devem ser mais cuidadosos ao usar ou propor refatoramentos que possam prejudicar a capacidade dos novatos de entender o código. Para os pesquisadores, esses resultados mostram o potencial das métricas visuais para revelar um impacto de refatoramentos que não podem ser capturados por métricas estáticas de código.

**Palavras-chave:** Refatoramentos; Compreensão de Código; Rastreio Ocular.

# Abstract

Code comprehension is crucial for software maintenance and evolution, however, it can be hindered by structural problems in the code. To address the structural problems in the code and make it easier to read and understand, developers often use refactorings. Refactoring is a disciplined technique for restructuring the code that aims to improve its internal structure preserving its behavior. However, the state-of-the-art on understanding the impact of refactorings on code comprehension lacks conclusive results and quantitative/qualitative details on how and why of possible correlations. While some studies found a divergent impact of refactorings on code comprehension, another found that certain refactorings led to the introduction of more structural problems. Aiming to further investigate the impact of refactorings on code comprehension, we conduct three controlled studies with eye tracking: the first one on the impact of clarified atoms of confusion with 32 novices in Python, the second one on the impact of Extract Method refactoring with 32 novices in Java, and the third one on the impact of `#ifdef` annotations with 64 majoritarily novices in the C language. Besides using multiple criteria such as time, the number of attempts, and opinions, we measured the visual effort of the subjects with eye tracking through fixation duration, fixations count, and regressions count. In our results, the clarified version of the code with an atom reduced the time to the extent of 38.6% and the number of answer attempts by 28%. In addition, we observed 47.3% fewer horizontal regressions count in the atom region, making its reading easier. The use of the Extract Method refactoring presented a significant reduction in the time of two tasks, which varied from 70% to 78.8%. We observed an increase in the accuracy of three tasks, which varied from 20% to 34.4%. The subjects solved these tasks with the Extract Method going back visually in the code 74.4% to 84.6% less often compared to the Inline Method. In the context of `#ifdef` annotations, one of the refactorings adds one extra variable and two extra lines of code, which is only a small impact on the metric Lines of Code (LOC), 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 raise educators' awareness about certain refactorings and their potential to ease or hinder the code comprehension for novices in Python, Java, and C. Practitioners and language designers should be more careful when using or proposing refactorings that

could possibly impair the novices' abilities to understand the code. For researchers, these results show the potential of visual metrics to reveal an impact of refactorings that cannot be captured by static code metrics.

**Keywords:** Refactorings; Code Comprehension; Eye Tracking.

*Fixing our eyes on Jesus, the pioneer and perfecter of faith. For the joy set before Him He endured the cross, scorning its shame, and sat down at the right hand of the throne of God.*

*Hb. 12:2*

# Acknowledgments

First of all, I wholeheartedly thank God for giving me the opportunity, the strength, and all blessings that made this possible. Certainly, with Him everything is possible.

I wholeheartedly thank my family for their love, my parents Maria das Neves and José Vicente, my brother Júnior, my sister Leidiane and her husband Daniel. You were always there for me, encouraging and supporting me through all these years. In addition, I thank my bother Júnior for participating and giving valuable feedback in the trials of my experiments.

I sincerely thank my advisor, Rohit Gheyi, for your guidance, patience, and support during this entire time. You helped me put things into perspective and prepared me for great things in the future. Your concern and genuine desire to help me become a better professional made a difference.

Many thanks to my church, the leaders, and all my friends for your support in prayers. I could not have gotten this far without you.

I thank the collaborators and contributors to this work Márcio Ribeiro, Vander Alves, Baldoino Fonseca, Rodrigo Bonifácio, Sven Apel, Alessandro Garcia, Flávio Medeiros, Volker Stolz, for your insightful feedback and comments. I thank Tiago Massoni, Fernando Castor, Rafael Mello, Leonardo Rabello, Tayana Conte, Auri Vincenzi, Hyggo Almeida, and the anonymous reviewers. Your insightful suggestions have improved this work.

I thank the Coordenação da Pós-graduação em Computação da UFCG (COPIN) for the professionalism in all the concerns regarding this entire course.

Finally, I thank the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) for the support in funding.

# Contents

# List of Figures

# List of Tables

# Listings

# Chapter 1

# Introduction

Code comprehension is crucial for software maintenance and evolution processes. However, it can be hindered by structural problems in the code. To address these structural problems and make the code easier to read and understand, developers often use refactorings. Refactoring can be understood as the process of improving the code structure to make it easier to understand, evolve, and maintain. It consists of a disciplined technique for transforming the internal structure without changing its observable behavior [40; 86]. Each transformation, or refactoring, consists of a small change in the code and a sequence of transformations can produce a significant amount of restructuring.

Fowler [40] proposed a catalog of refactorings that became popular for addressing a set of indicators of structural problems in the code called code smells. For instance, the more lines we find in a method, the harder it is to understand what the method does. In that case, we should apply the Extract Method refactoring, which helps isolate independent parts of code, make code less duplicated, and more readable and easier to understand.

Other catalogs of refactorings have been proposed by other researchers to deal with other smelly scenarios. Developers often use preprocessor directives, such as `#ifdef` and `#endif`, to make a block of source optional or conditional, with the purpose of tailoring software systems to different hardware platforms, operating systems, and application scenarios. However, the directive can have a negative impact on code understanding and maintainability. Medeiros et al. [75] proposed a catalog of refactorings to improve the code, making it easier to understand, evolve, and maintain.

Gopstein et al. [48] presented a catalog of atoms of confusion for the C language. Atoms

of confusion are the smallest units of code that can cause confusion in the developers when they read the code, making them misinterpret the code behavior. Gopstein et al. [48] proposed clarified alternatives for the confusing units. They compared the code containing atoms likely to cause confusion (obfuscated code) to functionally equivalent code hypothesized to be less confusing (clarified code). They experimentally showed that obfuscated code increased confusion among the subjects, largely composed of students, and the proposed clarified versions made the code easier to understand.

## 1.1 Problem

The understanding of the impact of refactorings on code comprehension lacks conclusive results. In this section, we explain the research gaps in the three aforementioned scenarios that deal with code transformations aiming to improve code comprehension. We present them in perspective from the smallest units, atoms of confusion (Section 1.1.1), going through bigger units, Extract Method refactoring (Section 1.1.2), and more complex units, configurable systems (Section 1.1.3).

### 1.1.1 Atoms of confusion

Often a developer reads the source code written by another developer, trying to understand its behavior. However, the developer's interpretation of a piece of code can differ from that of the one who wrote the code due to tiny patterns that can cause misunderstandings. These tiny patterns can obfuscate the code and confuse developers causing them to misjudge its behavior. The tiny patterns are called atoms of confusion [48; 49; 66] when there are functionally equivalent alternatives that lead to better performance.

Atoms of confusion are prevalent in open-source projects in C language. Investigating the presence of atoms of confusion in 50 C open-source software projects, Medeiros et al. [73] found more than 109,000 occurrences of 11 out of 12 atoms considered in their study. Some of these projects, such as *Apache*, *OpenSSL*, and *Python* comprise more than 200,000 lines.

Atoms of confusion also occur in other programming languages, such as Python, one of the most used languages nowadays.[1] For instance, in Figure 1.1(a), we illustrate a *Con-*

[1] https://madnight.github.io/githut/#/pull_requests/2021/4

*ditional 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 equals to three; otherwise, `num` receives one. If the implications of the study of Gopstein et al. [48] for C language are sustainable for Python as well, the *Conditional Expression* in Line 4 impairs the code understanding of undergraduate students because the assignment depends on the value of a variable, which can confuse the student about the behavior of the code. Thus, the code in Figure 1.3(a) can be obfuscated by the presence of the atom.

```
1 elements = [7, 4, 3]
2 num = 0
3 for elem in elements:
4  num = elem if elem == 3 else 1
5 print(num)
```

(a)

```
1 elements = [7, 4, 3]
2 num = 0
3 for elem in elements:
4  if (elem == 3):
5   num = elem
6  else:
7   num = 1
8 print(num)
```

(b)

Figure 1.1: Code adapted from *SwiftShader* with (a) obfuscated code containing the atom *Conditional Expression*, and (b) the clarified version of the code.

To clarify this source of confusion, Medeiros et al. [73] proposed an alternative solution which we adapted for Python and presented in Figure 1.3(b). In it, the line that contains the atom becomes four lines of code and the variable `num` is used twice, depending on the condition of the `if` statement in line 4. Besides proposing a clarified version of the code that contains the atom of confusion, Medeiros et al. [73] also investigated the subjective perception of experienced developers regarding the atom for the C language. Based on the answers of the developers, the code with the atom did not influence the understanding of the subjects negatively. In addition, the developers accepted pull requests with both obfuscated and clarified versions.

Langhout and Aniche [66] derived a set of atoms of confusion based on the work of Gopstein et al. [48], however, for Java language and performed a two-phase experiment with students investigating accuracy and perception. Out of 14 atoms, four presented results that were distinct from those presented in the study of Gopstein et al. [48]. One of these atoms

that presented distinct results was the *Conditional Operator*.

In a more recent qualitative study, Gopstein et al. [47] raised a serious concern. They pointed out that studies based only on the accuracy may be under-reporting the amount of code misunderstandings. For instance, using a qualitative research approach, they found that, even for subjects that evaluated a program in a correct manner, there was still significant confusion unnoticed in the program. The accuracy only tells us the outcome of programmers' performance, but not how or why they behaved that way. We need to investigate this using other perspectives combined with accuracy.

The structural transformation applied to Figure 1.1(a) turning it into Figure 1.1(b) aimed at making the code clearer to understand while preserving its behavior. We consider such transformation as refactoring in the sense it can be characterized as behavior-preserving code transformations aiming to improve the code [40]. Thus, when we refer to the clarified code version, we have in mind a refactored version aiming to clarify the code.

To contribute to the debate, we need more empirical evidence from a finer-grained perspective to better understand which atoms of confusion can affect the code comprehension and to what extent they do so. Since atoms are fine-grained code elements, coarser-grained approaches to assess code comprehension may be insufficient to capture their impact. A prior study showed the potential of eye tracking to investigate the effect of small-grained code changes on the code comprehension [30]. An eye tracker makes it possible to record the eye movements of human subjects and assess their visual attention [91]. The eye tracking data allowed researchers to mainly assess visual attention and effort by investigating where the subject is fixating, the duration of their fixations, and how the fixations switched from one location to another [99; 100; 16].

For instance, we simplified a sequence of fixations performed by two subjects in Figure 1.2. Each red circle represents a fixation that varies in size according to its duration. The sequence and direction of fixations are depicted in chronological order with a number inside. In the obfuscated version (Figure 1.2(a)), the subject makes eight fixations with six within the line of the atom (Line 4). In the clarified version (Figure 1.2(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 see nuances not observed by previous works. Thus, besides measuring time and number of attempts, we investigate the effects of atoms of confusion on visual effort through fixation duration, fixations count, and regressions count.

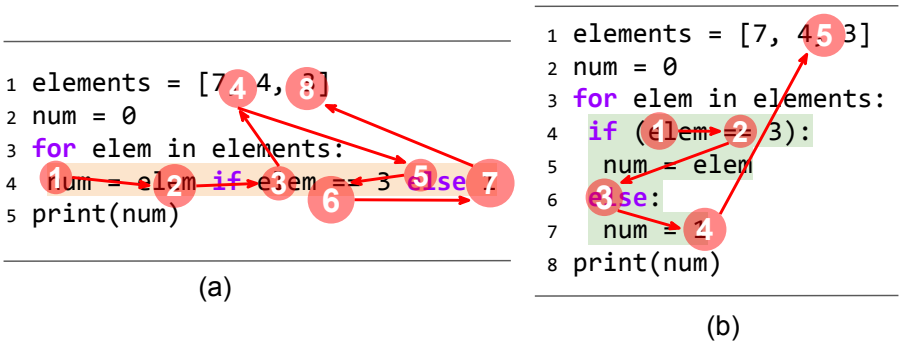



Figure 1.2: Code with eye gaze patterns for (a) obfuscated code containing the atom *Conditional Expression*, and (b) the clarified version of the code.

Previous studies have investigated the impact of atoms of confusion on code comprehension using multiple criteria such as time, answer correctness, and opinions. They compare obfuscated code to functionally equivalent code hypothesized to be less confusing (clarified code) [48]. For instance, Gopstein et al. [48] presented a catalog of atoms of confusion for the C language and experimentally showed that obfuscated code increased confusion by reducing the subjects' accuracy in comparison to their accuracy when analyzing clarified code. Besides showing that atoms are often found in the C real projects, Medeiros et al. [73] investigated the developers' opinions regarding the atoms of confusion. The subjects perceived obfuscated code to be more confusing. Combining accuracy and opinions, Langhout and Aniche [66] found that developers made more mistakes and perceived obfuscated code to be more confusing and less readable in the Java language. Combining different perspectives can give insights into the impact of atoms of confusion on code comprehension that a single perspective cannot. The subjects' answer correctness in an experiment can tell us the outcome of developers' performance but fail to show how or why they behaved in a certain way [47]. We need to observe the phenomenon from a combination of perspectives to understand better

how it impacts code comprehension.

Oliveira et al. [33] performed a controlled experiment including students and professionals that compared the obfuscated and clarified versions of the code with atom *Conditional Operator*. However, Oliveira et al. [33] investigated the objective performance of the developers solving tasks with both versions using an eye tracker. Their study did not find differences between the two versions regarding time, answer correctness, and visual attention in the main area of interest in their study, agreeing with the developers' perception according to Medeiros et al. [73]. However, Oliveira et al. [33] did not consider the fixation duration, fixations count, and regressions count to measure the extent of the impact of the atom on the visual effort of the subjects. Neither investigated how the atom affected the way the subjects read the code by distinguishing between horizontal and vertical visual transitions for all the subjects, which could give more insights into code reading.

### 1.1.2 Extract Method

Refactorings such as Extract and Inline Method are among the most commonly used [79; 56]. While the Extract Method takes a clump of code and turns it into its own method, the Inline Method is essentially the opposite, taking a method call and replacing it with the body of the code. Developers use Extract and Inline Method mainly to improve the code structure and then make the code clearer, more comprehensible, and easier to read [107].

According to Fowler [40], short methods with names that show their intention lead to clearer, more understandable, and easier-to-read code. However, it still needs more empirical evidence to better understand the impact of these practices, because under certain perspectives, extracting a method might have harmful effects instead. For instance, Cedrim et al. [19] 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 Extract Method is considered the "Swiss army knife of refactorings" [107]. According to the developers' opinions, the three most common motivations were to make code reusable, introduce alternative method signatures, and improve code readability. However,

there might be circumstances when extracting a method can instead hamper the code readability, and the Inline Method becomes a better alternative [40].

However, the notion of improving code readability is not always clear. Despite coding guidelines and standards found in the literature, we lack a better understanding of what circumstances developers should opt for inlining or extracting a method. Guidelines and standards can often be subjective, vague, and based on personal preferences, which can differ from the subjective preferences and performance of the developers.

For instance, in Figure 1.3, we depict two iterative programs to find the factorial of a number adapted from Geeksforgeeks.[2] On the left-hand side, in Figure 1.3(a), we have the inlined method version and on the right-hand side, in Figure 1.3(b), the version with the extracted method. While both print the same output, to ease code understanding, the extracted method version adds three more lines of code and uses a name that shows the intention of the method. However, making such code change might introduce side effects. From a quantitative perspective using static code metrics, Cedrim et al. [19] show that extracting a method can actually increase the number of code smells.

```java
public class Main{
  public static void main(String[] args){
    int num = 5;
    int result = 1;
    for (int i = 2; i <= num; i++){
      result = result * i;
    }
    System.out.println(result);
  }
}
```

(a)

```java
public class Main{
  static int calculateFactorial(int num){
    int result = 1;
    for (int i = 2; i <= num; i++) {
      result = result * i;
    }
    return result;
  }
  public static void main(String[] args){
    int num = 5;
    System.out.println(
        calculateFactorial(num));
  }
}
```

(b)

Figure 1.3: (a) Inline Method version of the code that calculates the factorial of a number, and (b) the Extract Method version of the code.

Nonetheless, code metrics can be insufficient to capture differences in small-grained code changes, such as in Figure 1.3. Previous works compared distinct code styles measuring time and answer correctness [48; 70; 95; 76], while the opinions and preferences of the developers can help researchers understand the impact of these code changes on the developers' perceptions [72; 39]. Prior studies used eye tracking to investigate the effect of

[2] https://www.geeksforgeeks.org/program-for-factorial-of-a-number/

small-grained changes on code comprehension [30; 33; 77]. Eye tracking allows recording the eye movements to assess visual effort [91] through eye fixations, duration of fixations, and regressions [99; 100; 16].

For instance, in Figure 1.4, we depict an example of a sequence of fixations of two subjects. Red circles represent fixations varying in size according to their duration. In the Inline Method version (Figure 1.4(a)), the subject fixated in eight different locations, seven within Lines 4–8. In the Extract Method version (Figure 1.4(b)), the subject fixated in five distinct locations, with three of them within Lines 2–8 and one in Line 11. The subject fixated more and for a longer time in the Inline Method version and regresses visually more times, one vertically from Line 8 to 6 and one time horizontally within Line 6. In the Extract Method version, the subject regresses once, between Lines 11 to 2. Examining their visual behavior at this small-grained level, we can see nuances not observed in previous works, such as the visual effort required by the transformation in terms of how subjects fixate their eyes on the code, and how it impacts the code reading, in terms of vertically or horizontal eye movements in the code.

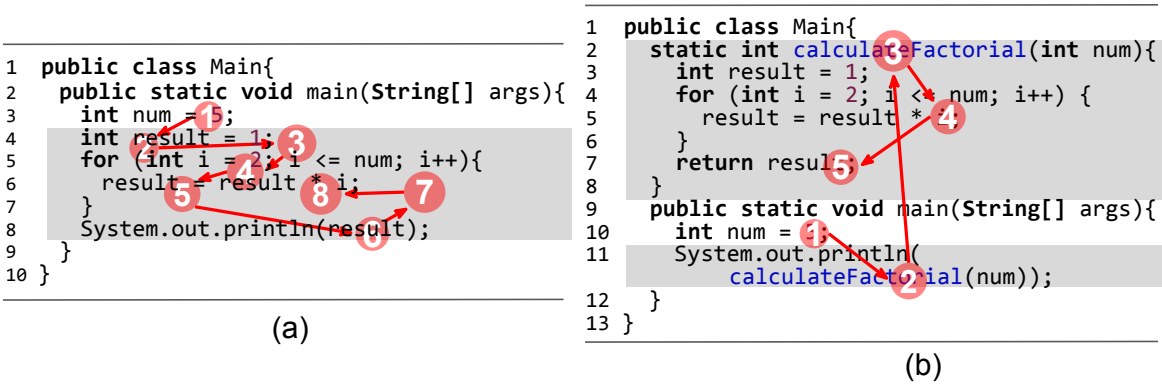



Figure 1.4: Code with eye gaze patterns for (a) the Inline Method version of the code that calculates the factorial of a number, and (b) the Extract Method version of the code.

Despite these results, we lack empirical studies that consider other perspectives, such as the human factor. Opinions and code metrics solely analyzed fail to capture the extent of the impact of fine-grained refactorings in the code comprehension of human subjects. We need more controlled experiments involving human subjects to better understand how it impacts code comprehension.

### 1.1.3 `#ifdef` annotations

The C preprocessor is widely used in practice, such as in *Linux*, and provides mechanisms to implement variability through conditional compilation [34]. Conditional compilation allows developers to conditionally include selected blocks of source code by annotating the code using directives, such as `#ifdef`. There are two types of annotations, *undisciplined* (or incomplete), and *disciplined* (or complete) [69]. Disciplined annotations are preprocessor directives that enclose only whole statements while undisciplined annotations do not, for example, wrapping only an opening bracket of a statement but not the closing one [45] as shown in Figure 1.5. Both achieve the same purpose but they differ in terms of whether they align with the syntactic structure of the code. According to Liebig et al. [69], *undisciplined* annotations are ill-formed annotations in which the number of `#ifdef` and `#endif` statements does not match, and they can produce syntax errors when removed, such wrapping the opening bracket without the closing one. According to Schulze et at. [95], *undisciplined* annotations include arbitrary annotations of code fragments, for instance, a single function parameter, and are difficult to refactor using tool support. We aligned our definition with these previous studies [69; 45; 95]. Based on this definition, Liebig et al. [69] analyzed 40 software projects with over 30 million lines of C code regarding the discipline of their annotations. They found that 84% of all annotations are disciplined.

Despite the relevance and prevalence of conditional compilation in practice, existing evidence confirms that comprehending code with `#ifdef` directives is far from trivial [109; 70; 75; 72]. Code with either disciplined or undisciplined annotations may affect program comprehension. However, empirical knowledge on the influence of the annotation discipline is still scarce. Medeiros et al. [75] proposed a catalog of refactorings to convert undisciplined annotations to disciplined ones. The refactorings were evaluated with respect to the preference of 246 developers regarding disciplined or undisciplined annotated code. For certain refactorings, developers showed a preference for the disciplined version, while for others, both disciplined and undisciplined versions had similar rates of preference. 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 [70; 95]. For instance, Malaquias et al. [70] conducted an experiment comparing undisciplined

annotations and their refactored version to make them disciplined. They found that undisciplined annotations are more time-consuming and error-prone. In contrast, Schulze et al. [95] found no differences between using disciplined and undisciplined annotations regarding task completion time and accuracy. Fenske et al. [39] conducted a survey study with 521 developers regarding annotations in the C language and found that their perception and their performance are different.

For instance, in Figure 1.5(a), we present a code snippet containing an undisciplined annotation. The annotation starts at Line 1 and wraps only the `if` statement in Lines 2 and 4 with their opening brackets but without the closing ones. Figure 1.5(b) presents the same code snippet but using a disciplined annotation. Liebig et al. [69] have shown that both kinds of annotations are present in a number of configurable systems. A number of refactorings has been proposed to change disciplined and undisciplined annotations [75; 45]. For instance, there is a refactoring [75] that allows us to convert the code snippet presented in Figure 1.5(a) to Figure 1.5(b).

```
1 #ifdef OPENSSL_SYS_VMS
2 if (access() != 0) {
3 #else
4 if (outdir != 0) {
5 #endif
6 // Lines of code here..
7 }
```

(a)

```
1 int test;
2 #ifdef OPENSSL_SYS_VMS
3 test = access() != 0;
4 #else
5 test = outdir != 0;
6 #endif
7 if (test){
8 // Lines of code here..
9 }
```

(b)

Figure 1.5: Code snippets adapted from *OpenSSL* with disciplined and undisciplined annotations.

Despite studies and discussions, it is difficult to reliably tell whether disciplined or undisciplined annotations improve code comprehension. For instance, both code snippets presented in Listing 1.5(a) and Listing 1.5(b) have almost the same values for code metrics such as Lines of Code (LOCs). Malaquias et al. [70] recommended avoiding undisciplined annotations because they are more time-consuming and error-prone. Schulze et al. [95] concluded

that there is no difference between disciplined and undisciplined annotations regarding time and accuracy. Fenske et al. [39] concluded that there is a difference between developers' perception and performance regarding annotations in C language.

Overall, in the research community, 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 state-of-the-art on understanding the impact of refactorings on code comprehension lacks conclusive results and qualitative/quantitative details on how and why of possible correlations. Our research problem is considered in the scope of the refactorings for atoms of confusion, Extract/Inline Method, and configurable systems in the context of novice programmers, with eye tracking.

## 1.2 Solution

In this work, we evaluate how different behavior-preserving code changes impact code comprehension of novice programmers from the perspective of controlled experiments. Our empirical research strategy uses a mixed-method, comprising three controlled experiments, interviews, and qualitative analysis, as seen in Figure 1.6. The usual approach to studying code comprehension has been the use of controlled experiments, in which one can measure the differences between treatments to sheds light on factors that affect comprehension [38]. Interviews and qualitative data analysis can be useful to support quantitative data and better understand the roles each factor can play in experiments involving human subjects.

Thus, our experiments consider distinct factors such as the human factor and different

granularities/complexities (atoms, Extract Method, configurable systems). In addition, it extends the usual coarse-grained dependent variables (time, accuracy, and opinions) with finer-grained ones (fixation time, fixation count, regressions count). Moreover, our analysis of the experiment results triangulates quantitative data with other obtained data from the interview and qualitative analysis.

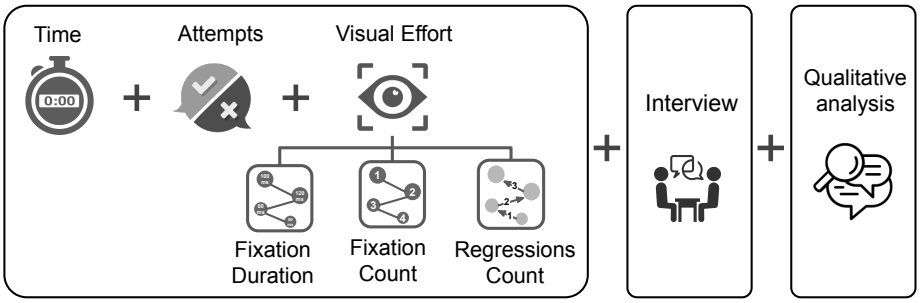



Figure 1.6: Overview of the methods employed in the controlled experiments.

In our controlled experiments, we use an eye tracking camera. An eye tracking camera is an equipment that tracks eye movements while someone looks at a stimuli [52]. It has been used in various fields, including source code reading and comprehension [29; 15; 116]. The data captured by the camera allow researchers to access where, when, and for how long a subject looks at a screen, and the eye tracking metrics evaluate how much subjects fixate and how they switch between distinct areas [82; 15; 99]. We can infer how much visual attention is given to specific elements on a screen along with the visual effort [29; 28; 8]. When the reader fixates more, and for a longer time, it can be associated with more attention to the stimuli [16], more time to understand code phrases [8], more attention to complicated code [28].

The visual effort combined with time and number of attempts can contribute to a better understanding of the novices' code comprehension. Eye tracking allows us to investigate 1) how much time the subjects spent in specific refactored code regions, i.e., line or lines of the code that contain an atom of confusion, an extraction of a method, a method call, a method inlining, or `#ifdef` annotations; 2) to what extent refactoring can impact visual metrics; and 3) how refactoring impacts the code reading with the visual regressions.

In our first experiment, we conduct a controlled experiment with eye tracking to evaluate the impact of six atoms of confusion on code comprehension. We evaluate to what extent

the obfuscated code containing atoms of confusion and the functionally equivalent clarified versions of the code impacted the time, the number of answer submissions, and visual effort.

In our second experiment, we evaluate the extent of the effects of Extract and Inline Method refactorings on code comprehension with eye tracking. Our definition of Extract Method takes a clump of code and turns it into its own method with a name that shows its intention, while preserving the behavior of the code, in the context of software evolution. We address code comprehension from the human subject perspective by triangulating the metrics of objective performance with subjective perceptions.

In our third experiment, we investigate how three fine-grained refactorings to discipline `#ifdef` annotations affect code comprehension. In particular, in addition to time and accuracy, we investigate the impact of these annotations on the visual effort by measuring the fixation duration, fixation count, and regressions count.

## 1.3 Evaluation

To investigate the impact of atoms of confusion, we report an eye tracking controlled experiment that evaluates how six atoms affect the code comprehension of 32 novices in Python. We consider as novices in Python undergraduate students who know how to program but have little experience with Python. We compare programs containing six atoms of confusion with functionally equivalent clarified versions aiming to observe how and to what extent they influence the performance of the subjects regarding time, number of attempts, and visual effort. We measure visual effort with three eye tracking metrics, namely, fixation duration, fixations count, and regressions count. Fixations and saccades-based metrics are among the most popular metrics employed in eye tracking studies and are easier to capture and understand. Other more elaborate metrics such as blinks or pupil dilation require more sophisticated equipment. Time and answer correctness have been employed before to measure effects on code comprehension [70; 95]. In addition, the visual effort has been measured before with fixation duration, fixations count, and regressions count [16; 8; 99]. We analyze these metrics in the whole code as well as in the main Area of Interest (AOI). The AOI defines the region of the code that contains 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.

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 both the entire code as well as in the AOI. We measure visual effort with fixation duration, fixations count, and regressions count. In the reading domain, fixations refer to the eyes focusing on certain spots when we read, while regressions refer to skipping back in the text to re-read a word or sentence. 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 of programs, one with a method inlined, and the other with the method extracted.

To investigate the impact of `#ifdef` annotations, we conduct a controlled experiment with 64 human subjects majoritarily novices. We consider all the subjects who know how to program but have little experience in C programming language "novices". The aim is to observe how disciplined annotations influence their performance on six tasks involving code comprehension in terms of time, accuracy, fixation duration, fixation count, and regressions count. Effects on code comprehension have been previously studied based on time and accuracy [70; 95]. Fixation duration, fixation count, and regressions count have been associated before with visual attention and effort in code comprehension scenario [16; 8; 99]. We measure these metrics in the code region in which both code versions differ after applying the refactorings, referred to as main AOI. For this study, we selected the three refactorings most preferred by developers to discipline annotations according to Medeiros et al. [75]. 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.

## 1.4 Conclusions

We found that the clarified version of the code containing the atom *Operator Precedence* reduced the time spent in the AOI and in the entire code to the extent of 38.6% and 20.1%, respectively. The metrics for the visual effort were also impacted to a considerable extent. The fixation duration in the AOI and fixations count was reduced as well. The most impacted metric was the regressions count in the AOI, with a reduction by 50%. These results can be associated with less visual effort. In addition, the number of attempts was reduced by 28.3%. On the other hand, the clarified version of the code containing *Multiple Variable Assignment* increased the time and the regressions count. We did not observe consistent reductions for the other atoms, however, they revealed other interesting nuances. We discuss more details in Chapter 3.

We found a reduction with the Extract Method for two tasks that varied from 70% to 78.8% in the time in the AOI, from 73.6% to 78.9% in the fixation duration, from 67.7% to 75.8% in the fixations count, and from 74.4% to 84.6% in the regressions count. For three tasks, the reduction varied from 20% to 34.4% in the number of attempts. These results might indicate more productivity and less visual effort. On the other hand, with the Extract method, there was an increase in the time in the AOI for three tasks that varied from 108.4% to 166.9%, from 73.1% to 130.1% in the fixation duration, from 137.1% to 194.2% in the fixations count, and from 100% to 200% in the regressions count. These results might indicate less productivity and more visual effort. In the interview, after coding the subjects' answers, we learned that extracting a method in the code favors the formulation of the hypotheses about the behavior of the code. We discuss more details in Chapter 4.

In our study, after applying *Wrapping Function Call* or *Alternative* `if` *Statements*, the total time spent in the AOI, fixation duration, fixation count, and regressions count were statistically significantly reduced. After applying *Alternative* `if` *Statements*, the number of incorrect attempts to solve the task, before submitting a correct answer, is reduced. However, no differences were observed in the number of attempts after applying the other refactorings. Even though for R2 we observed a statically significant increase in time in the AOI, it did not

result in statically significant differences in fixation duration, fixation count, and regressions count, therefore, the same amount of visual effort has been observed for this refactoring. Notably, our study setup reveals some nuances otherwise undetected by conventional code metrics. For instance, one of the refactorings adds one extra variable and two extra lines of code, which is only a small impact on the metric LOC, but it associated with reductions in AOI of 46.9% in the time, 44.7% in the fixation duration, 48.4% in the fixation count, and 60.5% in the regressions count. Overall, our results indicate that when a novice applies *Wrapping Function Call* or *Alternative `if` Statements*, she solves the task faster and with less visual effort. In addition, applying *Alternative `if` Statements* associated with improvements in the accuracy of her answers. We discuss more details in Chapter 5.

## 1.5 Summary of contributions

In summary, this work makes the following key contributions:

- We present a controlled experiment using eye tracking with novices in Python programming language to evaluate the impact of clarifying atoms of confusion on code comprehension (Chapter 3);
- We present a controlled experiment using eye tracking with novices in Java programming language to evaluate the impact of Extract Method on code comprehension (Chapter 4);
- We present a controlled experiment using eye tracking with novices in the C programming language to evaluate three refactorings that discipline `#ifdef` annotations in C programs (Chapter 5).

As a result of this work, we had the following contributions:

- An article [30] accepted for publication in the *Empirical Software Engineering* journal presenting the results of Chapter 5;
- We collaborated on an eye tracking study to investigate the effects of the presence of the atoms of confusion on time, accuracy, and focus of attention of developers, accepted for publication at *Brazilian Symposium on Software Engineering* [33];

- We collaborated in the investigation of the social representations of confusing code among two distinct communities of developers from the Brazilian software industry, accepted for publication at *International Workshop on Cooperative and Human Aspects of Software Engineering.* [32];

- An article submitted to the *Empirical Software Engineering* journal presenting the results of Chapter 3, from which we received a major revision, made the required changes, submitted again, and for the time being, is under revision;

- An article to submit to *IEEE Transactions on Software Engineering* journal presenting the results of Chapter 4, from which we received a major revision, made the required changes submitted again, and for the time being, is under revision;

- We worked together in the investigation of a Multivocal Literature Review on test smells submitted to *IEEE Transactions on Software Engineering* journal.

## 1.6 Organization

This dissertation is organized as follows: In Chapter 2, we present the background with concepts for understanding this dissertation. In Chapter 3, we present an eye tracking study with novices in Python programming language to evaluate the impact of atoms of confusion on code comprehension. In Chapter 4, we present an eye tracking study with novices in Java programming language to evaluate the Extract Method refactoring. In Chapter 5, we present an eye tracking study with novices in C programming language to evaluate three refactorings that discipline `#ifdef` annotations in C programs. In Chapter 6, we present the related work. And finally, in Chapter 7, we present the conclusions and future work.

# Chapter 2

# Background

In this chapter, we present the background to understand this work. In Section 2.1, we present an overview of refactoring code. In Section 2.2, we present an overview on the atoms of confusion. In Section 2.3, we present an overview of the configurable systems. In Section 2.4, we present an overview of code comprehension, and in Section 2.5, we present important concepts related to eye tracking approach.

## 2.1 Refactoring Code

We present the definition of refactoring in Section 2.1.1, examples in Section 2.1.2, and we describe how approaches evaluate the improvements in the design of the code in Section 2.1.3.

### 2.1.1 Definition

Refactoring can be understood as the process of changing a software system with the purpose of improving its internal structure without changing its external behaviour [40; 86]. It is characterized by code transformations with the purpose of improving the code design, making it easier to understand, evolve, and maintain. These transformations may involve several code changes such as renaming identifiers so that they could be more meaningful and easier to understand, and changing excessively long methods with Extract Method so that it becomes easier to figure out what the method does. Fowler [40] proposed a cata-

log of refactorings that became popular and commonly employed involving rename, Extract Method, move method, and others. Improving the code design involves making software easier to change and find bugs. This makes subsequent design iterations easier and makes the software more reusable [86].

### 2.1.2 Extract Method Refactoring

To extract a method, the developer creates a new method, and names it after the intention of the method. Then she copies the extracted code from the source method into the new target method. We have an example in Figure 2.1(a) converted to Figure 2.1(b), which exemplifies an Extract Method applied. In Figure 2.1(a), we have a code snippet that has the variables assignments, computes the smallest number of an array, and prints the output. However, the core that computes the smallest number of the array can be extracted with a descriptive name such as `getSmallestNumber` in Figure 2.1(b), which improves the code design. According to a prior study [107], developers extract a piece of code into a separate method to make the original method easier to understand, thus Figure 2.1(b) is better that Figure 2.1(a).

```java
public class Main {
    public static void main(String[] args) {
        int [] numbers = {2,1,3};
        int n = numbers[0];
        for(int i=1; i < numbers.length; i++){
            if(numbers[i] < n){
                n = numbers[i];
            }
        }
        System.out.println(n);
    }
}
```

(a)

```java
public class Main {
    static int getSmallestNumber(int[] numbers) {
        int n = numbers[0];
        for(int i=1; i < numbers.length; i++) {
            if(numbers[i] < n) {
                n = numbers[i];
            }
        }
        return n;
    }
    public static void main(String[] args) {
        int [] numbers = {2,1,3};
        int output = getSmallestNumber(numbers);
        System.out.println(output);
    }
}
```

(b)

Figure 2.1: Example of the Extract Method refactoring.

### 2.1.3 Improving the Design of Existing Code

Several factors are involved in the quality of the design of the code. These factors can be directly measured, in terms of internal quality attributes or indirectly measured, in terms

of external quality attributes. Internal quality attributes are measured through code metrics such as the number of LOCs, coupling, and cohesion, while understandability exemplifies an external quality attribute.

In a previous study [20], the effects of refactorings to improve software structural quality have been evaluated using software metrics. In using metrics, a study revealed that either neutral or negative effects of software refactoring are much more frequent than positive effects [19]. Software metrics provide a means to extract quantitative measurable information about the structure of the code and some popular metrics used to measure the observable effects of refactorings on code structure are the number of LOCs [22], McCabe's Cyclomatic Complexity [71], and coupling metrics [62]. However, using these conventional metrics to evaluate the effects of fine-grained refactorings on the code may be challenging. There are not always observable differences in these metrics after applying small transformations [19], such as adding one extra line of code or adding new fresh variables.

For instance, consider the Figure 2.1(a) compared to Figure 2.1(b). Both versions present the same values for Cyclomatic Complexity and coupling. However, the refactored version in Figure 2.1(b) adds four extra LOCs and one extra variable. In this snippet, refactoring could negatively affect the time it takes a developer to read the code with the addition of more lines. On the other hand, it has the advantage of code reuse since the new method can be used in other parts of the code. In addition, with a descriptive name, it can improve one's capacity to understand the code. In such case, the sole use of code metrics cannot reveal the nuances involved in the external quality attributes, such as the extent of the impact of the refactoring on code comprehension.

To capture additional dimensions, there are also qualitative studies that use surveys to investigate the developers' perception regarding the effects of refactorings on the code [85; 63; 75]. The quantitative and qualitative approaches to evaluate refactorings have been used in the context of single programs and in configurable systems, where we have several configuration options. In the next section, we discuss this topic in more detail.

## 2.2 Atoms of Confusion

When writing code, developers communicate their intent to other developers. The correct interpretation of their intent is crucial for the software maintenance and evolution processes. This interpretation occurs through a process of code comprehension, in which a developer reads the source code, often written by another developer, and understands its behavior. However, the developer's interpretation of a piece of code can often differ from that of the one who wrote the code due to tiny patterns that can cause misunderstandings. These tiny patterns that can obfuscate the code and confuse developers causing them to misjudge the code's behavior are called atoms of confusion [48; 49; 66] when it has been experimentally shown there are functionally equivalent alternatives that lead to better performance. Castor gives a more complete definition of atoms of confusion, mentioning four characteristics needed for a pattern to be considered as an atom of confusion [18]: 1) precisely identifiable; 2) likely to cause confusion; 3) capable of being exchanged for another less confusing pattern that does the same thing; 4) not divisible.

Gopstein et al. [48] cataloged a set of atoms of confusion as a result of empirical studies to validate atom candidates. The candidates were extracted from the International Obfuscated C Code Contest, a list of 19 potential atoms. The authors tested how well subjects could hand trace those atoms compared to functionally equivalent code snippets with the obfuscations removed. From the initial set of candidates, 15 were confirmed to be significantly confusing to our subjects. We have some examples in Table 2.1.

While many of these atoms have been shown to confuse developers, a prior study has shown that they are prevalent in real-world software [49]. The study mined several popular open-source C and C++ projects looking for instances of the 15 atoms of confusion cataloged. Atoms of confusion are prevalent in real projects. There are over 3.6 million atoms in 14 popular open-source C/C++ projects, appearing on average once every 23 lines. These projects include *git*, *vim*, *Linux*, among others.

| **Atoms** | **Description** | **Obfuscated** | **Clarified** |
|---|---|---|---|
| Multiple Variable Assignment | We can assign the same value to multiple variables by using = consecutively | `a = b = 1` | `b = 1`<br>`a = b` |
| True or False Evaluation | Directly checking whether something is `True` versus checking whether its negation is `False` | `(not a == b)` | `(a != b)` |
| Conditional Expression | Also called "ternary operator", an `if` statement can be written in one line with conditional expression | `a = b if b == c else d` | `if (b == c):`<br>`  a = b`<br>`else:`<br>`  a = d` |
| Operator Precedence | The precedence of the operators influences the outcome. Depending on the order, we might have different results | `a and b or c` | `(a and b) or c` |
| Implicit Predicate | An expression that does not produce a `bool` is used as a predicate | `(a % b)` | `(a % b != 0)` |
| Augmented Operator | A single operator adds a value to a variable and updates it | `a += 1` | `a = a + 1` |
| Preprocessor in Statement | Preprocessor directives must stand alone on their own line | `int V1 = 1`<br>`#define M1 1`<br>`+1;` | `#define M1 1`<br>`int V1 = 1 + 1;` |
| Post-Increment/Decrement | Confusion arises because the value of the expression is different from the resultant value of the variable | `V1 = V2++;` | `V1 = V2; V2 += 1;` |

Table 2.1: Atoms of confusion.

## 2.3 Configurable Systems

Costumers are often demanding customized software products with more diverse features. On the other hand, developers are under the pressure of time and cost constraints to meet such demands. In this scenario, the Software Product Line (SPL) engineering approach becomes helpful for improving the software development process. This approach presents the advantage of reusing existing software components to reduce the cost and time while increasing its diversification [23; 89; 117].

This idea of reuse in software products emerged from the automobile industry [89]. Overall, customers were content with standard cars. However, not all wanted the same kind of car for any purpose. For instance, some people drive alone; others have big families. Some people drive in the city, others in the countryside. They had distinct needs and demanded individualized products tailored to their own needs. Thus, companies started to introduce common platforms for their different types of cars by planning beforehand which components would be used in different car types. They could provide customized cars with distinct features by reusing common components at reasonable costs.

Similarly, in the context of software development, a key concept in SPL engineering consists of variability [117]. Variability can be considered as the ability to change or customize a system [118]. Individual systems are considered variations of a common theme and must be made explicit and systematically managed. Thus, a configuration is a process of binding the optional features of a system to produce a specific software system or variation [27]. We can configure the system by selecting and deselecting features and producing a new variation. Therefore, as a result of this process, the systems can be configured to tailor to the users' needs.

The differences between variant products can be described in terms of their features. A feature is a user-visible characteristic of a software system [24]. It can also be understood as a way to abstract from customers' requirements; thus, one particular feature may meet more than one requirement, and one requirement may apply to several features [118]. In practice, implementing variability can pose challenges to the developers for adding complexity. Given that the growth of variations as a function of the number of features is exponential, ensuring that all product variants meet given properties is a non-trivial issue [75].

In practice, developers mostly implement variability using a preprocessor. A preprocessor is a system that processes the input data and produces an output used as an input to the compiler [1]. Before the compilation, part of the compiler performs a set of operations over the source code, which is the preprocessing. These operations may include dealing with macros, which are abbreviations of longer constructions, dealing with the inclusion of files, and language extensions [1]. The C Preprocessor (CPP) is the most common example of a preprocessor, which we discuss in more detail in the next section.

### 2.3.1 The C Preprocessor

The CPP is widely used in practice to handle portability and variability in the C program families [109]. A family of programs can be understood as individual programs with extensive characteristics in common such as variants of the same program [87]. The CPP is used to implement the individual variants in several open-source projects such as *Linux* operating system, *Apache* web server, and *Dia* drawing software.

Some of the reasons for such popularity among developers can be explained because the CPP is simple and flexible. In it, developers annotate the code with simple and intuitive preprocessor directives which manipulate the source code before compilation occurs. These directives include file inclusion (`#include`), macro definition (`#define`), macro substitution and conditional compilation (`#if`, `#ifdef`, `#ifndef`, `#elif`, `#else` or `#endif`) [37]. In this study, we focus on the conditional compilation.

```
1  int main() {
2  #ifdef M1
3     int x;
4  #endif
5
6  #ifdef M2
7     int y;
8  #endif
9
10 #ifdef M3
11    int z;
12 #endif
13 }
```

Listing 2.1: Code snippet before preprocessing with configurations.

```
1  int main() {
2
3     int x;
4
5
6
7     int y;
8
9
10
11    int z;
12
13 }
```

Listing 2.2: Code snippet after preprocessing with all configurations enabled.

```
1  int main() {
2
3     int x;
4
5
6
7     int y;
8
9
10
11
12
13 }
```

Listing 2.3: Code processed with `M1` and `M2` enabled and `M3` disabled.

Conditional compilation annotations allow the developers to work with one source code, but they can define separate code branches, which are included or excluded from the final compilation depending on the value of conditions evaluated by the preprocessor [44]. The annotations are not part of the C language and they need to be evaluated and removed after the preprocessing. For instance, consider the Listing 2.1 as an example of a configurable system with `#ifdef` annotations [13]. In Listing 2.1, we present the source code before preprocessing with three macros, `M1`, `M2`, and `M3`. Each macro is a condition to be evaluated and can be included if enabled and excluded if disabled. In Listing 2.2, all the variables are declared which means that all the macros were enabled. In Listing 2.3, variable `z` was not declared since `M3` was disabled.

### 2.3.2 Undisciplined Annotations

Despite the benefits of CPP to deal with portability and variability, its usage has been criticized in academia and has been even considered harmful [109]. Studies observe that the use of CPP can cause problems such as the occurrence of syntactic and semantic errors during the generation of variant products [65; 61], the pollution of the code

due to scattered and tangled `#ifdef` annotations also known as "`#ifdef` hell" [109; 75], and the decrease in maintainability and in ability to evolve the system [36; 34].

In particular, the use of undisciplined annotations has the potential to aggravate these problems. Undisciplined annotations are preprocessor directives that encompass only parts of the statements [45], which leads to problems such as annotating an opening bracket without the closing one [34]. We define disciplined and undisciplined annotations based on the following definition: disciplined annotations are annotations on certain syntactic code structures, such as entire functions and statements, whereas undisciplined annotations are annotations of individual tokens or brackets that do not align with underlying code structure [69]. Thus, we consider undisciplined annotations as annotations that do not wrap entire statements. For instance, consider the Listing 2.4. The `#ifdef` encompasses the `if` statement and the opening bracket but does not encompass the closing one. In this sense, the areas of the code with this type of annotation appear as bad smells, which are candidates for a bad design.

```
1 #ifdef OPENSSL_SYS_VMS
2 if (access() != 0) {
3 #else
4 if (outdir != 0) {
5 #endif
6 // Lines of code here..
7 }
```

Listing 2.4: Example of undisciplined annotations wrapping parts of statements.

Consider Listing 2.5. In this example, the preprocessor usage occurs within single statements. A single statement can be understood as a statement that contains no compound blocks. Some examples include variable initializations, function calls, and return statements. In the example, the annotations wrap only parts of the function call.

```
1 mfp = open(mf_fname
2 #ifdef UNIX
3   , 600
4 #else
5   , IWRITE
6 #endif
7 );
```

Listing 2.5: Example of undisciplined annotations in single statements.

A prior study performed an empirical study on 41 C program families and found that almost 90% of syntax errors occur in undisciplined annotations [74]. Thus, maintaining and evolving a system in the presence of conditional compilation plus using undisciplined annotations can be a challenge for the developers. Undisciplined annotations have shown a negative influence on comprehension and maintenance of annotated code in a controlled experiment [70]. Since reuse is at the core of configurable systems, the design of the code has to be improved in such a way that the code is easy to read, maintain, and evolve. In the next section, we give an overview of refactoring undisciplined annotations.

### 2.3.3 Refactoring Undisciplined Annotations

Studies have proposed catalogs of fine-grained refactorings to convert from undisciplined annotations to disciplined ones [45; 75]. They are fine-grained in the sense that they involve simple and small code transformations. For instance, to remove the undisciplined annotations presented in Listing 2.4, Medeiros et al. [75] proposed a fine-grained refactoring that uses an alternative `if` statement and defines a fresh variable `test` that receives the evaluation of *COND_1* or *COND_2* depending on whether macro *EXP* is defined or not. It adds two extra lines of code to remove the undisciplined annotations. The template of the proposed refactoring can be seen in Listing 2.6 and Listing 2.7. This refactoring can be applied whenever the left-hand side of the template matches a piece of C code. In this context, a matching consists of an assignment of all meta-variables highlighted in capital letters to concrete values in the C code.

```
1 #ifdef EXP
2 if (COND_1) {
3 #else
4 if (COND_2) {
5 #endif
6   STMTS
7 }
```

Listing 2.6: Code with undisciplined annotations before refactoring.

```
1 int test;
2 #ifdef EXP
3 test = COND_1;
4 #else
5 test = COND_2;
6 #endif
7 if (test) {
8   STMTS
9 }
```

Listing 2.7: After refactoring, the annotations in the code of Listing 2.5 becomes disciplined.

## 2.4 Code Comprehension

Understanding code is a crucial activity in maintaining and evolving any system. On average, a maintenance developer spends 50—60% of her time with understanding source code [110]. From the code comprehension perspective, studies have argued that the preprocessor usage leads to more complicated code which is considered difficult to understand [37; 65; 109; 34]. Since the preprocessor directives are usually intermingled in the code, the developers can face difficulties in determining the scope of each conditional definition [42]. Reading and understanding source code are difficult because every time someone enters a conditional compilation section she has to stop and determine under what circumstances that code will be selected for compilation [88].

Code comprehension can be described by two main models, namely top-down and bottom-up processes. Other models may comprise a mix of these two models. In the top-down model, developers try to comprehend the code by first formulating hypotheses about the purpose of the code. Then, the developers examine the details and refine the hypothesis by developing other hypotheses. These other hypotheses are refined further until developers have low-level comprehension of the code [119]. In this refinement process, developers use beacons which are key defined as lines, fragments, or elements that help them [120]. The bottom-up model is used by developers who have insufficient knowledge about the code, so they cannot use beacons. Thus, they start to understand the code by examining first its

details, such as the singular statements or control constructs [105]. Statements that semantically belong together are grouped into chunks of higher-level abstractions. These chunks are integrated into larger chunks in a process that repeats until the developer has the comprehension of the whole code. Developers can also use both models when trying to comprehend the code. They use a more top-down comprehension process when they are familiar with the application domain and bottom-up comprehension when unfamiliar with it [96].

Investigating code comprehension is not a trivial task given the number of factors that have to be considered, in particular, in controlled experiments. Numerous studies have been conducted to investigate how developers understand source code, and controlled experiments are central to these investigations [105]. In these experiments, human subjects are asked to solve a programming task on a given source code. These tasks are designed so that, to solve it, the developers have first to understand the code. Thus, the effort and success in performing the task are a proxy to estimate the difficulties in code comprehension [38]. Therefore, the code, task, metrics, and human subjects are factors that play important roles in these studies, and they have to be carefully considered [38; 14].

To measure code comprehension, studies involving human subjects mainly measure the correctness or accuracy of the subjects' answers. Particularly, subjects are asked to provide information about the code, for instance, predict the output of the code, identify code elements, or explain a high-level functionality. They can also be asked to act on the code, such as modifying it. This metric relates to an objective perspective of the performance of the developers, which contrasts with a more subjective perception that comprises personal opinions and preferences [39]. Usually, studies investigate the preferences of the developers for a certain code snippet over another, rate their comprehension, rate their confidence in their answer, and their difficulties [84]. In addition to these two metrics, studies also investigate how much time is needed to complete the task or read the code. These metrics have not been used exclusively. Indeed, studies have combined them to investigate code comprehension.

More recently, new approaches to measuring code comprehension have gained attention. They can provide additional indicators of the visual and mental effort of the developers while solving code tasks. These approaches mainly include metrics related to eye and brain activities as indicators of difficulties in performing tasks [41; 38; 105]. For instance, functional magnetic resonance imaging (fMRI) allowed researchers to observe which brain regions got

activated when subjects performed defined tasks, such as understanding source code, and associated these different regions with different cognitive processes [106]. In the literature, visual attention can trigger cognitive processes that are required to perform tasks [60]. In addition, it is also a proxy for visual effort, a subset of cognitive effort, measured as the amount of visual attention allocated to parts of a visual stimulus [99]. Eye tracking metrics have been used to investigate code comprehension through visual effort. Given the scope of this work, in the next section, we discuss in more detail the particularities of the eye tracking approach.

## 2.5 Eye Tracking

Eye tracking involves assessing a subject's visual attention by recording eye movements [90]. The eye movements show where a subject is looking, the duration, and the eye gaze sequence in which her visual attention switches from one location to another [99]. Eye movements are essential to cognitive processes. The eye movements focus the subject's visual attention on the parts of a visual stimulus that are processed by the brain, triggering the cognitive processes that are required to perform tasks [60].

Visual attention works as a proxy for visual effort, which is a subset of cognitive effort, measured as the amount of visual attention allocated to parts of a visual stimulus [99]. Researchers can use eye trackers how a certain stimulus can impact the subjects' visual attention. In the code comprehension context, for instance, a visual stimulus could be a piece of code that the developer has to solve. While focusing her visual attention on specific parts of the code, and putting effort to understand it, the necessary cognitive processes are triggered, for instance, thinking about what it does and how, remembering identifiers and assigned values, or judging the behavior of the code. Thus, an eye tracker allows us to measure the visual effort involved in comprehending the code.

The terminology of eye tracking includes four main categories of metrics [90]:

- **Fixation**: the eye remains stable focusing on part of a stimulus for a period of time (Figure 2.2(a)). It is based on the idea that, as soon as a subject sees a word or a text in the code, she tries to interpret it, and fixates her attention on it until she comprehends it [60];

- **Saccade**: continuous and rapid eye movements occurring when the subject switches from one fixation to another (Figure 2.2(b));
- **Pupil dilation**: when a pupil dilates, it allows more light to get into the eye in low light conditions (Figure 2.2(c)). It can also be affected by the cognitive effort required by the stimulus;
- **Scanpath**: a series of fixations in chronological order that represents the switches in the eye movements (Figure 2.2(d)).

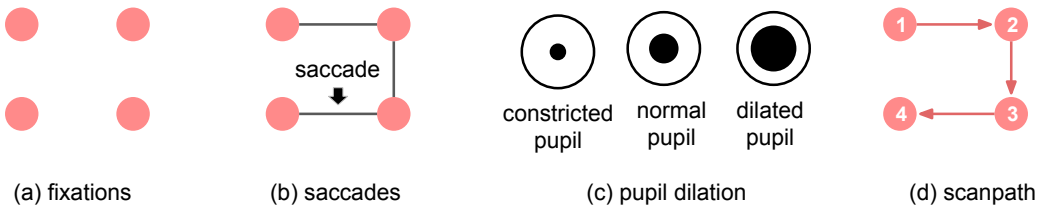



Figure 2.2: Eye tracking terminology.

The research community investigates eye movements with respect to specific areas in the visual stimulus called Areas of Interest (AOI). For instance, lines of code with the annotations as depicted in Figure 2.3. It could also include specific constructs, the body of a certain method, and an area of the code that contains a bug. These areas can be relevant to the researcher to formulate or refine hypotheses depending on how developers interact with them. For instance, observing whether converting from undisciplined to disciplined annotations impacts the metrics in the AOI, or renaming a method to observe whether it alleviates or increases the visual effort of the developer in that area. The AOI could cover the whole method extracted or a region where the candidate atom, which can allow us to observe how the fixations in the area can be impacted.

### 2.5.1 Metrics

We can divide eye tracking metrics according to the four mentioned categories: metrics based on fixations, metrics based on saccades, metrics based on scanpaths, and metrics of pupil size and blink rate [99]. With respect to metrics based on fixations, we commonly find the following metrics:

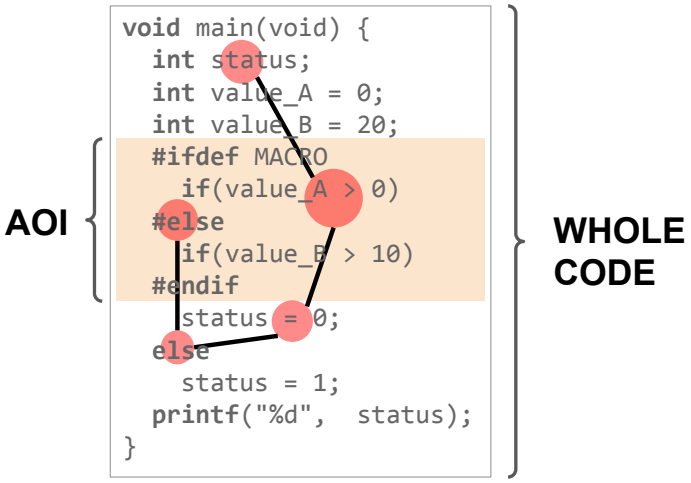


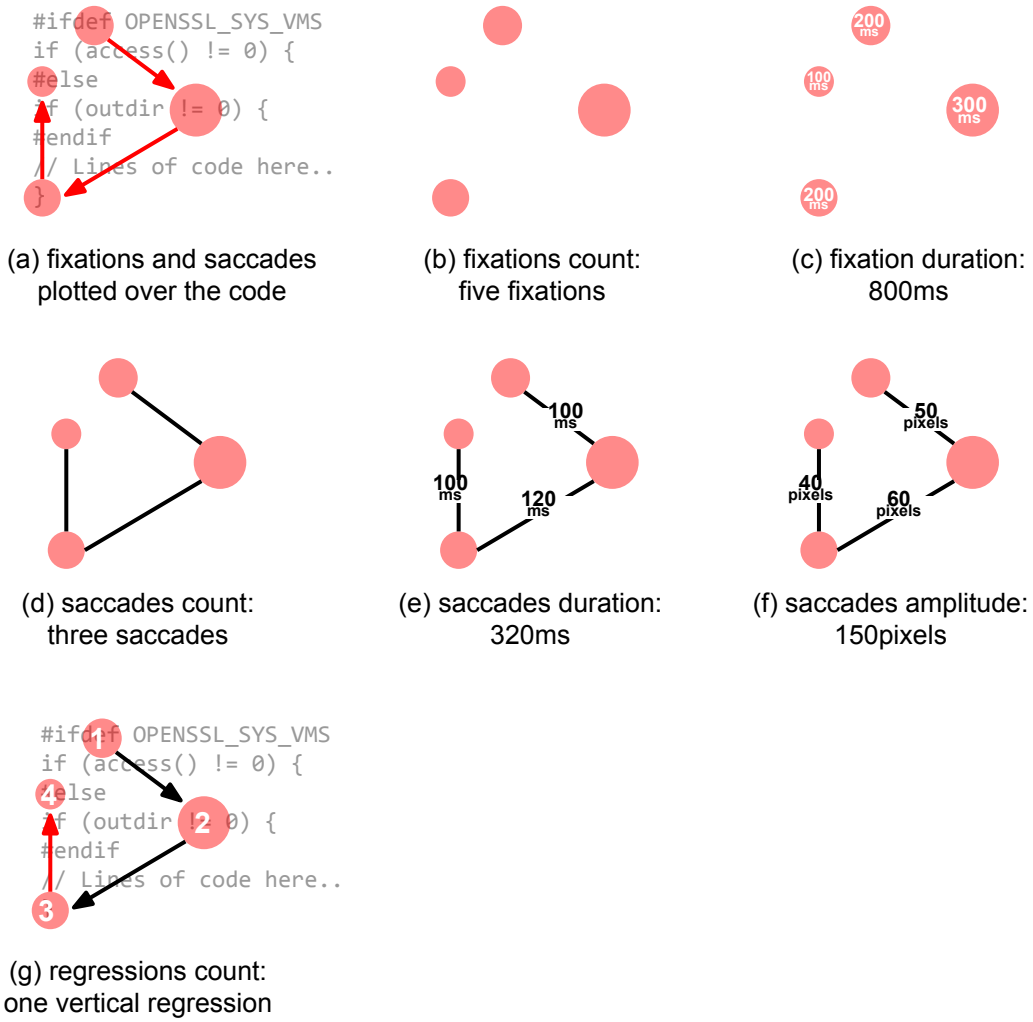

Figure 2.3: Example of AOI in a program.

- **Fixations count**: the total number of fixations in the AOI or in the whole stimulus (see Figure 2.4(b));
- **Fixation duration**: sum of the duration of all the fixations on an AOI or the stimulus (see Figure 2.4(c));

Fixation count and fixation duration have been used before to find the AOIs that attract more attention [29], including studies in the software engineering field [99]. A higher number of fixations devoted to a stimulus, such as an image on the computer screen, may indicate that the search for relevant information is not efficient [46]. In comprehension tasks, for instance, a higher fixation rate on a specific AOI may indicate that the subject shows a great interest in its content or it may also indicate effort or difficulties in understanding [57]. To distinguish between interest and difficulties, researchers usually perform a triangulation with qualitative interviews to better understand the motivations.

With respect to metrics based on saccades, we can mention the commonly employed following metrics:

- **Saccades count**: the total number of saccades in the AOI or in the whole stimulus (see Figure 2.4(d));
- **Saccades duration**: the duration of the saccade consists of the time one spends while not fixating; she is gazing between two fixations (see Figure 2.4(e));




(a) fixations and saccades plotted over the code

(b) fixations count: five fixations

(c) fixation duration: 800ms

(d) saccades count: three saccades

(e) saccades duration: 320ms

(f) saccades amplitude: 150pixels

(g) regressions count: one vertical regression

Figure 2.4: Example of eye tracking metrics based on fixations, saccades, and scanpath.

- **Saccades amplitude**: relates to the distance in pixels between two fixations (see Figure 2.4(f));

- **Regressions count**: the number of backward saccades of any length over the stimulus [15] (see Figure 2.4(g)).

Overall, more saccades can be associated with more visual searching [46], thus, a certain stimulus that requires more saccades to be analyzed can be associated with a less efficient search behavior. Besides, while examining the text, careful inspection of the stimulus resulted in shorter saccadic amplitudes [114]. Difficulties in reading texts can be inferred by the saccadic regression behavior. For instance, studies reported that as the text becomes conceptually more difficult, the frequency of regressions increases as well [91]. In the field of code comprehension, regressions may be an indicator of measuring visual effort [99].

With respect to metrics based on scanpaths, we can mention the commonly employed following metrics:

- **Attention Switching Frequency**: when we divide the stimulus, such as an image, into several areas of interest, we can compute the number of times one switches or jumps from one area to another;

- **Edit Distance**: computes the minimum editing cost of transforming one scanpath into another with the operations insertion, deletion, and substitution using the Levenshtein algorithm;

- **ScanMatch**: computes the similarity between two scanpaths;

- **Linearity**: describes how linear (top-to-bottom and left-to-right) is the searching strategy employed by the subject.

We can compare scanpaths to find patterns and study the subjects' visual strategies while solving the task [15]. The edit distance has been used to indicate the similarity among subjects' viewing or reading strategies [98]. In addition, for source code, the linearity of the scanpath has been evaluated with respect to how similar subjects read code to text and how similar they read to execution order following the code control flow [15].

With respect to metrics based on pupil dilation and eye blinks, we can mention the commonly employed following metrics:

- **Blink rate**: the rate of eye blinks while the subject is looking at the AOI or at the whole stimulus;
- **Pupil size**: the physical diameter of the pupil is usually given in millimeters.

The blink rate correlates with visual demand. When more attention is needed, eye blinks are inhibited and delayed to a moment when the demand is reduced [112; 31]. Lower blink rates have been associated with higher workload or attention [41]. In addition, increased effort and heavier cognitive workloads given task difficulty are related to larger pupil sizes [6].

### 2.5.2 Visualization Techniques

Eye tracking experiments can generate a large amount of data in terms of fixations and saccades which increases according to the number of subjects. All these data can be analyzed by statistical methods with the support of visualization techniques. While statistical analysis provides a quantitative perspective of the results, visualization techniques allow researchers to analyze different levels and aspects of the data in an exploratory and qualitative manner. This manner comprises an analysis of temporal evolution of the position of data points, distribution of attention, scanpath analysis, or spatio-temporal structure of eye tracking data [10].

Since eye tracking studies cover a wide range of distinct areas, we concentrate on presenting the visualizations most employed in studies that relate to code comprehension and their aspects. In this context, we can mention gaze plots and heatmaps.

**Gaze plots:** provide a static view of the eye-gaze data and show the time sequence of looking using the locations, orders, and duration of fixations on stimuli [100]. Sharafi et al. [100] provides two examples of gaze plots which can be seen in Figure 2.5, which present fixations in red circles. In Figure 2.5, the size of circles remains the same but the size of the circle can increase according to the duration of the fixations. Longer fixations are represented by larger circles.

**Heatmaps:** Heatmaps can be defined as two-dimensional graphical representations of data where the values of a variable are shown as colors and are used to determine the amount of interest generated by various elements of the stimulus, thus, distribution of visual attention [11]. They represent the intensity of a measure, for instance, fixations, through colors [100]. Their usage is compelling because the interpretation is intuitive, since the amount

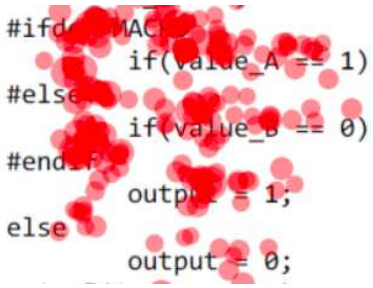


Figure 2.5: Visualization of fixations. The size of fixations varies according to their duration.

of heat is proportional to the level of the represented variable, and because heatmaps show the data directly over the stimulus, so they are easy to read [11]. They can be used for only one subject or aggregate several subjects. For instance, in Figure 2.6, we present an example of a heatmap. It is based on 16 novices aggregated who solved the same task before applying a refactoring. In Figure 2.6, the subjects solve the task before the refactoring is applied.In code comprehension studies, they have been used to show the distribution of attention over the stimuli.

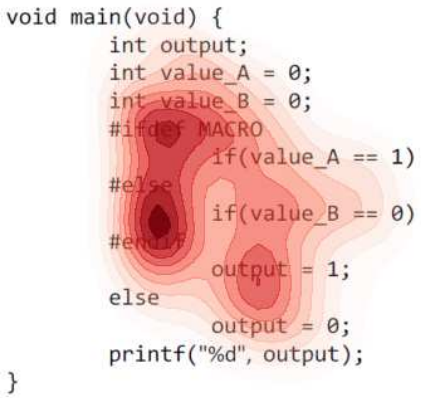



Figure 2.6: Visualization of the distribution of attention on a program.

**Visualizing saccades:** The eye movement from one fixation to another is called a *saccade* [93; 90]. We can map the chronological sequence of saccades between the code lines to visualize and better understand the dynamics of eye movements between distinct elements of code before and after applying the refactoring. A graph can help us visualize such dynamics with a node representing a line of code and an edge representing a saccade such as in Figure 2.7. 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 the weight. The more intense the color, the heavier the weight, which implies the more frequently they made the same saccade.

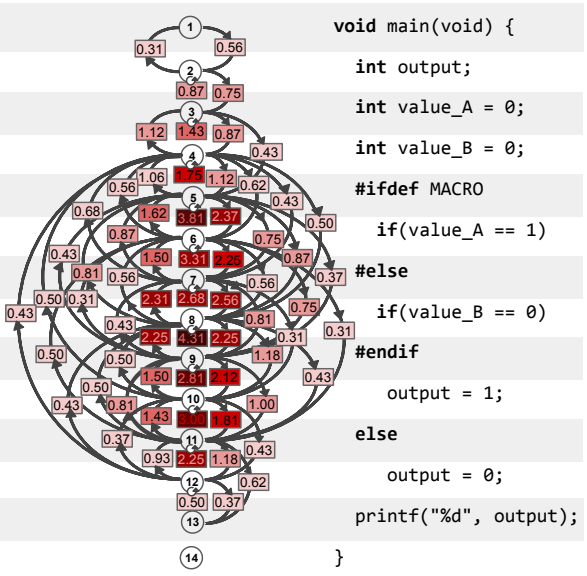



Figure 2.7: Visualization of saccades in a graph on a program.

# Chapter 3

# Study I: Refactoring Atoms of Confusion

In this chapter, we compare programs obfuscated by the atoms with functionally equivalent clarified versions to investigate the impact of clarified programs on code comprehension. This chapter is organized as follows: Section 3.1 presents the study definition, and Section 3.2 presents the study methodology. Section 3.3 presents the obtained results and discussion. Section 3.4 discusses the threats to validity, and finally, Section 3.5 presents the conclusion.

The transformations applied to the obfuscated code are structural transformations aiming to make them clearer to understand while preserving its behavior. We consider those code clarifications as refactorings in the sense that they can be characterized as behavior-preserving code transformations aiming to improve the code in certain aspects [40], such as code comprehension. However, instead of focusing on the transformation process, we focus on the results of the transformations. Thus, in this chapter, when we refer to the clarified code version, we have in mind a refactored version aiming to clarify the code.

## 3.1 Study Definition

In this section, we present the definition of our study following the Goal-Question-Metrics approach [5]. **We compare** programs containing obfuscating atoms with functionally equivalent clarified versions of these programs **for the purpose of** understanding how the clarified programs associate with improvements **with respect to** code comprehension **from the point of view of** novices in Python programming language **in the context of** tasks adapted from

introductory programming courses.

We address five research questions (RQs). Our null hypothesis for each RQ is that there is no difference between the obfuscated and clarified programs with respect to the collected metric.

- **$RQ_1$: To what extent do the atoms affect task completion time?**

  To answer this question, following prior studies [48; 33], we measure how much time it takes for the subject to specify the correct output of the task. In addition, we measure how much time the subject spends in specific areas of the code.

- **$RQ_2$: To what extent do the atoms affect task the number of attempts?**

  To answer this question, also following prior studies [48; 33], we measure the number of attempts by counting the number of attempts made by the subject until answering the task correctly.

- **$RQ_3$: To what extent do the atoms affect fixation duration?**

  To answer this question, we measure the duration of each fixation found in the captured data of the novices. In the code comprehension scenario, fixations with high duration have been associated with an increased level of attention [16].

- **$RQ_4$: To what extent do the atoms affect fixations count?**

  To answer this question, we count all fixations found in the captured data of the novices. A high number of fixations has been associated with a longer time to process and understand code phrases [8], more attention to complex code [28], and more visual effort to recall identifiers' names [101].

- **$RQ_5$: To what extent do the atoms affect regressions count?**

  Regressions in the context of natural language reading may indicate that the reader did not understand what they read [91]. In the programming context, regressions have been used to assess the linearity of code reading [15]. In an imperative programming language, text lines may be read left-to-right, top-to-bottom, similarly to natural language. However, there are constructs, such as loops, that require the reader to read

bottom-to-top at some points. To make the comparison fair, both obfuscated and clarified versions of the code have loops that iterate over the same number of elements. Thus, to measure regressions, we compute the number of saccades with a direction opposed to the writing system, which can happen from a line of code to a previous one, or within the same line. We compute the number summing all regressions across all attempts. To compare the programs, we compute the median number of regressions on each program.

## 3.2 Methodology

In this section, we present the methodology of our study. We present the pilot study (Section 3.2.1), experiment phases (Section 3.2.2), subjects (Section 3.2.3), treatments (Section 3.2.4), evaluated atoms of confusion (Section 3.2.5), programs (Section 3.2.6), eye tracking system (Section 3.2.7), fixation and saccades instrumentation (Section 3.2.8), and finally the analysis (Section 3.2.9).

### 3.2.1 Pilot Study

Before conducting the actual experiment, we conducted pilot studies with five human subjects. The purpose was to refine the material, such as forms and programs, and evaluate the experiment setup and design. We do not consider these five subjects in the analysis of the results.

Our study material comprises a set of programs, a form for characterizing the subjects, and questions for a semi-structured interview. To evaluate our set of programs, we tested complete code snippets from introductory programming courses. We validated the level of difficulty of the programs, code font size, font style, spaces between the lines of code, and indentation. In addition, we estimated the average time of each task, which allowed us to set a proper time limit for them. We found that each of our programs usually took less than two minutes to be solved. We also refined the questions from the forms.

Since our subjects are native Brazilians, we designed our programs and the vocabulary to be in the Brazilian Portuguese language, thus, avoiding problems in code comprehension given the lack of knowledge of words in English, for instance. We evaluated a limited vocab-

ulary of words to name the variables in the programs. The identifiers were carefully selected, discussed by the researchers, and designed to convey some but not all of the information. For instance, we used words such as `elements` and `items` which are general terms for lists of elements; `value`, `result`, and `total` for receiving the result of operations and printing the output; in addition, we used two abbreviated words such as `elem` and `cont` to specify an element and a counter, which are commonly employed in the context of teaching introductory programming languages. In specific programs, we also used words such as `grade`, `bonus`, `average`, and `final` to convey meaning in the specific context.

While previous empirical studies used variables with meaningless single-letter names [48; 47; 66], we opted for names that conveyed some but not all the information. Meaningless names can make the code intrinsically harder to understand and, therefore, differences are likely to be accentuated. However, in real code reading tasks, developers are usually, though not always, faced with variable names that use real words and have a meaning for them [67]. Therefore, following this approach is arguably closer to a practical scenario.

Through the conduction of the pilot studies, we learned that, when studying in the context of novices, we should provide the tasks in their mother tongue, otherwise comprehension would be hampered by natural language barriers. In addition, we should evaluate tasks with different levels of difficulty, which allows us to have a better set of tasks. Finally, we should ask the subjects questions about the programs, identifiers' names, and other suggestions to refine the tasks.

The pilot studies allowed us to evaluate and refine our experiment design, which consists of four phases: (1) Tutorial, (2) Warm-up, (3) Task, and (4) Qualitative Interview. We then estimated an average of around 60 minutes for each subject to complete all phases. Next, we describe these phases in detail.

### 3.2.2 Experiment Phases

As the subject enters the room, we introduce ourselves and explain the main idea of the study, what data we are going to capture and for what reasons. We asked each subject to fill out a consent form, agreeing to participate in the study and giving the researchers permission to use the data for academic purposes only. The access to the collected data was restricted

to the researchers and the identity of the subjects was kept in anonymity. In addition, the subjects filled another form with questions related to programming background experience for the characterization purpose.

In phase one, we present a tutorial with explanations regarding the execution of the experiment. All subjects reported being familiar with Python language, thus, we present some snippets to make sure they were familiar with them. We instruct the subjects on how to sit properly on the chair in front of the camera and how to perform the task. In addition, we explain that the subject can quit at any time and does not need to provide any reasons for doing that. Once the subject is seated comfortably in front of the camera, we explain how the camera calibration process works and we proceed with a calibration of the camera on each subject's eye. In the camera calibration process, the subject looks at specific locations on the screen indicated by the camera software, and the same software indicates when calibration is successfully done. For some subjects, we had to re-calibrate the camera until we gained confidence that the data captured by the camera was reliable.

In phase two, we simulate the execution of the experiment with a simple warm-up task. While solving the task, we demonstrate how the subjects can specify its output, how the subject can close their eyes for two seconds before and after solving the task, how we signal the correct and incorrect answer, and how we signal the time limit. The idea is that the subject can be comfortable with the experiment setup and equipment.

In phase three, we run the actual experiment with twelve programs, six of them containing a distinct obfuscating atom each, and six functionally equivalent clarified programs. To avoid learning effects, we use a Latin Square design [12] for the experiment. We explain this in more detail in Section 3.2.4.

In phase four, we end the experiment with a semi-structured interview. The goal is to obtain qualitative feedback on how the subjects examined the programs along with their subjective impressions. We go through each of the twelve programs and ask three questions: (1) How difficult was it to find the output: very easy, easy, neutral, difficult, or very difficult? (2) How did you find the output? (3) What were the difficulties you had, if any?

The coronavirus pandemic made the running of eye tracking experiments more challenging. It is worth mentioning that the health and safety of our subjects are of utmost importance to us. We started running the experiment only after the end of the social distancing measures

in the country and during a time when the infections were decreasing, and the number of vaccinated people was increasing. All subjects had at least one dose of a COVID-19 vaccine. Still, we arranged an environment with fresh air and all subjects had Personal Protective Equipment (PPE) and disinfecting supplies such as hand sanitizers and face masks. Since we had one subject at a time, we limited the number of people in the environment to only two.

In addition to taking care of the environmental condition for the health of the subjects, we were also careful with environmental aspects to reduce noise in the data. For instance, we did not use a swivel chair because, in previous pilot studies, subjects tended to move, which reduced the precision of the eye tracking equipment. Despite the measures we have taken, obtaining perfect data is virtually impossible, given camera limitations. Thus, we as researchers have discussed the collected data, plotted, interpreted, and performed data correction by slightly shifting chunks of fixations in the *y*-axis. We discuss in more detail this strategy in the threats to validity section (see Section 3.4.1). The dataset generated during the current study is available on a repository within a replication package also containing forms, programs, fixation data, data correction strategy, and other materials [59].

### 3.2.3 Subjects

Our study included 32 undergraduate students that we call "novices". On average, our subjects had 20 months of experience with programming languages, which mainly included Python, Java, JavaScript, C, and C++. However, only in Python, the language in which the programs were written, they had on average seven months of experience. Thus, we refer to our subjects as novices in Python language. They were recruited from three distinct universities in Brazil, invited mainly through e-mails or text messages. All subjects were Brazilian Portuguese speakers enrolled in academic universities.

For the sample size, we followed Cohen's effect types and suggested conventional values to estimate a large effect size of 0.8 [26] and based on a related work on atoms of confusion that estimated the nominal power of 0.8 [48]. We computed the number of subjects necessary to have a minimal power of 0.8 with a significant level of 0.05 using the T-test sample size computation. Our analysis revealed that we need 26 subjects in two samples to have a minimal power of 0.8 with a significant level of 0.05. Alternatively, since we have 32 subjects

instead of 26, our study can also detect a moderate effect size of 0.5 with a power of 0.5 with a significant level of 0.05.

### 3.2.4 Treatments

Each subject has the task of examining 12 programs ($P_1$–$P_{12}$). To avoid that the same subject takes a program in obfuscated and clarified versions, causing learning effects, we have designed 24 distinct programs divided in two sets of programs ($SP_1$ and $SP_2$), each containing 12 programs. A subject takes six obfuscated programs (O) of the first set, for instance, $SP_1$, and six clarified programs (C) of the second set, $SP_2$, as seen in Figure 3.1. Both sets $SP_1$ and $SP_2$ have the same atoms, however, instantiated in distinct programs with distinct outputs. The obfuscated programs are our baseline group, and the clarified are the treatment group.

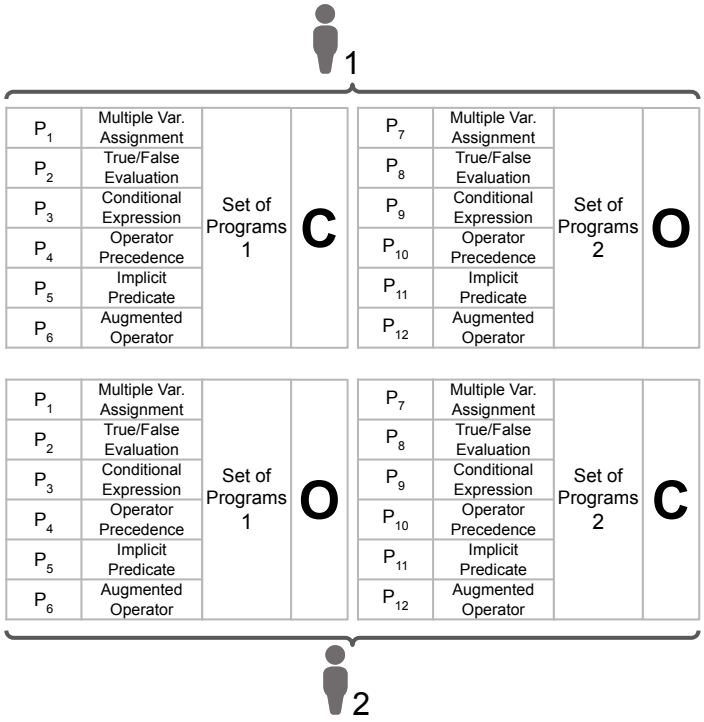



Figure 3.1: Structure of a latin square. $Subject_1$ takes six programs ($P_1$–$P_6$) which are clarified versions (C) of the programs containing each of the atoms. These programs are from Set of Programs 1 ($SP_1$). $Subject_1$ also takes six programs ($P_7$–$P_{12}$) from the Set of Programs 2 ($SP_2$) comprising the obfuscated code (O) containing the atoms. $Subject_2$ takes the complement to that.

For instance, consider Figure 3.2, which depicts how the atom *Multiple Variable As-*

*signment* ($A_1$) is evaluated in the programs $P_1$ and $P_7$ of the set of programs $SP_1$ and $SP_2$, respectively. The first subject examines the program $P_1$, which is the clarified version of a program containing $A_1$, present in the set of programs $SP_1$ ($P_1$ - $A_1$ - $SP_1$ - C). The same subject examines another program, $P_7$, which contains the obfuscating atom $A_1$, present in the set $SP_2$ ($P_7$ - $A_1$ - $SP_2$ - O), which prints a distinct output from that one in $SP_1$ to reduce learning effects. A similar idea applies to the second subject. The second subject examines $P_1$ containing the obfuscating atom $A_1$ present in $SP_1$ ($P_1$ - $A_1$ - $SP_1$ - O) and then the same subject examines another program, $P_7$, which is the clarified version of a program containing $A_1$, present in $SP_2$ ($P_7$ - $A_1$ - $SP_2$ - C). It is important to mention that being in the same set, the programs $P_1$ - $A_1$ - $SP_1$ - O prints the same output of $P_1$ - $A_1$ - $SP_1$ - C, however, the first program contains the obfuscating atom and the second is a clarified version of the obfuscated program. Both programs are examined by distinct subjects as well. In all the programs, the subjects had the task of specifying the correct output in an open-ended fashion, which means that each subject has to read the entire code and find the output without being presented with multiple options.

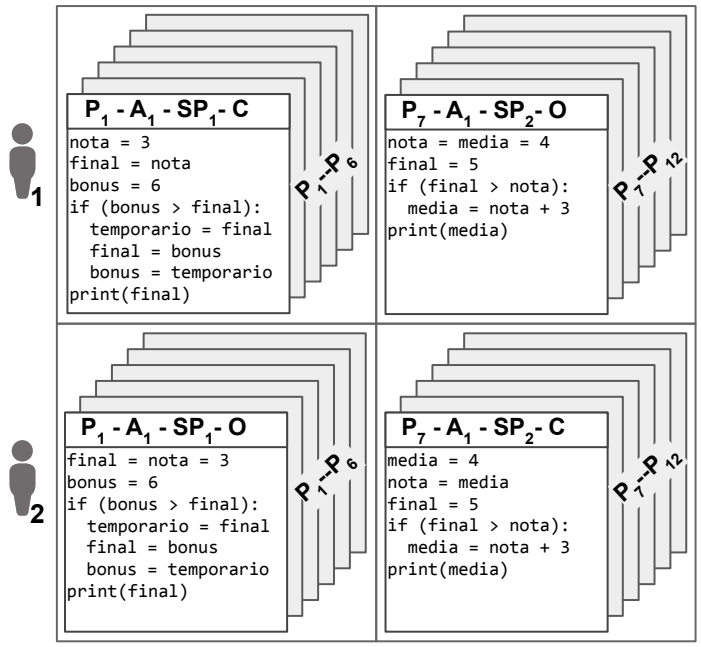



Figure 3.2: The structure of the programs $P_1$ and $P_7$, whether obfuscated or clarified, from $SP_1$ and $SP_2$. We present all the programs with the obfuscating atoms and the clarified versions of the programs from $SP_1$ and $SP_2$ in our supplementary material.

### 3.2.5 Evaluated Atoms of Confusion

We evaluated six atoms of confusion, which are summarized in Table 3.1. We selected four of them, namely *Multiple Variable Assignment*, *Conditional Expression*, *Operator Precedence*, and *Implicit Predicate* from a popular catalog of atoms for C that was proposed by Gopstein [48; 49], and further adapted to Java [66; 78]. We selected atoms that varied in their characteristics, which we found relevant for our investigation purpose, could be adapted for Python, and that were found in real projects. Gopstein et al. [48] studied the prevalence of the atoms of confusion in 14 large and significant open-source projects. The four selected atoms in our study that were based on their work are among the seven most found atoms in the evaluated projects.

| **Atoms** | **Description** | **Obfuscated** | **Clarified** |
|---|---|---|---|
| Multiple Variable Assignment | We can assign the same value to multiple variables by using = consecutively | `a = b = 1` | `b = 1`<br>`a = b` |
| True or False Evaluation | Directly checking whether something is `True` versus checking whether its negation is `False` | `(not a == b)` | `(a != b)` |
| Conditional Expression | Sometimes called "ternary operator", an `if` statement can be written in one line with conditional expression | `a = b if b == c else d` | `if (b == c):`<br>`  a = b`<br>`else:`<br>`  a = d` |
| Operator Precedence | The precedence of the operators influences the outcome. Depending on the order, we might have different results | `a and b or c` | `(a and b) or c` |
| Implicit Predicate | An expression that does not produce a `bool` is used as a predicate | `(a % b)` | `(a % b != 0)` |
| Augmented Operator | A single operator adds a value to a variable and updates it | `a += 1` | `a = a + 1` |

Table 3.1: Atoms evaluated in this study.

In their catalog, Gopstein et al. [48] exemplify *Multiple Variable Assignment* as `V1 = V2 = 3`. This atom was originally named *Assignment as Value*. Nevertheless, in Python, assignments are not expressions, and this pattern has different semantics. It means that the same value is assigned to multiple variables. Due to the syntactic similarity and semantic difference, we use it as an atom and name it *Multiple Variable Assignment*.

In addition to those four atoms, we selected two other patterns that are found in style guides for Python language, namely *True or False Evaluation*[1] and *Augmented Operator*.[2]

[1] https://google.github.io/styleguide/pyguide.html
[2] https://www.python.org/dev/peps/pep-0577/

While planning the experiment, we came up with distinct atom candidates that had the potential to cause confusion. We gave preference to candidates that were commonly used in practice, comprising distinct characteristics, and that we had alternative ways to write them. However, the pilot tests were crucial to determining what candidates would be more likely to cause confusion. We aimed to understand how our approach to evaluating those four atoms of confusion could possibly be applied to evaluate these two atom candidates. In general, we conducted pilot tests to arrive at these particular six atoms through refinements, investigation of feasibility, and discussions.

### 3.2.6 Programs

We selected code snippets by manually analyzing code repositories of programming activities for introductory programming students. We mainly targeted GeeksForGeeks[3] and Leetcode[4], which are popular code bases with programming activities for practicing and coding interviews. Given that we focused on novices, we selected easy, small, and complete problems and adapted them to the camera constraints. We present an example of our programs for each atom of confusion in Figure 3.3.

A prior systematic literature review on code comprehension conducted by Oliveira et al. [84] revealed that 70% of the studies in this domain involves asking subjects to provide information about the code, such as predicting the output. Following this commonly adopted methodology, we provided a code snippet to the subjects and asked them to specify the correct output. We are aware of the existence of other types of tasks in code maintenance and evolution, such as adding a functionality, refactoring the code, and fixing bugs. However, we based this study on the assumption that the subject will need at least to know the output of the code or the state of the variables to perform these other activities.

We used programs with less than ten lines of code to fit completely on the screen. All the programs are free of syntactic errors. We used simple constructions that commonly occur in many programming languages. We made sure that each program contains exactly either one or zero instances of one atom, whether obfuscated or clarified, respectively. Each program prints only one correct output given a set of possible outputs. We avoided letting

[3] https://www.geeksforgeeks.org/

[4] https://leetcode.com/

the programs present only two possible outputs, even though, given the logic of the code, the code containing the *Conditional Expression* (Figure 3.3(c)) and *Operator Precedence* (Figure 3.3(d)) do that. We made sure that each version of the program, obfuscated and clarified, from the same set of programs, presented the same output. For instance, the right-hand side and left-hand side of Figure 3.3(a). It is important to stress that, due to the use of a Latin Square design, no subject was presented with the clear and obfuscated versions of the same program. The programs followed Consolas font style, font size 16, line spacing of 1.5 inches, and eight white spaces of indentation.

### 3.2.7 Eye Tracking System

We used the Tobii Eye Tracker 4C in our experiment with a sample rate of 90 Hz. The calibration of the eye tracker followed the standard procedure of the device driver: while calibrating, the subject is asked to look at five points appearing one at a time randomly, twice, then, at the final stage, eight points appear for checking the calibration, three to the left, two in the middle, and three to the right. The eye tracker was mounted on a laptop screen with a resolution of 1366 x 720 pixels, a width of 30.9 cm, and a height of 17.4 cm, at a distance of 50-60 cm from the subject. The code tasks were displayed as an image in the full-screen mode but no Integrated Development Environment (IDE) was used, nor number for the lines. From this distance, we compute an accuracy error of 0.7 degrees which translates to 0.6 lines of inaccuracy on the screen, considering the font size we used and the line spacing. The line spacing was tested in the pilot study to be sufficiently large so we could overcome the eye tracker accuracy limitations. For processing the gaze data, we implemented a script in Python, which allowed us to analyze and collect the metrics.

### 3.2.8 Fixation and Saccades Instrumentation

During a fixation, our visual attention is focused on a specific area of the stimulus and triggers cognitive processes [60]. Thus, a fixation can be understood as the stabilization of the eye on part of a stimulus for a period of time, and the rapid eye movements between two fixations are called saccades [93; 55]. The visual stimulus can be any object, for instance, a piece

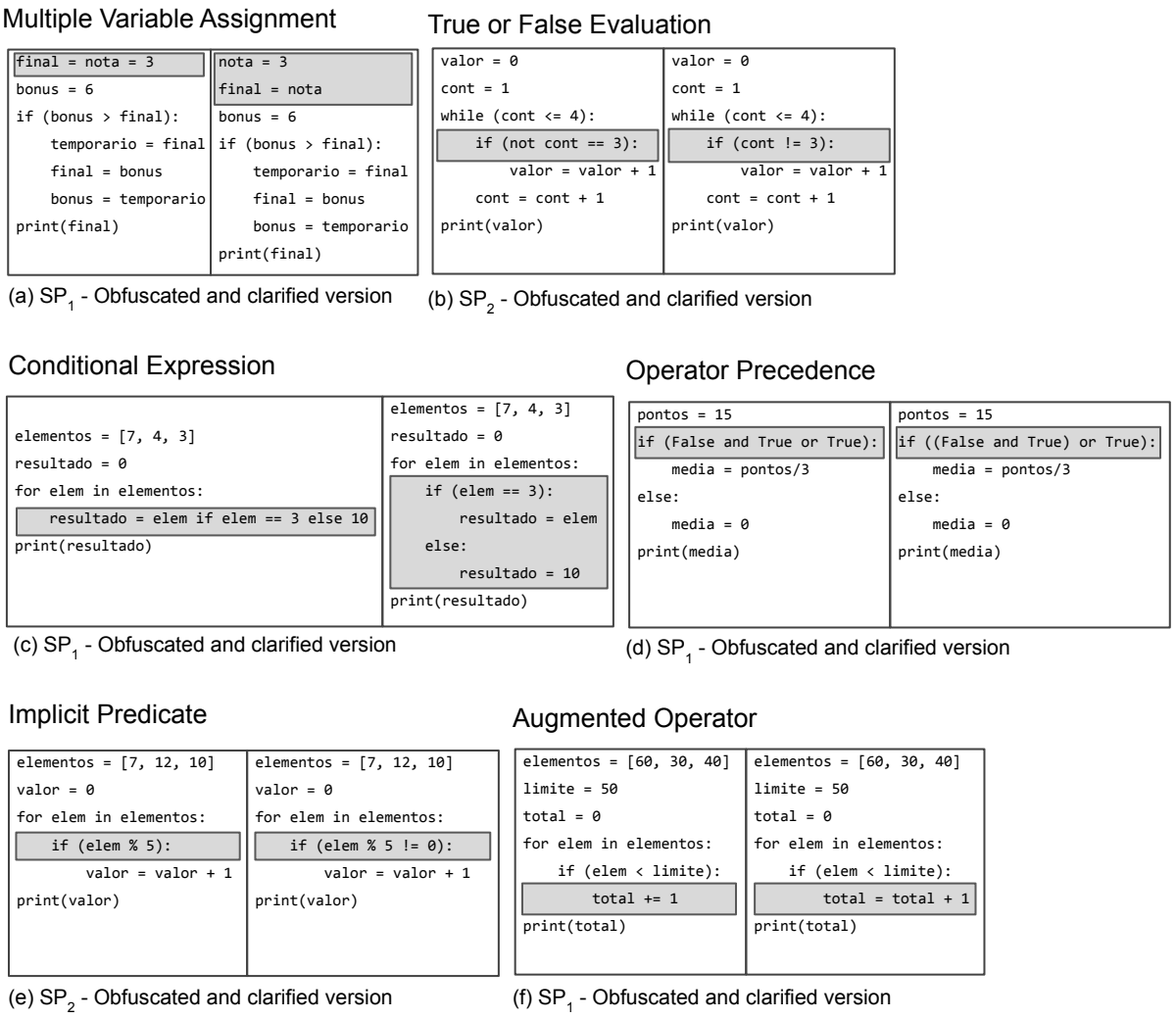



Figure 3.3: Examples of programs from set of programs $SP_1$ and $SP_2$ with obfuscated (left-hand side) versions of the code containing the atoms *Multiple Variable Assignment*, *True or False Evaluation*, *Conditional Expression*, *Operator Precedence*, *Implicit Predicate*, and *Augmented Operator*, and their respective clarified (right-hand side) versions. Shaded areas represent the AOIs, which are the code lines in which both obfuscated and clarified versions differ.

of source code, over which the subject performs a task, and whose visual perception by the subject triggers cognitive processes and actions, such as edit of a statement in a source code file [100].

There is no standardized threshold in the literature to specify the exact period of time for a fixation because duration usually depends on the processing demands of the task. However, we have some guidelines popular among eye tracking researchers. Salvucci and Goldberg [93] define a fixation as the eye being stable for a period of time between 100 and 200 ms, while according to Rayner [91], our eyes remain relatively still during fixations for about 200–300 ms when reading natural language text. Thus, after analyzing our programs, we used 200 ms as our threshold. Eye tracking researchers usually use an algorithm to classify gaze samples into fixations based on this threshold.

In this study, we used a Dispersion-Based algorithm to classify the fixations. In particular, we used the Dispersion-Threshold Identification (I-DT) [93]. We also classified gaze samples as belonging to a fixation if the samples are located within a spatial region of approximately 0.5 degrees [81]. This region corresponded to 25 pixels on our screen.

### 3.2.9 Analysis of the Results

From the total of 384 programs, the subjects solved 329, which corresponds to 85.6% of the programs. This set of solved programs includes programs that were solved either in the first attempt or after many attempts, however, both were solved within the time limit. We based our analysis only on these solved programs within the time limit. From this total, 160 programs were obfuscated and 169 were clarified.

Due to technical issues with the camera, we missed the data for two programs. They consist of only 0.5% of the data. Since they were not associated with correct or incorrect answers, and as a requirement of the statistical analysis based on the Latin Square design, we decided to impute them. Aiming to impute missing data for two programs, we used the Multivariate Imputation by Chained Equations (MICE) method implemented as a mice package in R for multiple imputations namely Predictive Mean Matching (PMM). The PMM method uses the predictive mean matching [58] to impute univariate missing data. This method performs better when the data sample size is sufficiently large [64], which was our case.

Once we had collected our data, we performed a statistical analysis to test our null hypotheses. We determine a significance level of 0.05, which means 5% risk of concluding that there is a difference when there is no actual difference. Whenever our *p*-value is equal or inferior to 0.05, we reject the null hypothesis that there was no difference between the median of the treatments.

We used statistical tests to compare two groups regarding the time, number of attempts, and visual metrics. The obfuscated programs are the control group, and the clarified programs are our treatment group. Following a practical guide on eye tracking studies [100], to test if data were normally distributed, we used Shapiro-Wilk Test [97]. For the normally distributed data, we performed the parametric *t* test for the two independent samples. The *t* test can be used to verify whether there is a statistically significant difference between two groups [104; 100]. However, before performing the *t* test, we verified whether the variances of the two groups were equal [104]. For the data that do not follow a normal distribution and that could not be normalized, we used the non-parametric test Mann-Whitney, also known as the Wilcoxon test, which can be applied to this specific situations [104; 100]. The mean value in the data might not be appropriate to characterize the fixations because the central tendency might depend on some very high values [43]. Thus, we based on the median as a measure of central tendency. To compare the six levels of atoms (six groups), we used ANOVA for normal data and Kruskal–Wallis for non-normal data. We used the post-hoc Dunn's Test with p-values adjusted with the Bonferroni method to identify which groups differed.

We adopted the following methodology to identify code reading patterns in our data: we identified a set of regions in the code, such as variable definition, loop condition, if condition, and others. Then, we defined these regions in pixels on the images of the tasks. We defined the regions inside the code according to a previous guideline [55]. Our hypothesis drove the regions; their positioning was precisely defined with 10 pixels to the right, left, top, and bottom, considering the camera limitations so that we could have a margin between the regions and avoid overlapping. In addition, we defined the white-space as a region so that we could be aware of any threat to validity given the camera limitations. Using the chronological order of the fixations and their positions, we identified a sequence of visited regions for each subject. We then built a big picture of the sequences by simplifying repeated transitions from

one region to the same region. We make the sequences and the images of the tasks with the regions identified available in our supplementary material [59].

## 3.3 Results and Discussion

In this section, we present and discuss our results for each atom. We present the *Multiple Variable Assignment* (Section 3.3.1), *True or False Evaluation* (Section 3.3.2), *Conditional Expression* (Section 3.3.3), *Operator Precedence* (Section 3.3.4), *Implicit Predicate* (Section 3.3.5), and *Augmented Operator* (Section 3.3.6). We also present the coding of the subjects' answers (Section 3.3.7) and other analysis (Section 3.3.8).

### 3.3.1 Multiple Variable Assignment

In Figure 3.4, we depict the obfuscating atom *Multiple Variable Assignment* on the left-hand side and the clarified version on the right-hand side. They differ in that the clarified version has two lines of code in the AOI, and one variable is repeated. In Table 3.2, we consider two perspectives of the metrics, one examining only the AOI and the other examining the whole code. While the time in the code, for instance, consists of the time one requires to examine and solve the task regardless of the fixations made, the time in AOI consists of examining only the region of the atom. As shown in Table 3.2, the subjects spent 30.1% more time and 60% more visual regressions in the AOI with the clarified version of the code containing *Multiple Variable Assignment*. In addition, they exhibit 38.4% more fixations, and the duration of the fixations is 22.9% higher in the clarified.

**Obfuscated**

```
final = nota = 3
```

**Clarified**

```
nota = 3
final = nota
```

Figure 3.4: Obfuscating atom *Multiple Variable Assignment* and clarified version.

The clarified version has one more element to observe and one line to go back, which can explain the need for more time, more fixations, and more regressions examining. To better understand it, we investigated distinguishing between a regression to a previous line, vertical regression, and a regression within the same line, or horizontal regression. We

| | | In the AOI | | | | | In the Code | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **Atoms** | **Metrics** | O | C | PD % | PV | ES | O | C | PD % | PV | ES |
| | Time (sec) | 8.3 | 10.9 | ↑30.1 | **0.03** | 0.31 | 41.1 | 44.3 | ↑7.6 | 0.85 | n/a |
| | Attempts | n/a | n/a | n/a | n/a | n/a | 1.19 | 1.07 | ↓10.0 | 0.16 | n/a |
| Multiple | Fix. Duration (sec) | 4.2 | 5.1 | ↑22.9 | 0.15 | n/a | 19.4 | 18.2 | ↓6.1 | 0.93 | n/a |
| Variable | Fix. Count | 13.0 | 18.0 | ↑38.4 | 0.17 | n/a | 59.5 | 59.5 | 0.0 | 0.86 | n/a |
| Assignment | Reg. Count | 2.5 | 4.0 | ↑60.0 | **0.04** | 0.29 | 26.0 | 25.0 | ↓3.8 | 0.67 | n/a |
| | Horiz. Reg. Count | 2.5 | 2.0 | ↓20.0 | 0.98 | n/a | 11.5 | 10.0 | ↓13.0 | 0.95 | n/a |
| | Vert. Reg. Count | 0.0 | 2.0 | ↑Inf | **0.000** | – | 14.0 | 14.5 | ↓3.5 | 0.42 | n/a |
| | Time (sec) | 20.3 | 16.3 | ↓19.6 | 0.24 | n/a | 61.2 | 55.9 | ↓8.5 | 0.93 | n/a |
| | Attempts | n/a | n/a | n/a | n/a | n/a | 1.25 | 1.37 | ↑9.6 | 0.36 | n/a |
| True or | Fix. Duration (sec) | 10.1 | 10.6 | ↓4.7 | 0.47 | n/a | 35.4 | 27.0 | ↓23.7 | 0.81 | n/a |
| False | Fix. Count | 30.5 | 28.0 | ↓8.2 | 0.22 | n/a | 92.0 | 79.0 | ↓14.1 | 0.77 | n/a |
| Evaluation | Reg. Count | 9.5 | 5.0 | ↓47.3 | **0.03** | -0.34 | 41.5 | 35.0 | ↓15.6 | 0.74 | n/a |
| | Horiz. Reg. Count | 9.5 | 5.0 | ↓47.3 | **0.03** | – | 23.5 | 15.0 | ↓36.1 | 0.58 | n/a |
| | Vert. Reg. Count | 0.0 | 0.0 | n/a | n/a | n/a | 17 | 16.0 | ↓5.8 | 0.87 | n/a |
| | Time (sec) | 30.7 | 32.1 | ↓4.3 | 0.84 | n/a | 71.6 | 62.7 | ↓12.3 | 0.24 | n/a |
| | Attempts | n/a | n/a | n/a | n/a | n/a | 1.22 | 1.14 | ↓8.8 | 0.46 | n/a |
| Conditional | Fix. Duration (sec) | 18.9 | 14.0 | ↓25.5 | 0.44 | n/a | 34.3 | 26.9 | ↓21.5 | 0.18 | n/a |
| Expression | Fix. Count | 59.0 | 41.0 | ↓30.5 | 0.33 | n/a | 107.0 | 78.0 | ↓27.1 | 0.21 | n/a |
| | Reg. Count | 15.0 | 14.0 | ↓6.6 | 0.71 | n/a | 43.0 | 35.0 | ↓18.6 | 0.38 | n/a |
| | Horiz. Reg. Count | 14.0 | 8.0 | ↓42.8 | **0.01** | – | 26.0 | 14.0 | ↓46.8 | **0.02** | – |
| | Vert. Reg. Count | 0.0 | 6.0 | ↑Inf | **0.000** | – | 17.0 | 21.0 | ↑23.5 | 0.45 | n/a |
| | Time (sec) | 20.2 | 12.4 | ↓38.6 | **0.009** | -0.37 | 43.5 | 34.7 | ↓20.1 | **0.04** | 0.29 |
| | Attempts | n/a | n/a | n/a | n/a | n/a | 1.62 | 1.16 | ↓28.3 | **2x10**$^{-4}$ | -0.46 |
| Operator | Fix. Duration (sec) | 11.0 | 7.3 | ↓34.1 | **0.02** | -0.33 | 19.9 | 17.1 | ↓14.1 | 0.08 | n/a |
| Precedence | Fix. Count | 32.5 | 22.0 | ↓32.3 | **0.02** | -0.32 | 57.5 | 49.0 | ↓14.7 | 0.07 | n/a |
| | Reg. Count | 10.0 | 5.0 | ↓50.0 | **0.02** | -0.33 | 25.0 | 19.0 | ↓24.0 | 0.06 | n/a |
| | Horiz. Reg. Count | 10.0 | 5.0 | ↓50.0 | **0.02** | – | 14.0 | 10.0 | ↓28.5 | **0.04** | – |
| | Vert. Reg. Count | 0.0 | 0.0 | n/a | n/a | n/a | 11.5 | 9.0 | ↓21.7 | 0.09 | n/a |
| | Time (sec) | 26.4 | 17.3 | ↓34.4 | 0.29 | n/a | 71.2 | 47.6 | ↓33.1 | 0.10 | n/a |
| | Attempts | n/a | n/a | n/a | n/a | n/a | 1.42 | 1.18 | ↓16.9 | 0.16 | n/a |
| Implicit | Fix. Duration (sec) | 16.1 | 11.0 | ↓31.5 | 0.42 | n/a | 35.1 | 25.9 | ↓26.2 | 0.33 | n/a |
| Predicate | Fix. Count | 43.0 | 29.0 | ↓32.5 | 0.45 | n/a | 86.0 | 64.5 | ↓25.0 | 0.28 | n/a |
| | Reg. Count | 10.0 | 8.0 | ↓20.0 | 0.88 | n/a | 40.0 | 28.5 | ↓28.7 | 0.26 | n/a |
| | Horiz. Reg. Count | 10.0 | 8.0 | ↓20.0 | 0.88 | n/a | 22.0 | 14.5 | ↓34.0 | 0.44 | n/a |
| | Vert. Reg. Count | 0.0 | 0.0 | n/a | n/a | n/a | 21.0 | 12.0 | ↓42.8 | 0.13 | n/a |
| | Time (sec) | 7.2 | 5.9 | ↓17.6 | 0.18 | n/a | 45.7 | 36.5 | ↓20.0 | 0.30 | n/a |
| | Attempts | n/a | n/a | n/a | n/a | n/a | 1.13 | 1.20 | ↑6.1 | 0.69 | n/a |
| Augmented | Fix. Duration (sec) | 4.4 | 2.8 | ↓35.4 | 0.09 | n/a | 20.4 | 16.4 | ↓19.3 | 0.29 | n/a |
| Operator | Fix. Count | 11.5 | 8.0 | ↓30.4 | 0.22 | n/a | 58.5 | 47.5 | ↓18.8 | 0.41 | n/a |
| | Reg. Count | 2.0 | 1.0 | ↓50.0 | 0.36 | n/a | 24.0 | 17.0 | ↓29.1 | 0.31 | n/a |
| | Horiz. Reg. Count | 2.0 | 1.0 | ↓50.0 | 0.36 | n/a | 9.0 | 6.0 | ↓33.3 | 0.08 | n/a |
| | Vert. Reg. Count | 0.0 | 0.0 | n/a | n/a | n/a | 13.0 | 11.0 | ↓15.3 | 0.71 | n/a |

Table 3.2: Results for all metrics for all atoms. O = obfuscated code; C = clarified code; PD = percentage difference; PV = *p*-value; ES = effect size (Cliff's delta). Columns O and C are based on the median as a measure of central tendency, except for attempts, which are based on the mean.

found that, while the subjects regress less horizontally, they make more vertical regressions with the *Multiple Variable Assignment* (see Table 3.2).

The clarified version presented slightly more vertical regressions in the code and fewer horizontal ones when considering the two sets of programs. In Figure 3.5, we depict an example of the distribution of the regressions for two subjects who examine a program of $SP_1$. We selected subjects whose patterns hold for all subjects. One takes the code with the atom *Multiple Variable Assignment*, and the other takes its respective clarified version of the code. In the graph, each edge represents a regression with a direction to a previous line of code or the same line. Each node represents a line of code. The grayscale intensity of the edge represents the number of times such regression was repeated. Adding one more line might explain such increase in the vertical regressions. The reduction in the number of horizontal regressions in the AOI might be because, in the clarified version, we have two lines, but they are shorter compared to the obfuscated version.

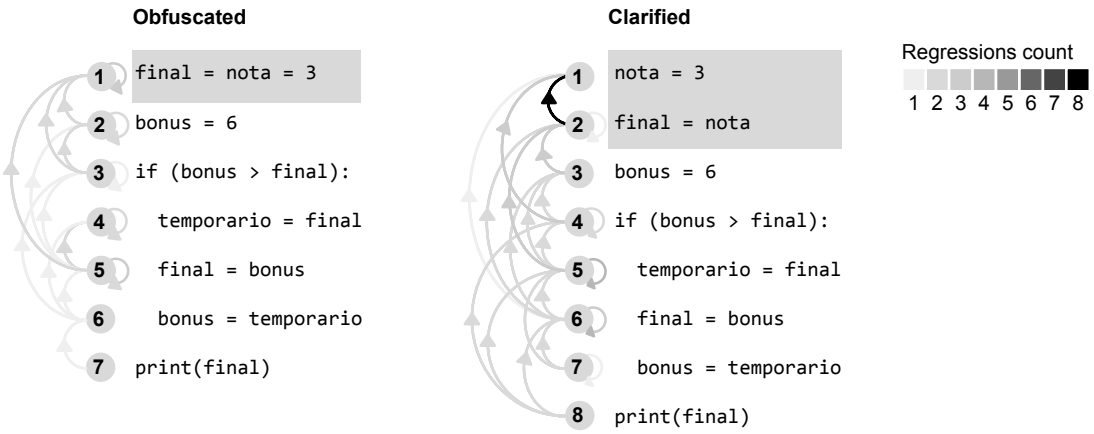



Figure 3.5: Two subjects visually regressing horizontally and vertically while examining the code, one with the code containing the obfuscating atom *Multiple Variable Assignment* and the other with the clarified version of the code.

To deepen our analysis and look for reading patterns in our programs, among our subjects, we identified a set of Regions (R) in the code as in Figure 3.6. The colors distinguish between distinct regions and are identified by codes such as R1, R2, and so forth. Since each program has its control flow, we expect to find sequential patterns within the sequence of regions read in a specific order. We analyzed the chronological order of the

regions fixated by the subjects. For the *Multiple Variable Assignment*, we observed that the sequence R3 → R4 in the clarified is made 25% more times than the same sequence R1 → R2 in the clarified version. Still, 52% of subjects in both versions do the reverse sequence, going from the R4 → R3 and R1 → R2. The association and understanding of this pair of variables, R1 receiving R2 in the obfuscated, depends on the value assigned in R3; therefore, one can understand the whole expression going forward or backward. However, breaking this multiple assignments into two lines made it difficult to associate R1 to R3. The sequence R3 → R1, for example, in the obfuscated, happens 68% more than expected, while in the clarified one, it occurs 112% more.

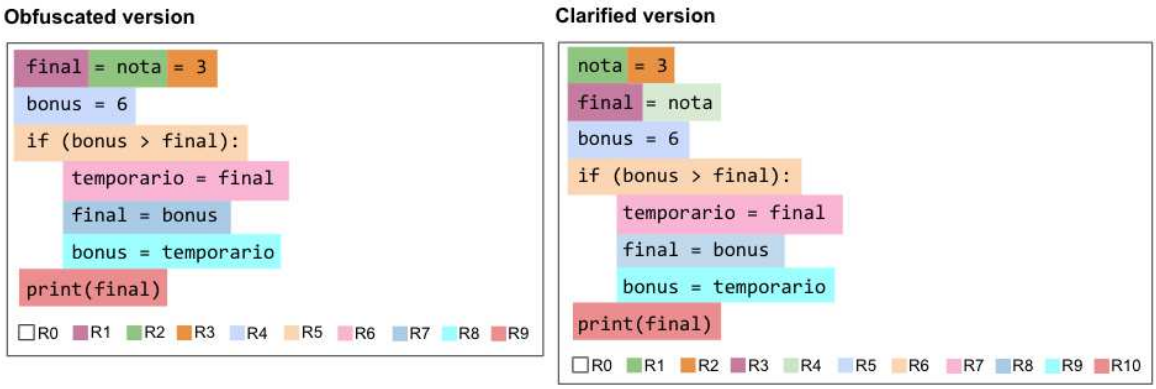



Figure 3.6: Set of regions inside the code version with *Multiple Variable Assignment* atom and in the code with the clarified version of code.

In the clarified version, the subjects go back and forth between the two lines to observe the same variable. Of the subjects, 50% make R1 → R4 16 times, while 50% go back making R4 → R2 12 times. This can indicate confusion, given that the subjects have difficulty associating the same variable between different lines or that they need to remember the assigned value. Of the two subjects who failed to solve the task, on average, they go back and forth between R1 and R4 6 times. The transition R3 → R4 in the clarified is expected to be performed at least once by each subject. But we found that 68% of subjects exhibit it for 209% more than expected. Subjects may forget or make incorrect associations with the variable R3, which is used five times in the code, and in one of them, the variable is updated. When the task requires more use of temporary memory, it is necessary for subjects to go back in code to refresh their memory. On average, 52% of subjects return from the lines that

later use R3 to it.

We used a five-point scale to assess the opinions of the subjects concerning how difficult they perceived the programs to be solved. We asked the subjects to rate each program individually whether they found it very easy, easy, neutral, difficult, or very difficult. In Figure 3.7, we compare their perceptions with obfuscated and clarified versions of the code containing the evaluated atoms.

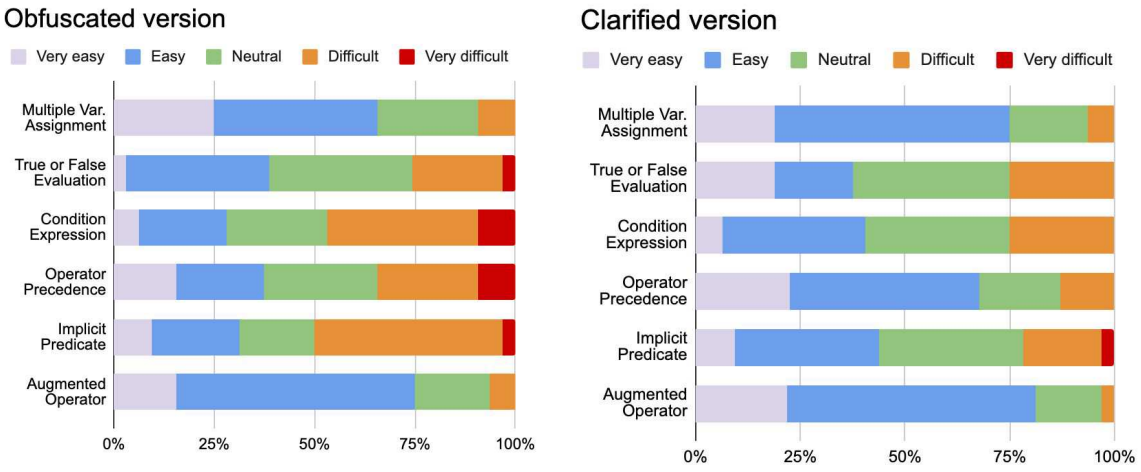



Figure 3.7: Perception of difficulties with obfuscated code containing the evaluated atoms and their clarified version.

The subjects perceive the obfuscated and clarified versions of the code as similar in terms of difficulty, according to Figure 3.7(a). The figure shows that 9.4% of the subjects found the obfuscated version difficult to be solved, while 6.3% had that opinion about the clarified version. At the same time, 25% considered the obfuscated version to be very easy, whereas for the clarified version 18.8% had the same opinion. However, from the quantitative perspective, the subjects need 30.1% more time and make 60% more regressions with the clarified version. It indicates a discrepancy between how subjects subjectively perceive the difficulty of the task and their performance on it. Such disagreement can be explained by the fact that self-evaluation of difficulties can be an intrinsically difficult activity. Not all subjects might be aware of their own effort while performing a task, such as going back and forth in the code or remembering the variables' intermediate states. In this scenario, we triangulate the subjective feedback with the other perspectives to better comprehend the phenomena of

comprehension. Regarding the difference in time in the AOI and Code, we observed that, in the clarified version, the subjects mentioned difficulties with memorizing variables and difficulties with swapping values in the if condition. This might indicate that repeating the variable in the atom region may influence both their opinions and the variables' associations in the swapping region code outside AOI.

We conducted semi-structured interviews mainly to identify the subject's difficulties, better interpret the results, and perform minor sanity tests. In the obfuscated version, the subjects mentioned the issues: *"first line is confusing"*, *"first line caused me trouble"*, *"first line is hard"*, *"many variable assignments"*, and *"beginning strange"*. In the clarified version, the subjects mentioned: *"many variables"* and *"`if` is confusing"*. As a takeaway, in the obfuscated version, the sources of confusion concentrate in the first line with the atom, while in the clarified version, they concentrate in the number of variables.

> In our study, in the clarified version of code with *Multiple Variable Assignment*, there is an increase in the time in the AOI and in the number of vertical regressions between the two lines. While in obfuscated, subjectively, the sources of confusion concentrate in the first line with the atom, while in the clarified version, they concentrate on the number of variables.

### 3.3.2 True or False Evaluation

In Figure 3.8, we depict the obfuscating atom *True or False Evaluation* on the left-hand side and the clarified version on the right-hand side. The clarified version removes the `not` operator and replaces the equality operator ("==") by a not equals ("!=") operator. In Table 3.2, the subjects spent 19.6% less time and 47.3% fewer visual regressions in the AOI with the clarified version of the code containing *True or False Evaluation*. They also exhibited slight reductions by 4.7% and 8.2% in the fixations count and fixation duration in the clarified. We found a positive correlation between time in the AOI and the number of attempts for the clarified version, which means increases in time were associated with more attempts.

The clarified version reduced the median number of horizontal regressions in the AOI and the number of entries and exits from the AOI. For instance, the obfuscated version of the program has 50% more horizontal regressions in the code than the clarified one. Within the

**Obfuscated**

```
if (not cont == 3):
```

**Clarified**

```
if (cont != 3):
```

Figure 3.8: Obfuscating atom *True or False Evaluation* and clarified version.

AOI, the obfuscated version has almost twice as many horizontal regressions. In Figure 3.9, we depict an example of the distribution of the regressions of two subjects on a program of $SP_2$ with the code containing the *True or False Evaluation* and the clarified version. Besides having more elements to observe horizontally, in the obfuscated version, the subjects have to check whether the right-hand side and the left-hand side are equal and then apply the `not` operator. A interpretation for the same-line regressions for non-AOI lines in only the obfuscated is that the variable in the AOI depends on the repetitive control structure of the loop before, and it affects the incrementing variable outside the AOI after. If the AOI is confusing, one can also make more same-line regressions in regions outside the AOI.

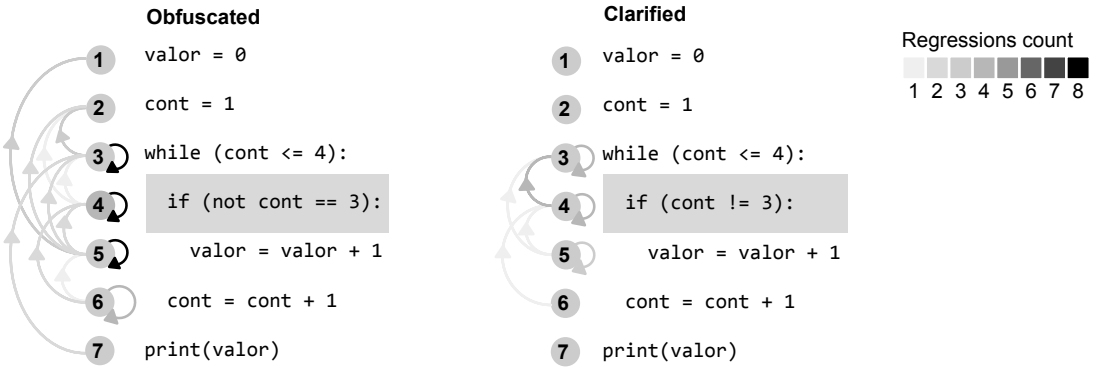



Figure 3.9: Two subjects visually regressing horizontally and vertically in the code, one with the obfuscated and the other with the clarified version of the code containing the atom *True or False Evaluation*.

The subjects present a similar number of transitions between the AOI and the rest of the code for the obfuscated code containing the *True or False Evaluation* and the clarified version. While regressions are represented as downward arrows, a transition describes eye movements in both forward and backward directions. They are depicted only in the transition graphs because we aimed to examine how many times the subjects enter and exit the AOI. The median number of times the subjects visually enter the AOI is the same for both versions. Nevertheless, in the clarified version, the subjects transition more with the

upper part of the code. In Figure 3.10, we give an example of these transitions with two subjects who examine a program of $SP_2$ with *True or False Evaluation*, one subject in each version. To convey the concept more concisely, the transitions between the lower part of the code and the upper part of the code are not present in the graph. In the obfuscated version, the subjects reported difficulties with the operator `not`, and the subject seems to visit more the lower part to make sense of it. In the clarified version, the subjects mentioned the increment and the loop, and they seem to visit the upper part with the `while` statement more times.

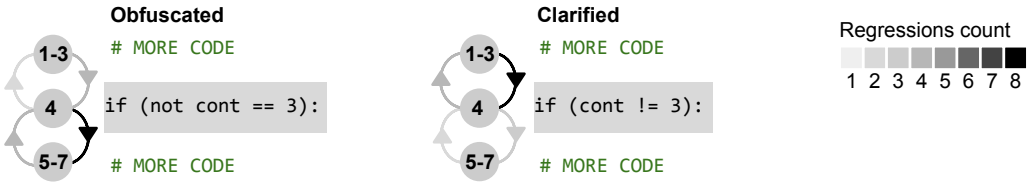



Figure 3.10: Two subjects visually entering and exiting the AOI in the code, one with the obfuscated code containing the *True or False Evaluation* and the other with the clarified version of the code.

In Figure 3.11, we depict the *True or False Evaluation* atom and its clarified version. We observed that for the obfuscated program, the addition of the particle `not` changes the visual dynamics, from computing to expressions, and possibly the way of understanding. In obfuscated, 87% of subjects go linearly R3 $\rightarrow$ R4, R4 $\rightarrow$ R5, and R5 $\rightarrow$ R6, which is more linear. While some subjects mentioned having difficulties understating R4, none mentioned making wrong associations regarding the order of precedence. The sequence R5 $\rightarrow$ R6 is expected to occur three times in the loop, but it occurs 109% more in the obfuscated. As the operator is equality, the subject can do the inverse, R6 $\rightarrow$ R5, which was observed in 87% of the subjects. However, this pattern is 76% higher than expected. In the clarified, the sequence R4 $\rightarrow$ R5 is seen by 93% of the subjects 44 times. This pattern is intuitive for subjects when they remember the value of the variable to be compared. As this value varies according to the increment inside the loop, the subject must be aware of these updates in the variable and how it interacts with other parts.

The subjects reported the same difficulties with both obfuscated and clarified versions.

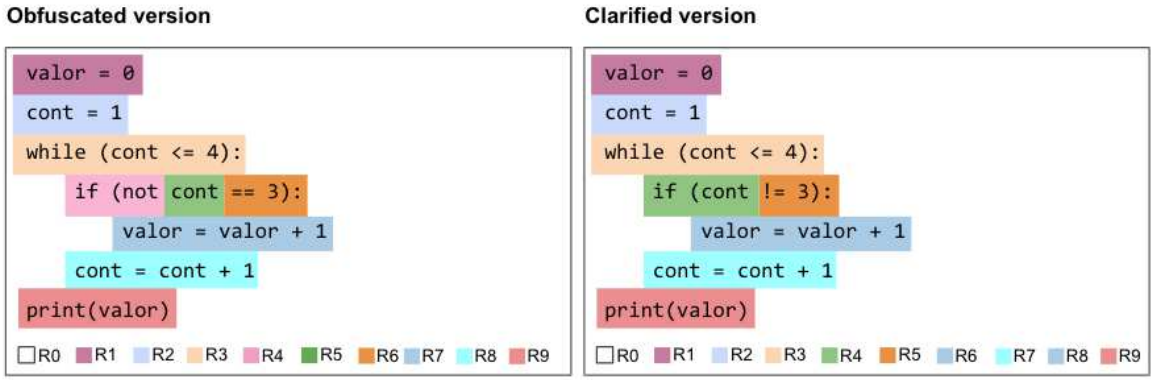



Figure 3.11: Set of regions inside the code version with *True or False Evaluation* atom and in the code with the clarified version of code.

Consider the *True or False Evaluation* in Figure 3.7(b). In it, 25% of the subjects found the code with the obfuscated version difficult or very difficult, while with the clarified version, 25% of the subjects found the code difficult (0% very difficult). However, from a quantitative perspective, the subjects tend to perform better with the clarified version, spending less time and making fewer regressions in the AOI and in the whole code.

In the obfuscated version, the subjects mainly mentioned the following issues: *"difficulties with* `not`*"*, *"*`not` *made it complicated"*, *"didn't understand* `not`*"*, *"*`not` *is strange"*, *"*`if` *complex"*, and *"*`not` *is confusing"*. For the clarified version, they mentioned: *"increment confusing"*, *"difficulties with* `if`*"*, and *"confused the loop"*. As a takeaway, in the obfuscated version, the sources of confusion are more concentrated in `not`, while in the clarified version, the sources are more diverse.

> In our study, in the clarified version of code with *True or False Evaluation*, there is a reduction in the number of horizontal regressions in the AOI. The other metrics were not affected as much. In the obfuscated version, subjectively, the sources of confusion are more concentrated in `not`, while in the clarified version, the sources are more diverse.

### 3.3.3 Conditional Expression

In Figure 3.12, we depict the obfuscating atom *Conditional Expression* on the left-hand side and the clarified version on the right-hand side. They differ in that the clarified version

has three more lines of code in the AOI than the obfuscated version and more elements, such as the repetition of one variable. However, with the clarified, we observed reductions in the duration of the fixations by 25.5%, 30% in the fixations count, and 42.8% in the horizontal regressions count. On the other hand, we observe an increase in the average number of vertical regressions from zero to six. We depict an example of a program of $SP_1$ in Figure 3.13. In the obfuscated version, the regressions of the subject are more concentrated horizontally within the same line that contains the atom, while in the clarified version, the regressions of the subject are distributed over more lines vertically.

**Obfuscated**

```
resultado = elem if elem == 3 else 10
```

**Clarified**

```
if (elem == 3):
  resultado = elem
else:
  resultado = 10
```

Figure 3.12: Obfuscating atom *Conditional Expression* and clarified version.

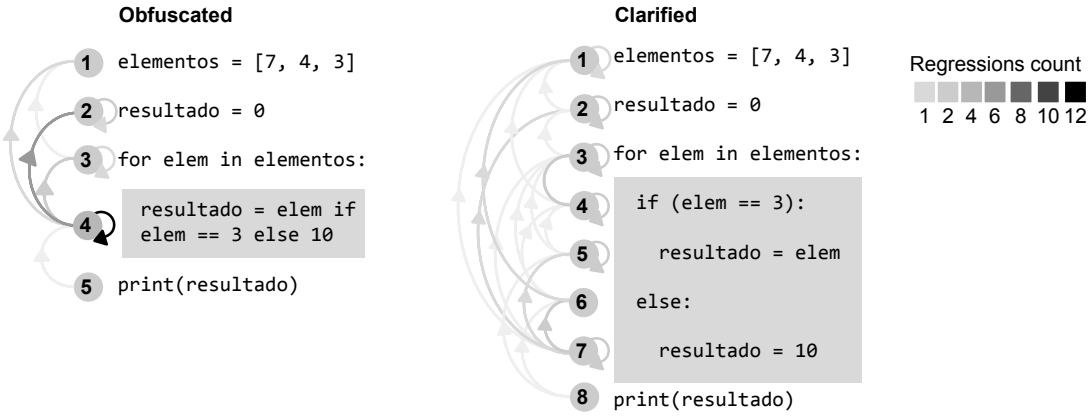



Figure 3.13: Two subjects visually regressing horizontally and vertically in the code, one with the obfuscated code containing the *Conditional Expression* and the other with the clarified version of the code.

We observed that all the subjects made the sequence R3 → R4 in the obfuscated, while the sequence with the same elements R3 → R5 in the clarified was made by only half of the subjects. Adopting a linear strategy, we expect the subjects to make the sequence R3 → R4 one time and return to it only when the if condition is tested as true. But we noticed that, in the obfuscated, they exhibited this pattern 162% more than expected. The type of structure adopted in the obfuscated version disfavors the type of linear reading since the subject tends

to perform the pattern of observing the R4 more than expected. In the clarified, however, 81% of the subjects exhibited the linear reading sequence R3 → R4 → R5. The sequence R3 → R4 is seen by all subject in the clarified occurring 81% more than expected, since the loop repeats three times. Therefore, despite the subjects exhibiting similar behaviors in both versions, in general, the structure of the obfuscated version shows more indicative of confusion.

The obfuscated version was associated with more programs that were not solved. With the obfuscated, seven programs were not solved, while in the clarified, only two. For instance, concerning the program in Figure 3.14, a subject who could not solve it, exhibited the sequence R5 → R4 600% more than expected. This behavior of returning several times, may indicate that the type of structure adopted in the obfuscated version can confuse the subject, making him/her return unnecessarily repetitively. Similarly, the sequence R3 → R4 is made 200% more than expected.

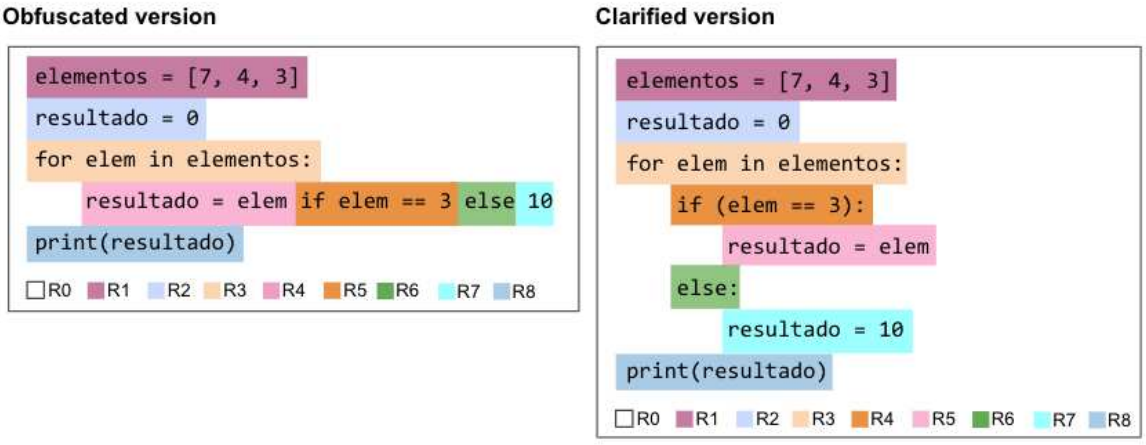



Figure 3.14: Set of regions inside the code version with *Conditional Expression* atom and in the code with the clarified version of code.

One of the reasons why we have larger differences in the code than AOI can be that, the clarified version makes the subjects go fewer times to the outside of the AOI. For instance, in Conditional Expression, in clarified, the subjects go 22% fewer times to the top where we have the variables declared.

The subjects perceive the obfuscated version as more difficult to understand. In Fig-

ure 3.7(c), 46.9% of the subjects found the code with the obfuscated version difficult or very difficult to be understood, whereas only 25% found the clarified version difficult. Quantitatively, we observed that there is an alignment between perception or difficulty and actual results. The subjects needed more time and exhibit more visual effort with the obfuscated version.

The main issues mentioned by the subjects for the obfuscated version were: *"conditional expression is confusing"*, *"more time to validate* `if` *conditional"*, *"* `if` *conditional is complicated"*, *"unsure about* `if` *conditional"*, *"didn't understand the list"*, *"didn't remember* `if` *conditional"*, and *"* `if` *difficult and I prefer another style"*. For the clarified version, they mentioned: *"* `for` *loop"*, *"* `if` *condition is difficult"*, *"difficulties with* `elem`*"*, *"remember* `cont` *variable"*, and *"indentation confusing"*, *"* `else` *difficult"*. As a takeaway, in the obfuscated version, confusion sources are more concentrated in the conditional expression. In the clarified version, the sources are more concentrated in variables and the condition of the `if` statement.

> In our study, in the clarified version of code with *Conditional Expression*, there is an increase in the number of vertical regressions in the AOI. The other metrics were not affected as much. In the obfuscated, subjectively, confusion sources are more concentrated in the conditional expression. In the clarified, they concentrate on variables and the condition of the conditional.

### 3.3.4 Operator Precedence

In Figure 3.15, we depict the obfuscating atom *Operator Precedence* on the left-hand side, and the clarified version on the right-hand side. The only change in the clarified version consists of the addition of two parentheses with the intention of clarifying the precedence of the boolean operators. With the clarified, we observed reductions in the time in the AOI by 38.6%, in the number of attempts by 28.3%, in the duration of the fixations by 34.1%, 32.3% in the fixations count, and 50% in the horizontal regressions count. We found a positive correlation between time in the AOI and the number of attempts for the obfuscated version, with increases in time associating with more attempts.

Knowing the order of precedence of the operators is essential to correctly solve the code since the wrong order yields a wrong output. For instance, `(False and True) or`

**Obfuscated**

```
if (False and True or True):
```

**Clarified**

```
if ((False and True) or True):
```

Figure 3.15: Obfuscating atom *Operator Precedence* and clarified version.

`True` when correctly interpreted according to Figure 3.16, right-hand side, prints `True`, while `False and (True or True)`, evaluated wrongly, prints `False`. The lower number of attempts with the clarified version combined with the reduced number of fixations and regressions, as well as fixation duration, suggests that the clarified version is improved by the addition of the parenthesis.

Eye tracking allows us to see the impact of adding the parenthesis at a fine-grained level. We observe that the clarified version reduced the median number of horizontal regressions by 28% in the code, possibly freeing the subjects from the need to go back and forth in the statement trying to figure out the right order of the boolean operators. That becomes more emphasized in the reduction of regressions in the AOI, by 47%. In Figure 3.16, we give an example of this reduction with two subjects who examine the program of $SP_1$ containing *Operator Precedence*, one subject in each version. In the obfuscated version, the regressions within the same line in the `if` statement are more intense, including more regressions to the previous line, compared to the clarified version.

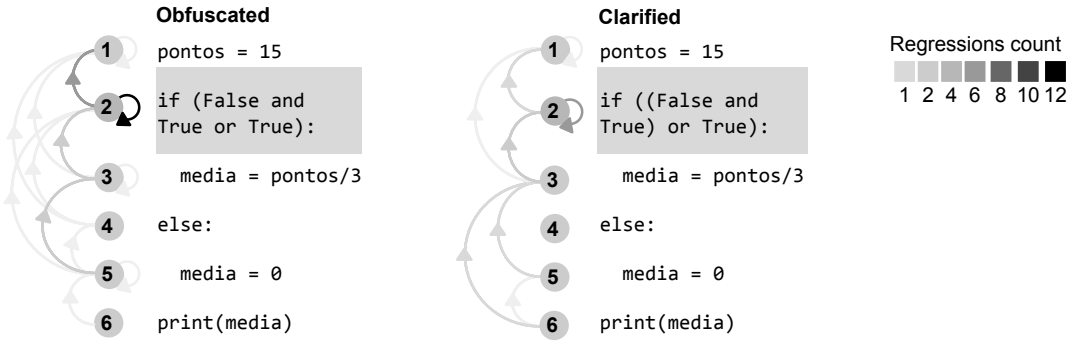



Figure 3.16: Two subjects visually regressing horizontally and vertically in the code, one with the obfuscated code containing the *Operator Precedence* and the other with the clarified version of the code.

In the obfuscated version, we observed that 75% of the subjects make the linear sequence R2 $\rightarrow$ R3 $\rightarrow$ R4 in both versions, however, the subjects made the sequence R2 $\rightarrow$ R3 16% more times than in the clarified, while they made the sequence R2 $\rightarrow$ R3 30% more times.

When subjects reach the end of the entire expression, they are expected to enter the true condition R4 as a result of correct comprehension. However, in the obfuscated version, this expected transition is 43% smaller than in the clarified version, which can indicate that, due to the lack of understanding of priority, subjects tend to get back on the line to try to understand again. We observed that in the obfuscated version, 68% of the subjects returned from R4 → R3, while in the clarified version, 62%. However, the number of regressions between R4 → R3 is 52% higher in the obfuscated version, which may indicate a wrong association because of priority. In addition, in the obfuscated, two subjects go from R4 → R7, which is moving to the incorrect condition. Both only got the programs solved in the second attempt and presented time in the AOI above the average.

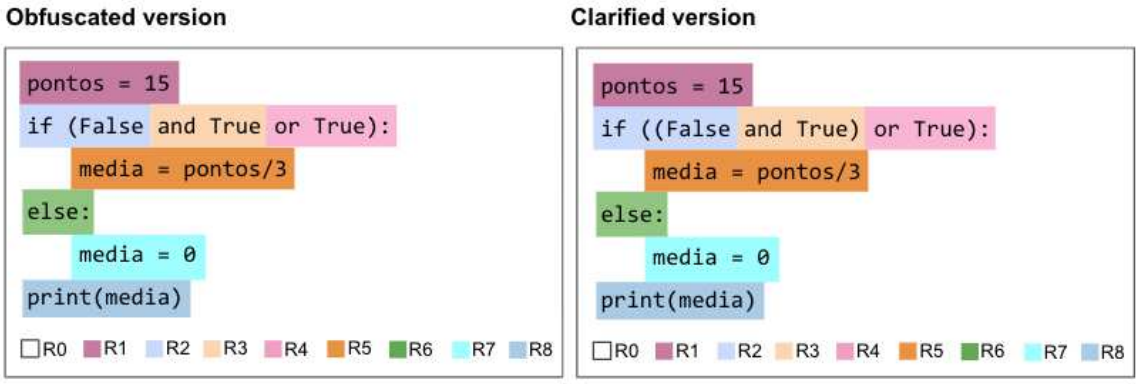



Figure 3.17: Set of regions inside the code version with *Operator Precedence* atom and in the code with the clarified version of code.

The subjects perceived the obfuscated version as more difficult to solve. In Figure 3.7(d), we observe clear differences for the *Operator Precedence*. In it, 34.4% of the subjects found the code with the obfuscated version difficult or very difficult to be solved while 12.5% found the code difficult in the clarified version. From the quantitative perspective, with clarified version, the subjects spent less time and have less visual effort in the AOI and the whole code. In addition, they make fewer attempts to solve the programs. Thus, we observe an alignment between their perception of difficulties and their objective performance.

In the obfuscated version, the subjects mainly mentioned the following issues: *"order of precedence"*, *"`if` difficult"*, *"misunderstood `if`"*, *"`if` difficult to validate"*, *" misunderstood `and` with `or`"*, *"I missed parentheses"*, *"precedence"*, *"order confusing"*, *"boolean difficult"*, *"didn't remember and or"*. In clarified version, they mentioned: *"confused `true`*

*and* `false`*"*, *"validate* `and` *with* `or` *difficult"*, *"validation complicated"*. As a takeaway, in the obfuscated version, sources of confusion are more concentrated in `if` condition, while in clarified version, still in `if` condition, however, less often.

> In our study, in the clarified version of code with *Operator Precedence*, there were reductions in the time in the AOI, attempts, duration of fixations, fixations count, and horizontal regressions count. In the obfuscated version, subjectively, sources of confusion are more concentrated in `if` condition, while in clarified version, still in `if` condition, however, less often.

### 3.3.5 Implicit Predicate

In Figure 3.18, we depict the obfuscating atom *Implicit Predicate* on the left-hand side, and the clarified version on the right-hand. The obfuscated version assumes that the expression in the condition of the `if` statement can be used as a predicate. The change in the clarified version consists of making the condition explicit, by adding a comparison with zero. With the clarified, we observed reductions in the time in the AOI by 34.1%, 16.9% in the attempts, 31.5% in the duration of the fixations, 32.5% in the fixations count, and 20% in the horizontal regressions count.

**Obfuscated**

```
if (elem % 5):
```

**Clarified**

```
if (elem % 5 != 0):
```

Figure 3.18: Obfuscating atom *Implicit Predicate* and clarified version.

The subjects perceive the obfuscated version as more difficult to solve. According to Figure 3.7(e), 50% of the subjects found the code with the obfuscated version difficult or very difficult to be solved while only 21.9% had the same opinion about the clarified version. From the quantitative perspective, with the clarified version, the subjects spend less time and have less visual effort in the AOI and in the whole code. Also, fixation duration, fixations count, and regressions count are reduced with the clarified version. The perception of the difficulties of the subjects is aligned with their objective performance.

This comparison was associated with reductions in horizontal regressions in the AOI, vertical regression in the code, and in the number of times needed to enter the AOI. The clarified version reduced the median number of horizontal and vertical regressions in the

code by 34% and 42% respectively. The reductions in the AOI reached 20%. In Figure 3.19, we give an example of these reductions with two subjects who examine the program of $SP_1$ containing the *Implicit Predicate*, one subject in each version. The subject seems to go back more times in the code to decipher the missing information with the implicit predicate. This is supported by the reduction in the number of times they exit from the AOI to the upper part of the code.

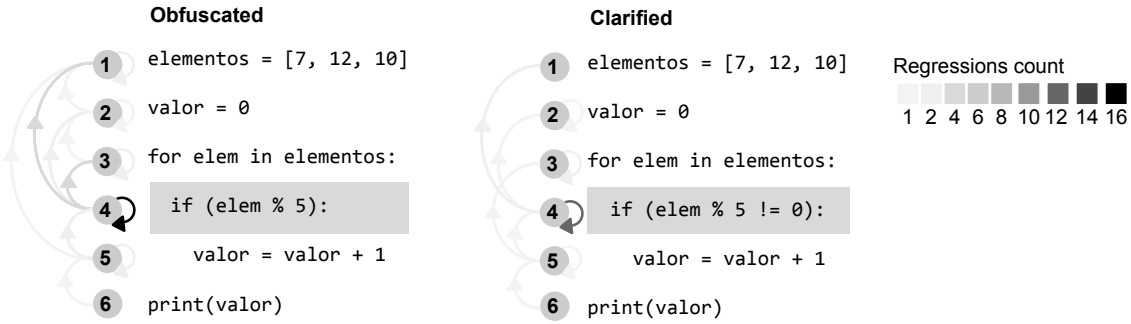



Figure 3.19: Two subjects visually regressing horizontally and vertically in the code, one with the obfuscated code containing the *Implicit Predicate* and the other with the clarified version of the code.

According to the program flow, in the obfuscated version, subjects are expected to evaluate the expression R4 → R5 in Figure 3.20 at most three times, once for each iteration in the loop. However, we observed that subjects make this transition 116% more times than expected, which may indicate more effort to understand. It is possible to arrive at the result by performing the R5 → R4 regression. However, subjects regress 68% more than expected. We observed a frequent back-and-forth. Furthermore, in obfuscated subjects look specifically at the region containing the module 489% more times than expected, which might indicate difficulty with this region. In the clarified version, adding the predicate explicitly splits the reading effort between three regions: R5, R6, and R7. With the explicit predicate, 50% of the subjects performed R5 → R6 → R7 linearly. The module region is seen on average 205% more than expected while the explicit predicate region R6, with 112% more than expected. Both versions demonstrated diverse reading patterns, but the addition of the explicit predicate divides the effort by making it lower compared to the implicit predicate.

In obfuscated version, the subjects mainly mentioned the following issues: *"`if` is complicated, "`if` is difficult", "% symbol confusing", "modulo", "% made it difficult", "diffi-*

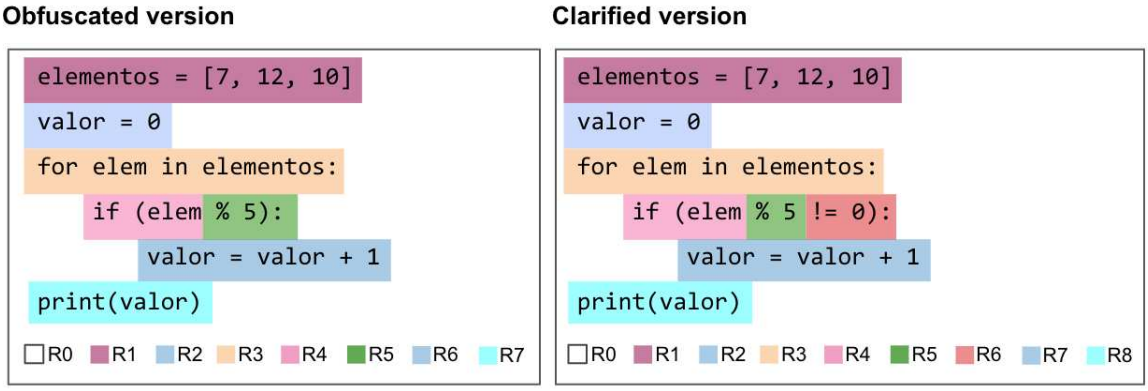



Figure 3.20: Set of regions inside the code version with *Implicit Predicate* atom and in the code with the clarified version of code.

*culties with modulo"*, *"unsure about % part"*, *"% 5"*, *"didn't understand %"*, *"*`if` *confusing"*, *"*`if` *hard to validate"*, *"difficult because of %"*. In clarified version, they mentioned: *"difficulties with % and !="*, *"modulo"*, *"*`while` *and* `if` *together is difficult"*, *"incrementing"*, *"*`!= 0` *confusing"*, *"counter in* `while` *difficult"*, *"confused %"*, *"*`if` *takes more time to understand"*. As a takeaway, in the obfuscated version, the sources of confusion are more concentrated in the condition of the `if` statement, while in the clarified version, still in the `if` statement but with more diverse sources. Along with the performance data, it indicates that both versions are similarly confusing.

> In our study, in the clarified version, we did not observe a significant impact on the metrics evaluated. In the obfuscated version, subjectively, the sources of confusion are more concentrated in the condition of the `if` statement, while in the clarified version, still in the `if` statement but with more diverse sources.

### 3.3.6 Augmented Operator

In Figure 3.21, we depict the *Augmented Operator*. This atom was considered easy to understand by most of the subjects. As indicated in Figure 3.7(f), for the obfuscated version, 6.3% of the subjects found the code difficult to be solved, while with the clarified version, the percentage was 3.1%. From the quantitative perspective, with the clarified version, the subjects spend less time and have less visual effort in the AOI and the whole code. The reductions with the clarified version in the fixation duration, fixations count, and regressions

count are noticeable, and the perception of difficulties of the subjects is aligned with their objective performance.

**Obfuscated**

```
valor *= 10
```

**Clarified**

```
valor = valor * 10
```

Figure 3.21: Obfuscating atom *Augmented Operator* and clarified version.

To make the expression shorter, the combination of the arithmetic operator with the assignment operator can confuse the subject about what receives the result of the operation. When the subjects have to pay more attention to the augmented operator or look more often at the assigned variable can give evidence of the effort in the understanding. In the obfuscated version, we observed that the region that contains the assignment symbol next to the operator with the assigned value is seen 51% more times than the same region in the clarified. Interestingly, to solve the program, only 43% of the subjects visited R5 in Figure 3.22 while in the obfuscated version, it was 93%. The structure without the augmented operator may alleviate the effort of looking at the same variable every time on the same line.

**Obfuscated version**

```
elementos = [20, 10, 50]
valor = 80
for elem in elementos:
    if (elem > 30):
        valor *= 10
print(valor)
```

R0 R1 R2 R3 R4 R5 R6 R7

**Clarified version**

```
elementos = [20, 10, 50]
valor = 80
for elem in elementos:
    if (elem > 30):
        valor = valor * 10
print(valor)
```

R0 R1 R2 R3 R4 R5 R6 R7

Figure 3.22: Set of regions inside the code version with *Augmented Operator* atom and in the code with the clarified version of code.

In obfuscated version, the subjects mainly mentioned the following issues: *"line with `total` is strange"*, *" `elem` variable and `limit`"*, *" `*=` symbol is strange"*, *"element variables is confusing"*, *"didn't recognize `*=` symbol"*, *"`for` and `elem` are difficult"*. In clarified version, they mentioned: *"`for` and `elem`"*, *"confused values"*, *"difficulties with `elem` variable"*, *"didn't understand the list"*, *"lost myself in the values"*. As a takeaway, in the

obfuscated version, the sources of confusion are more concentrated in the lack of knowledge of the $\star=$ symbol, while in the clarified version, the sources were concentrated in the values.

The clarified version aims to clarify the operation for updating an integer variable by repeating the same variable even if it becomes more verbose. It is worth remembering that, in Python, we can update a variable by adding and assigning (+=), multiplying and assigning ($\star=$), and using other operators. In our programs, we used these two operators. Thus, the clarified version has more elements to be observed by the subject. However, instead of increasing the visual effort, it is reduced, and the reduction in the number of regressions in the AOI is even more noticeable.

The clarified version in the two sets of programs reduced the median number of horizontal and vertical regressions in the code by 15% and 33%, respectively. In AOI, the obfuscated version presented twice the number of regressions. The subjects make fewer transitions between the AOI and the rest of the code in the clarified version containing the *Augmented Operator*. The subjects entered the AOI slightly fewer times with the clarified version than the obfuscated one. In Figure 3.23, we give an example of these reductions. In the obfuscated version, in the example, the transitions between the AOI and the rest of the code are more intense than in the clarified version. Most of the difference in the number of transitions occurs between AOI and the upper part of the code. Since we have a loop that iterates over three elements, we expect the number of entries and exits to be three. It seems that using syntactic sugar in the obfuscated version leads the subjects to turn to the upper part more times, at least for the multiply and assign operator, as reported by the subjects.

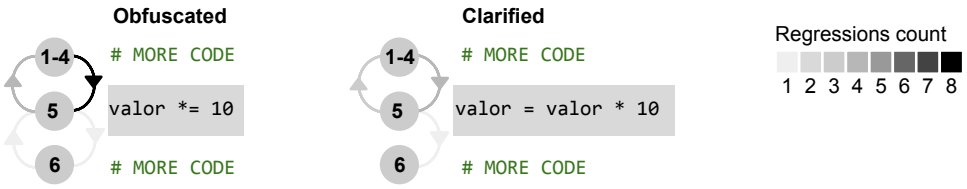



Figure 3.23: Two subjects visually entering and exiting the AOI in the code, one with the obfuscated code containing the *Augmented Operator* and the other with the clarified version of the code.

In our study, in the clarified version, we did not observe a significant impact on the metrics evaluated. In the obfuscated version, subjectively, the sources of confusion are more concentrated in the lack of knowledge of the $\star=$ symbol, while in the clarified version, the sources were concentrated in the values.

### 3.3.7 Coding Subjects' Answers

We used the method of grounded theory proposed by Strauss and Corbin [113] to analyze our qualitative data. A tentative explanation for most of the quantitative results is the presence of certain obfuscating atoms. However, by employing the grounded theory, we aim to understand and discuss whether we have qualitative evidence to support this theory in our study or whether the qualitative evidence reveals other alternative potential sources of confusion.

Grounded theory aims at coding and categorizing to describe a phenomenon found in the data, avoiding preconceived theories to focus on only the data. We used the following steps. First, during the interview, we ask questions to the subjects, break their answers into smaller chunks of data, and identify the major idea by assigning it a code that emerged from their answers. Thus, we perform coding in the first step. Second, we read all the codes and search for opportunities to group them into higher-level concepts. Third, we identify categories by discussing how similar the concepts were according to their properties. Fourth, we derive a theory through an inductive approach. All these steps can be seen in Table 3.3. We make all these steps available with more detail in our replication package [59].

| **Code Step** | **Concepts Step** | **Category Step** | **Theory Step** |
|---|---|---|---|
| Many assignments in one line complicates | Single line assignments | Code Style | Sources of Confusion |
| Got confused in indentation | Indentation | | |
| `if` inside loop is difficult | nesting structures | Control Flow | |
| Evaluating if takes longer | Conditional structure | | |
| Few iteractions over loop | Repetition structure | | |
| Not used to ternary | Knowledge of Idiom | Knowledge | |
| `for` in Pyhton in strange | Knowledge of Language | | |
| Simple calculations | Math calculation | Mathematics | |
| Difficulties with of the division | division calculation | | |
| Incrementing is difficult | Increment | Memory Load | |
| I got lost in the iteration | Iteration | | |
| Index of the loop confuses | Index of loop | | |
| Trouble in memorizing the variables | Memorization of Values | | |
| Too many variables | Amount of Variables | | |
| Difficulties in swapping the variables | Swapping Variables | | |
| Unnecessary variable | Temporary Variable | | |
| Difficulties with modulo and arithmetic operators | Combination of Operators | Operators | |
| Found modulo operator confusing | Arithmetic Operator | | |
| Found boolean operator complex | Relational Operator | | |
| Confused true and false | Logical Operators | | |
| Difficulties with Precedence | Operator Precedence | | |
| I had no difficulties | No difficulties | No difficulties | |

Table 3.3: Steps of the coding process of the subjects' answers.

In coding, we coded the answers of the subjects. We focused on synthesizing what phenomenon is described by the subject. Our codes ranged from a single word to short sequences of words. We had a set of 142 codes across 384 answers. Our codes included issues such as *"I was confused with if ternary"*, *"confusion with the rest of the division"*, and *"trouble in memorizing the variables' assignments"*. More details can be seen in Table 3.3.

We focused on abstracting, connecting, and grouping multiple codes in the concept step. We found 22 concepts that emerged from the codes. For instance, we grouped "trouble in memorizing the variables' assignments" and *"it was difficult to remember the value of the variable"* in the same underlying concept, which is *"memorization of values"*.

In the category step, we group the concepts into more abstract categories. Our concepts were included in seven broad categories: code style, control/repetition structures, knowledge, mathematics, memory load, operators, and no difficulties. Code style comprised concepts such as indentation or multiple assignments in the same line; control flow comprised both conditional structures, repetition structures, and their combination nested; knowledge comprised both knowledges of programming language and idiom. More details can be seen in Table 3.3.

In the theory step, a research question emerged from the data: what are the potential sources of confusion found in the data? Nevertheless, since the subjects reported mostly the difficulties found when examining the code, we expected to find a theory related to this.

As main lessons learned in our programs, we found various sources playing important roles in the confusion of the novices. Regarding the flow category, we found 31 codes related to the three concepts: conditional structures, repetition structures, and their combination. Most of the atoms we evaluated have control flow which can explain why this category is so broad. However, difficulties were associated more frequently with the obfuscated code versions in this category, especially in the atom *Conditional Operator*. For instance, we found six codes associating its obfuscated version with more complication, confusion, difficulties, and more time. Even though it did not affect the number of attempts, three subjects mentioned *"It takes longer to validate the* `if` *statement"*. Indeed, we observed that the subjects fixated on a longer duration in the AOI, which affects time, and presented more horizontal regressions.

Regarding the memory load category, we found 36 codes related to seven concepts, com-

prising memorization of values, iterating over loops, and swapping variables, among others. We carefully designed the tasks to have a few variables that iterate over a short list to avoid memory load effects. The clarified version of *Multiple Variable Assignments* was the main atom related to the concepts associated with the number of variables. The clarified version repeats one variable and breaks the assignment into two lines. The subjects mentioned that *"There are too many variables"* and we observed more visual horizontal regressions associated with these specific programs. We did not observe an impact in the number of attempts, but the clarified version might indicate that short-term memory can be affected.

Regarding the operators' category, we found 29 codes related to five concepts comprising arithmetic, relational, and logical operators, their combination, and precedence. The obfuscating atom *Operator Precedence* did not specify in which order the subjects should evaluate the expression. We had 21 answers related to difficulties in identifying the correct order or the ease of using parenthesis. Six subjects need two attempts to solve the programs and present more horizontal regressions. One subject that needed one more attempt mentioned *"I evaluated from left to right, then I realized that I should examine the operator* `AND` *first"*.

### 3.3.8 Other Analyses

**All atoms.** We analyze all atoms combined in Table 3.4. The idea of combining all atoms follows the Latin Square design methodology. We assign two subjects to each square. Each subject solves six programs obfuscated by the atoms from one set of programs, and six clarified programs from another set of programs. We can combine the atoms in two ways. We can perform combinations of the individual atoms, but not pairing the subjects in the squares, and we can combine the atoms by pairing the subjects. In the former, if we have a reduction in time in code for all six individual atoms, we will have a reduction across all atoms as well. However, in the latter one, which we used, we can better control for differences among the subjects and we may have slight increases in the combination. Combined, the differences were not so evident except for the number of horizontal and vertical regressions in the AOI clarified. The results revealed that, across all atoms, the average number of horizontal regressions reduced by 31.6% while the average number of vertical reduced from zero to 6.5. For *Multiple Variable Assignment*, *True or False Evaluation*, and *Conditional Expression*, we have shorter lines of code in the clarified version, however, for the *Operator Precedence*,

*Implicit Predicate*, and *Augmented Operator*, even with more elements, we observed reductions.

| | | In the AOI | | | | | In the Code | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **Atoms** | **Metrics** | O | C | PD % | PV | ES | O | C | PD % | PV | ES |
| All atoms | Time (sec) | 83.5 | 81.4 | ↓2.5 | 0.67 | n/a | 268.4 | 267.5 | ↓0.3 | 0.70 | n/a |
| | Attempts | n/a | n/a | n/a | n/a | n/a | 6.53 | 6.25 | ↓4.2 | 0.29 | n/a |
| | Fix. Duration (sec) | 50.9 | 45.8 | ↓10.0 | 0.32 | n/a | 124.5 | 128.1 | ↑2.8 | 0.77 | n/a |
| | Fix. Count | 143.5 | 127.5 | ↓11.4 | 0.46 | n/a | 367.0 | 392.5 | ↑6.9 | 0.86 | n/a |
| | Reg. Count | 39.5 | 34.5 | ↓12.6 | 0.35 | n/a | 159.0 | 164.0 | ↑3.1 | 0.89 | n/a |
| | Horiz. Reg. Count | 39.5 | 27.0 | ↓31.6 | 0.03 | – | 84.5 | 69.5 | ↑17.7 | 0.34 | n/a |
| | Vert. Reg. Count | 0.0 | 6.5 | ↑Inf | **3x10$^{-11}$** | – | 73.0 | 87.0 | ↑19.1 | 0.37 | n/a |

Table 3.4: Results for all metrics for all atoms. O = obfuscated code; C = clarified code; PD = percentage difference; PV = *p*-value; ES = effect size (Cliff's delta). Columns O and C are based on the median as a measure of central tendency, except for attempts, which are based on the mean.

In Table 3.5, we present each of the null-hypotheses with its respective confirmation or rejection. The two most sensitive metrics were the regressions count, which showed significant differences in three out of six atoms, followed by time, which showed differences in two atoms. The most noticeable effect size was observed for the code containing the *Operator Precedence* with respect to the clarified version in $RQ_2$ (Cliff's delta of -46). The other effects observed for clarified versions of the code containing the *Multiple Variable Assignment*, *True or False Evaluation*, and *Operator Precedence* were also noticeable, but to a smaller degree.

| **Atoms** | **Time** | **Attempts** | **Fix. Duration** | **Fix. Count** | **Reg. Count** |
|---|---|---|---|---|---|
| Multiple Var. Assignment | Reject | | | | Reject |
| True or False Evaluation | | | | | Reject |
| Operator Precedence | Reject | Reject | Reject | Reject | Reject |

Table 3.5: Summary of the rejection of the null hypothesis in isolated atoms in the AOIs. Null-Hypothesis consists of no difference between control and treatment groups with respect to the mentioned metric.

As shown in Table 3.5, we found consistent reductions in the time, number of attempts,

and visual metrics only for the clarified version of the code with *Operator Precedence* and partially increases the *Multiple Variable Assignment*. However, we did not observe such differences for the other four atoms in our study, which might indicate that both versions in their cases are similarly confusing or similarly understandable. According to Gopstein et al. [48], measuring time and answer correctness, in the C language, the atoms *Assignment as Value*, *Conditional Operator*, *Operator Precedence*, and *Implicit Predicate* cause confusion. However, our study added a new perspective of visual metrics, which complements time and answer correctness. The new atom added in our study, the *True or False Evaluation*, did not affect time and number of attempts consistently, such as *Operator Precedence*. However, we found evidence for the reductions in the number of visual regressions within the line of code containing the atom. We need more studies with other demographic groups and more atoms.

Possible explanations for the lack of consistent differences across our metrics can be because we varied the programming language or the demographic group. For instance, the syntax for the *Conditional Operator* in the C language includes the symbol "?:", which led to 31% more errors than `if` statements. In the syntax of Python, the *Conditional Operator* includes the test of a condition in a single line with the if-else. In Python syntax, we did not observe an impact in the number of attempts, which can be easier to understand than C's syntax. These differences can shed light on the differences in the results. In addition, since we focused on novices, we had to resort to simple programs, which can also explain the lack of differences in the number of attempts.

**Interaction of the atoms.** We tested the interaction of the atoms by defining them as independent variables with the six levels. We found statistically significant differences in time spent in the AOI for the obfuscated ($p$-value ¡ $8x10^{-11}$) and clarified ($p$-value $1x10^{-12}$) versions, according to atom types. The post-hoc test revealed that the time the subjects spent in the AOI obfuscated by *Multiple Variable Assignment* was significantly lower than in the AOIs of the other atoms, except for *Augmented Operator*. Similarly, the time in the *Multiple Variable Assignment* clarified was significantly lower than in the AOIs of the other atoms except for *Operator Precedence* and *Implicit Predicate*. The time in the AOI of the code obfuscated by *Augmented Operator* was significantly lower than in the other atoms, except for *Multiple Variable Assignment*, while in the clarified, the time was lower than any other AOI evaluated. In addition, for the clarified version, the time in the AOI with *Condition Expres-*

*sion* was higher than any other atom. These results indicate more difficulties with *Condition Expression* and fewer difficulties with *Multiple Variable Assignment* and *Augmented Operator*. We performed the same analysis on fixation duration, fixations count, and regressions count and observed similar results.

Concerning the number of attempts, we found statistically significant differences for the obfuscated ($p$-value ¡ 0.003) but not for the clarified version, according to atom types. The *Operator Precedence* with the obfuscated version presented a significantly lower number of attempts than every other atom. These results indicate more difficulties with *Operator Precedence*.

**Sets of programs.** As an additional analysis, we compared the two distinct programs with the same atoms instantiated, trying to find possible differences. The clarified versions presented the most relevant differences for the *Operator Precedence*. For instance, the novices spent 41% less time in the AOI and 33% fewer regressions examining `if True or (True and False)` compared to `if (False and True) or False`. The explanation might be that, since the priority is at the beginning of the expression, it can help the novices to make sense of it faster and not need to go back many times. For the *Implicit Predicate*, we observed the the novices spent 39% less and regressed 42% fewer times in `if (elem % 5 != 0)` compared to `if (elem % 4 != 0)`. A possible explanation might be that it is easier to compare with odd numbers since the rest is different from zero.

For the *Augmented Operator*, the novices spent 55% less time in the AOI, fixated 58% less, and made 200% fewer regressions examining `total += 1` compared to `valor *= 10`. A possible explanation might be that "+=" operator is seen more frequently than "*=" for novices or that multiplication might be more complex and requires more memory than the sum operation. For the clarified version, the novices spent 40% less time in the AOI, fixated 51% less, and made 50% fewer regressions examining `valor = valor * 10` compared to `total = total + 1`.

## 3.4 Threats to Validity

Here we describe potential issues and threats to the validity of our study: internal validity (Section 3.4.1), external validity (Section 3.4.2), and construct validity (Section 3.4.3).

### 3.4.1 Internal Validity

We performed the experiment in four different locations to gather more subjects and have a variety of subjects from distinct higher-education institutions. However, different locations may influence the visual attention of the subjects. We carefully arranged the rooms to have similar conditions to mitigate this threat. For instance, they were quiet rooms with minimum distractions, similar temperatures, and artificial light sources. In future work, we aim to keep track of which subject performed over which location so we can bind possible differences.

Despite our best efforts, the presence of a researcher in the room may have unintentionally influenced the visual attention or performance of the subjects since they were aware of being observed. To mitigate this threat, we put effort into letting the subjects feel comfortable with the researcher's presence. In addition, we avoided any interaction with the subjects while they were examining the programs so that they could be concentrated.

The eye tracker equipment has limitations, such as calibrating the eyes of the subjects. We carefully calibrate and even re-calibrate when necessary. However, we still saw a need for an adjustment in the gaze points. We adopted the following strategy. We selected programs with a long horizontal line of code, specifically with lines that the subjects mentioned they were looking at in the interviews. Then we systematically analyzed whether the fixations plotted and heatmaps were on white spaces close to that long line, which could suggest a need for an adjustment. For certain subjects, the heatmap revealed a red color over a blank area not touching the code. Similarly, the plot of the fixation points sometimes revealed points over blank areas. A small adjustment was sufficient for these cases to get the data corrected. The error for these cases was systematic, meaning all the fixations for a particular program received the same adjustment. The adjustment in the points influences its interpretation. In recurring meetings, we discussed these adjustments by going systematically through the data for each subject. Thus, to mitigate the threat of working with uncalibrated equipment we generated another threat. However, we decided that the threat of adjusting the points would be preferable to analyze the data with points not touching the code, given the uncalibrated equipment. It is important to mention that the median number of pixels used to correct the fixations in $y$-coordinate was 30 pixels, which translated to 0.6 lines of inaccuracy on the screen, and the maximum value was 80 pixels. For the $x$-coordinate, following this strategy, we did not need to adjust the $x$-coordinate. We made the generated fixations and adjustment

strategy available in our replication package [59].

In pilot studies, we observed that a swivel chair could impair data collection by the camera or negatively affect the captured data. To mitigate this threat, we used chairs without swiveling capability in all rooms we used to conduct the experiment.

The total time we allocated for each subject was one hour and assigned them 12 programs, which may have influenced the visual effort. To minimize this threat, we designed simple and short programs with only one atom instantiated and put a time limit of two minutes. Given the simplicity of the programs, most subjects solved them before the time limit. Since our programs consist of non-minimal snippets, in the sense that they do not contain only the atom region, the extra lines of the code might introduce working memory as a confounding factor. However, with eye tracking, it is possible to measure and compare time and visual effort only in the atom region. In addition, both programs, in obfuscated and clarified versions with the same atom, had the same extra lines of code to make the comparison fair.

All the subjects had the option to keep making attempts until getting the correct output. We then compared the number of attempts until they answered correctly. However, following this strategy, if one makes wrong attempts, she could make more fixations or even longer ones, with more regressions. Alternatively, we collected the eye tracking data separately for each attempt made so that we could compare only the first one. We performed an analysis based only on the data from the first attempt, but we found similar results.

Using the Latin Square design, we blocked the set of programs to control noise. Besides performing combinations of the programs in the squares, we analyzed the programs with the atoms individually. The analysis of individual programs in the set of programs violates the design. The extent of such violation does not have an estimated impact. However, to better understand the effects of the atoms, analyzing them combined and individually can give a more nuanced and complete understanding of their effects.

### 3.4.2 External Validity

We resorted to small programs with less than 10 lines of code aiming at fitting the code onto the screen. This approach may restrict generalization to larger programs. However, previous work on the same subject has resorted to code snippets with a similar number of lines [48; 84]. If we find differences in small code snippets, we expect that larger snippets may tend

to show greater differences. Nevertheless, we need to conduct other studies with larger code snippets to provide empirical evidence for those expectations.

In our study, we focused on novices in Python. Thus, we cannot generalize our results to more experienced developers in Python. Novices have also been the subject of other eye tracking studies on code comprehension [15]. In the future, we intend to explore the same topic of this study with experienced developers.

Since we have focused on atoms in Python programming language, we cannot generalize our findings to other programming languages. To mitigate this threat, in our programs, we used constructions that are common in other languages, and most of our subjects reported some experience with other languages. In addition, since our subjects were Brazilian Portuguese native speakers, our programs were designed to contain identifiers in their mother tongue.

Our programs were designed to have only one output, which was a numeric value. All subjects had to solve the task by specifying the correct output aloud after reading the code. The results for this type of task may not generalize to other types, such as finding a bug, fixing a syntax problem, or adding a feature. In addition, since the font style may influence the attention of the subject, to minimize a possible threat, we consistently used the same font style and size for the programs, no syntax highlighting, and no bold font.

The number of atoms instantiated in a program may influence the performance and visual effort of the subjects. To minimize threats related to the number of atoms, we consistently used only one atom in each program.

### 3.4.3 Construct Validity

Time and answer correctness metrics are often employed to assess the phenomenon of code comprehension [95; 70] and in particular, to investigate atoms of confusion [48; 33]. With respect to eye tracking methodology, other studies have employed similar metrics to measure visual-related aspects [77; 103; 7]. Other works have combined time, answer correctness, and visual effort [102; 33; 30]. In particular, fixation duration and fixations count have been used as a measure of visual effort [102; 8]. According to Sharafi et al. [99], metrics based on saccades, such as number of saccades or saccades duration are metrics whose definitions are identical to the ones based on fixations. Thus, we decided to explore eye movement

regressions since they have been explored and associated with visual effort [99].

Inviting people to participate in eye tracking studies may influence the decisions of the subjects regarding their visual behavior. For instance, we have to make them aware that their eyes are being tracked, which may influence where or how much they look at some regions of the code. To minimize this threat, we did not make the subjects aware of the precise goals of the study to avoid hypothesis guessing.

## 3.5 Conclusions

In this chapter, we report on a controlled experiment with eye tracking to evaluate the impact of six atoms of confusion on code comprehension. We evaluated to what extent the obfuscated code containing atoms of confusion and the functionally equivalent clarified versions of the code impacted the time, number of attempts, and visual effort of 32 novices in Python.

Our results revealed impacts to a considerable extent with the evaluated atoms in the code. The clarified version of the code containing the *Operator Precedence* reduced the time in the AOI and in the entire code by 38.6% and 20.1%, respectively. The fixation duration in the AOI, the fixations count, and the regressions count reduced to the extent of 34.1%, 32.3%, and 50%, respectively. The clarified version of the code containing the *True or False Evaluation* particularly reduced the regressions count in the AOI to the extent of 47.3%. However, the clarified version of the code containing the atom *Multiple Variable Assignment* increased the time in the AOI and the regressions count by 30.1% and 60%, respectively. With respect to the number of attempts, the *Operator Precedence* reduced by 28.3% in the number of attempts. We found reductions in the time in the AOI, the fixation duration, the fixations count, and the regressions count with the clarified version of the code containing the *Conditional Expression*, *Implicit Predicate*, and *Augmented Operator*, however, to a lesser extent. In general, we observed a substantial impact of the obfuscated and clarified code on the subjects' abilities to understand the code even in small and simple programs. Hence, with larger and more complex code snippets, we expect the impact to be even greater. However, we need to conduct more studies. In addition, the fact that it was still possible to detect some differences even using more meaningful names than in previous studies supports the evidence of the impact of these atoms.

With eye tracking, we investigated how much time the subjects spent in a specific region of the code that contained the atoms of confusion and their clarifying versions, to what extent the atoms impacted the fixation duration, fixations count, and regressions count, and how the atoms impacted the way the subjects read the code. Thus, our findings contribute with some relevant implications. For the education community, our study contributes with raising concerns regarding teaching methods that may hinder code comprehension for Python novices. Educators should be careful when preparing the teaching material for introductory courses, avoiding using code snippets with atoms that can confuse the novices. For instance, the subjects in our study needed 28.3% more attempts to solve the code containing the *Operator Precedence*, which associated with a negative impact on their abilities to understand the code. For Python novices, the positive impact of most of the clarified versions of the code containing the atoms in the time in the AOI and fixation duration, fixations count, and regressions count may indicate improvements in their productivity, understanding, and visual effort.

For the research community, our study setup exploring the visual effort dimension contributes with 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 nuances and allow us to dive deeper into how this atom impacts difficulty beyond visual effort. This can be a future direction for research. For practitioners and for language designers, the use of syntactic sugar in the language syntax has to be done considering whether the pattern will impair the novices' abilities to understand the code. Some languages have abolished constructs because they can create obstacles for novices. For example, C-style `for` loops were

removed from Swift [92] because, among other things, they offer *"a steep learning curve from users arriving from non C-like languages"*.

In future work, we aim to evaluate other types of atoms proposed by Gopstein et al. [48]. Cedrim et al. [19] studied the influence of refactorings on code smells and found that their majority are neutral, and some refactorings even lead to new smells in the code. We aim to explore this topic with the perspective of eye tracking. In addition, we aim at conducting more experiments with experienced developers in Python, with a larger number of subjects, explore other programming languages, programs with variables names that have no meaning at all, other types of tasks such as finding a bug and investigate a higher number of atoms instantiated in a single task. Finally, we intend to add other eye tracking metrics based on saccades, blink rate, pupil dilation, and explore code reading patterns based on the gaze transitions. As future work, we also envision proposing heuristics or building a model whereby a programmer receives an arbitrary source code to read, and we use eye tracking data to extract which elements were atom candidates or at least confusing regions.

# Chapter 4

# Study II: Extract Method Refactoring

In this chapter, we present a controlled study to evaluate the impact of the Extract Method on code comprehension from the perspective of the eye tracking metrics with novices in Java. It is worth mentioning that this study was conducted after the experiment evaluating atoms of confusion and included other subjects. This chapter is organized as follows: Section 4.1 presents the study definition, and Section 4.2 presents the study methodology. Section 4.3 presents the obtained results, and Section 4.4 discusses a qualitative interview with the novices. Section 4.5 discusses the threats to validity, and Section 4.6 presents the conclusion.

## 4.1 Study Definition

In this section, we present the definition of our study according to the Goal-Question-Metrics approach [5]. **We compare** programs with Inline Method with functionally equivalent Extract Method version programs **for the purpose of** understanding how inlining and extracting methods associate with improvements **with respect to** code comprehension **from the point of view of** novices in the Java programming language **in the context of** tasks adapted from introductory programming courses.

We address five research questions (RQs). For each RQ, our null hypothesis is that there is no difference between the Inline Method and Extract Method versions of the programs with respect to the collected metric.

**$RQ_1$: To what extent does the Extract Method refactoring affect task completion time?**

Following prior studies [48; 33], 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 Extract Method refactoring 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 Extract Method refactoring affect fixation duration?** Fixations with increased duration have been associated with more attention to the stimuli [16]. To answer this question, we measure the duration of each fixation in the programs.

**$RQ_4$: To what extent does the Extract Method refactoring affect fixation count?** An increased number of fixations has been associated with more time to understand code phrases [8], more attention to complicated code [28], and more visual effort to recall identifiers' names [101]. To answer this question, we count the number of fixations in the programs.

**$RQ_5$: To what extent does the Extract Method refactoring affect regressions count?** When a reader does not understand what she reads in natural language, she makes eye regressions [91]. Likewise, the regression rate has been used for programming tasks to measure the linearity of code reading [15]. In imperative programming languages, developers may read code lines following the fashion left-to-right, top-to-bottom, similarly to natural language, except for loops, which require the reader to read bottom-to-top at some points. To answer this question, we compute the number of regressive eye movements with a direction opposed to the writing system, right-to-left, and bottom-to-top. To make the comparison fair, both Inline and Extract Method versions have loops that iterate over the same number of elements.

## 4.2 Methodology

In this section, we present our methodology. We present the pilot study (Section 4.2.1), experiment phases (Section 4.2.2), subjects (Section 4.2.3), treatments (Section 4.2.4), evaluated refactorings (Section 4.2.5), programs (Section 4.2.6), eye tracking system (Section 4.2.7), fixation and saccades instrumentation (Section 4.2.8), and finally the analysis

(Section 4.2.9).

### 4.2.1 Pilot Study

Before the actual experiment, we conducted pilot studies with six human subjects. We aimed to refine our experiment material and evaluate the experiment setup and design. We do not consider these subjects in the analysis of the results.

Our experiment material includes programs, a subject characterization form, and a questionnaire for a semi-structured interview. To evaluate our programs, we tested complete code snippets from introductory programming courses with distinct levels of difficulty. We tested the code font size, font style, line spacing, and indentation. We also evaluated the questions from the form and questionnaire.

Our subjects comprise native Brazilians. We designed the vocabulary of the programs to be in Brazilian Portuguese to avoid obstacles in language comprehension. The identifiers and methods' names were carefully selected and discussed by the researchers. We used names such as `result` and `counter` for receiving the result of operations and counting; in addition, we used the abbreviated word `num` to designate a number.

For methods names, we used lessons learned from previous studies and guidelines [40; 3; 17; 68; 4; 53]. They systematize the use of capitalization, size of identifiers, number of words, the meaningfulness of name, use of verbs, among other aspects. We used methods named to show their intention, which were selected to the best of our abilities. This approach is arguably closer to a practical scenario. In addition, we refined the names of the methods in the pilot study, testing how capable the names were in showing the intention of the method. We discussed the methods' names to find the most appropriate ones.

After experiment refinement, we organized it into five phases: (1) Characterization, (2) Tutorial, (3) Warm-up, (4) Task, and (5) Qualitative Interview. We estimated an average of 60 minutes for each subject to complete all phases. Next, we describe these phases in detail.

### 4.2.2 Experiment Phases

In phase one, as the subject enters the room, we explain the study, what data we will capture, and how we aim to use the data. Each subject voluntarily fills out a consent form, agreeing

to participate and being aware that their identity was anonymous. Then they filled out a characterization form with questions related to programming experience.

In phase two, we present a tutorial explaining the execution of the experiment. We instruct the subjects on how to sit properly in front of the eye tracker and how to perform the tasks. We then proceed with a calibration of the camera on the subject's eyes. During the camera calibration, each subject should look at specific locations on the screen that the camera software indicates. The camera software also reports when the calibration is successful.

In phase three, each subject warms up for the experiment by solving a simpler program. During the warming up, we demonstrate how to specify the output out loud, instructing them to close their eyes for two seconds before and after solving the program and how we signal the correct and incorrect answer. After warming up, the subjects should be more comfortable with the experiment setup and equipment.

In phase four, we run the actual experiment with eight programs, four with the Inline Method version and four with the Extract Method version. To avoid learning effects, we use a Latin Square design [12], which we explain in more detail in Section 4.2.4.

In phase five, we end the experiment with a semi-structured interview. We investigate how the subjects examined the programs and what were their subjective impressions. We go through each program and ask three questions: (1) How difficult was it to find the output: very easy, easy, neutral, difficult, or very difficult? (2) How did you find the output? (3) What were the difficulties you had, if any?

We started running the experiment after the end of the social distancing measures in the country when the coronavirus infections were decreasing, and the number of vaccinated people was increasing. For the safety of everyone involved, all the researchers had hand sanitizers and face masks. We limited the number of people in the environment to only one subject at a time.

We arranged the environment of the experiment to reduce noise in the data. We used a fixed chair which increased the precision of the eye tracker equipment in pilot studies. However, given the camera limitations, obtaining perfect data is virtually impossible. To mitigate it, we as researchers plotted, discussed, and performed data correction by slightly shifting chunks of fixations in the $y$-axis. We discuss this strategy in detail in the threats to validity section (see Section 4.5.1). A replication package of the experiment material is

available on our website [108].

### 4.2.3 Subjects

We recruited 32 undergraduate students that we refer as "novices". They reported having 30 months of experience with programming languages in general, which mainly included Java, Python, JavaScript, C, and C++. However, only in Java, they had on average 12 months of experience. They were recruited from two distinct universities in two cities in Brazil, invited mainly in person or through text messages. They were Brazilian Portuguese speakers enrolled in academic universities.

We computed the minimal number of subjects necessary to have a minimal power of 0.8 with a significant level of 0.05 using the T-test sample size computation. We found that we need 26 subjects in two samples to have a minimal power of 0.8 with a significant level of 0.05. Alternatively, because we have 32 subjects, our study can also detect a moderate effect size of 0.5 with a power of 0.5 with a significant level of 0.05.

### 4.2.4 Treatments

Each subject examined eight programs ($P_1$–$P_8$). To avoid learning effects, we designed 16 distinct programs divided into two Sets of Programs ($SP_1$ and $SP_2$). One subject examines four programs with Inline Method (I) of the set $SP_1$, and four programs with Extract Method (E) of set $SP_2$, as seen in Figure 4.1. Another subject examines four programs with Extract Method version (E) of the set $SP_1$, and four programs with Inline Method version (I) of set $SP_2$.

Being in the same set $SP_1$ but with different refactorings, both programs $P_1$ print the same output. Both programs are examined by distinct subjects as well. We designed the programs with Inline Method version to be our baseline group, and the ones with the Extract Method version to be the treatment group. In all the programs, the subject has the task of specifying the correct output but without multiple answer options. For instance, given the input in the program, the subject has the task to calculate the factorial, calculate the next prime, among others.

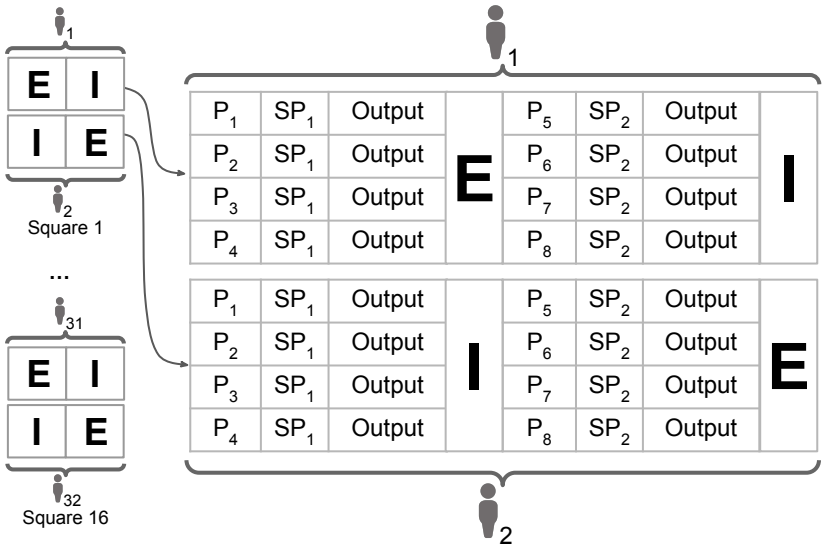



Figure 4.1: Structure of the experiment in terms of experimental units divided into 16 squares. Subject$_1$ takes four programs ($P_1$–$P_4$) with the Extract Method (E) of *Sum Numbers* ($P_1$), *Calculate Next Prime* ($P_2$), *Return Highest Grade* ($P_3$), *Calculate Factorial* ($P_4$). These programs are from set of programs 1 ($SP_1$). Subject$_1$ also takes four programs ($P_5$–$P_8$) from the set of programs 2 ($SP_2$) comprising the Inline Method (I) of *Count Multiples of Three* ($P_5$), and *Calculate Area of Square* ($P_6$), *Check If Even* ($P_7$), and *Count Number of Digits* ($P_8$). Subject$_2$ takes the complement to that. "Output" describes a task in which the subject has to specify the correct output.

### 4.2.5 Evaluated Refactorings

We selected and evaluated two refactorings, namely Extract Method and Inline Method. These refactorings are among the most common refactorings used in practice [19; 80; 107]. To perform the Extract Method refactoring, the following mechanics can be used: the developer creates a new method, and names it after the intention of the method. Then she copies the extracted code from the source method into the new target method [40].

To perform the Inline Method refactoring, the following mechanics can be used: the developer finds a call to the method and replaces it with the content of the method, and then deletes the method [40]. Inline Method is essentially the opposite of the Extract Method. It may vary from one line to multiple lines of code. The main motivation reported by developers to apply the Inline Method is to eliminate unnecessary or too trivial methods [107].

### 4.2.6 Programs

We selected code snippets by manually analyzing code repositories of introductory programming assignments. We selected and designed programs that varied in size and complexity so that we could have a better understanding of code comprehension. We included simple and small programs, such as to calculate an area of a square, and more complex ones, such as to compute the next prime. We present our programs for each refactoring in Figure 4.2. The main sources of tasks were GeekForGeeks[1] and Leetcode[2], which are popular for learning and practicing programming. We selected assignments with small and complete code snippets which we adapted for camera constraints. In the experiment, for each program, a subject had to specify the correct output in an open-ended fashion, meaning that no answer options were provided. Providing information about the code, such as finding the output, is a methodology employed by 70% of the studies in the domain of code comprehension [84].

The evaluated programs had 6–18 lines of code, with a median number of 12 lines. We restricted the number of lines to fit completely on the screen. It is worth mentioning that having programs that fit completely on the screen is beneficial due to the fatigue caused by the use of eye tracking setup. All the used programs were free of syntactic errors. The programs followed Consolas font style, font size 11, line spacing of 1.5 inches, and eight

[1] https://www.geeksforgeeks.org/

[2] https://leetcode.com/


Figure 4.2: Programs evaluated in our study: *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*. Shaded areas represent the AOIs, which are the code lines in which both inlined and extracted versions differ.

white spaces of indentation. Even though written in the Java language, we used simple constructions that commonly occur in many programming languages. We made sure that each program with the inlined method contained exactly one method in its scope, the Java main method. Each program with the extracted method contained exactly two methods, the Java main method, and the extracted method. Both inlined and extracted versions of the same task, presented the same output. However, due to the use of a Latin Square design, no subject was exposed to inlined and extracted versions of the same task. To bring diversity to the programs, we used different styles such as assigning the methods' call to a variable and printing the variable or calling the methods in the print statement.

### 4.2.7 Eye Tracking System

We used the Tobii Eye Tracker 4C in our experiment which has a sample rate of 90 Hz. The calibration of the eye tracker followed the standard procedure of the device driver with five points. The eye tracker was mounted on a laptop screen with a resolution of 1366 x 720 pixels, a width of 30.9 cm, and a height of 17.4 cm, at a distance of 50-60 cm from the subject. We displayed the code tasks as an image in the full-screen mode, but no Integrated Development Environment (IDE) was used, nor number for the lines. We computed an accuracy error of 0.7 degrees which translates to 0.6 lines of inaccuracy on the screen, considering the font size we used and the line spacing. We tested the line spacing in the pilot study. We designed it to be sufficiently large so we could overcome the eye tracker accuracy limitations. For processing the gaze data, we implemented a script in Python, which allowed us to analyze and collect the metrics.

### 4.2.8 Fixation and Saccades Instrumentation

Fixation can be understood as the stabilization of the eye on part of a visual stimulus for a period of time, and the rapid eye movements between two fixations are called saccades [93; 55]. As we fixate our eyes, we trigger cognitive processes [60]. In the code comprehension scenario, source code can work as a visual stimulus over which the subject performs the task of reading to specify the output.

In the literature, there is no standardized threshold of time for a fixation because it de-

pends on the processing demands of the task. However, popular guidelines among eye tracking researchers indicate a threshold between 100 and 300 ms [93; 91]. Thus, after analyzing our programs, we used 200 ms as our threshold.

We used a Dispersion-Based algorithm to classify the fixations. Particularly, we used the Dispersion-Threshold Identification (I-DT) [93]. As its parameters, we classified gaze samples as belonging to a fixation if the samples are located within a spatial region of approximately 0.5 degrees [81], which corresponded to 25 pixels in our screen.

### 4.2.9 Analysis of the Results

From a total of 256 possible observations (32 subjects $\times$ 8 programs), the subjects solved a set of 248 (96.9%) of them. This set includes programs that were solved either in the first submission or after many submissions. We imputed missing data for eight (3.1%) programs using the Multivariate Imputation by Chained Equations (MICE) method. This method is available in the Mice R package for multiple imputations namely Predictive Mean Matching (PMM). The PMM method uses the predictive mean matching [58] and performs better when the data sample size is sufficiently large [64], which was our case.

We performed statistical analysis to test the null hypotheses of our RQs using a significance level of 0.05. It means that we have a 5% risk of finding a difference when there is no actual difference. For $p$-values equal or inferior to 0.05, we reject the null hypothesis that there was no difference between the median of the treatments.

We used Shapiro-Wilk Test [97] to test the distribution of the data. When normally distributed, we verify whether the variances of the two groups compared were equal [104]. Then we performed the parametric $t$ test for the two independent samples to verify whether there is a statistically significant difference between the two groups [104; 100]. When the data do not follow a normal distribution and we could not normalize it, we used the non-parametric test Mann-Whitney, also known as the Wilcoxon test, which can be applied to these specific situations [104; 100]. In the scenario of fixations, the mean value might not be appropriate since it can be dependent on some very high values [43]. In our analysis, we used the median as a measure of central tendency.

## 4.3 Results

In Sections 4.3.1–4.3.5, we address our research questions. In these sections, when we mention a statistically significant difference, it means we rejected the null hypothesis for the mentioned metric.

### 4.3.1 $RQ_1$: To what extent does the Extract Method refactoring affect task completion time?

We consider two measures of time, the time the subjects spent examining the AOI and the time examining the complete code. The AOI comprises only the lines of code in which both versions differ (method extracted) and provide a finer-grained analysis than the whole code. They are shown in Table 4.1, columns "In AOI" and "In Code", respectively. With the Extract Method refactoring, the subjects spent 78.8% less time in the AOI to calculate the *Factorial* and 70% less time to determine the *Highest Grade* of a list. On the other hand, the subjects spent 146.2% more time in the AOI to *Sum Numbers* from one to N, 166.9% more time to calculate the *Area of Square*, and 108.4% to analyze a number and *Check If Even*. We observed statistically significant differences for these cases.

In the complete code, the task to calculate the *Factorial* required 72.8% less time. On the other hand, the tasks *Sum Numbers*, *Area of Square*, and *Check if Even* presented increases in time by 93.9%, 94.4%, and 28.3, respectively. We also perform combinations of the time spent from both perspectives. The idea of combining all programs follows the Latin Square design [12]. Each subject in each square solves four programs with the Extract Method and four with Inline Method. We then combine the time for all programs with the Extract Method and compare it with the Inline Method. However, this combination did not present statistically significant differences.

We used Cliff's Delta [25] to yield the effect size. The effect size of 0.2 suggests a small effect, 0.5 a medium effect, and 0.8 a large effect [26]. The negative sign implies that the values of the treatment group (Extracted Method versions) are greater than the control group (Inlined Method version). For the time, the effects varied from -0.47 to 0.93 meaning that the impact on those tasks was noticeable.

Table 4.1: Results for **time spent in AOI and in Code** ($RQ_1$). I = Inline Method; E = Extract Method; PD = percentage difference; PV = *p*-value; ES = Cliff's Delta effect size. Columns I and E are based on the median as a measure of central tendency.

| **Tasks** | **In AOI** | | | | | **In Code** | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | I sec | E sec | PD % | PV | ES | I sec | E sec | PD % | PV | ES |
| Sum Numbers | 8.8 | 21.6 | ↑146.2 | **0.0001** | 0.75 | 15.9 | 30.9 | ↑93.9 | **0.005** | 0.57 |
| Next Prime | 121.2 | 53.9 | ↓55.5 | 0.06 | n/a | 132.6 | 61.6 | ↓53.5 | 0.06 | n/a |
| Highest Grade | 77.7 | 23.7 | ↓70.0 | **0.01** | -0.50 | 92.6 | 32.3 | ↓65.0 | 0.09 | n/a |
| Factorial | 62.2 | 13.1 | ↓78.8 | **0.02** | -0.47 | 81.3 | 22.1 | ↓72.8 | **0.01** | -0.51 |
| Multiples of Three | 24.6 | 37.5 | ↑52.4 | 0.22 | n/a | 39.2 | 49.0 | ↑24.9 | 0.18 | n/a |
| Area of Square | 2.5 | 6.9 | ↑166.9 | **0.0000** | 0.93 | 7.7 | 14.9 | ↑94.4 | **0.008** | 0.53 |
| Check If Even | 4.7 | 9.8 | ↑108.4 | **0.0009** | 0.66 | 28.3 | 36.3 | ↑28.3 | **0.03** | 0.42 |
| Number of Digits | 34.5 | 26.0 | ↓24.7 | 0.25 | n/a | 66.4 | 38.2 | ↓42.4 | 0.12 | n/a |
| All Programs | 127.5 | 111.2 | ↓12.8 | 0.21 | n/a | 191.8 | 177.5 | ↓7.4 | 0.14 | n/a |

**Finding 1:** In our study, the subjects exhibit a reduction in the time spent in the AOI with the Extract Method refactoring to determine the *Highest Grade* and to calculate the *Factorial*. However, for the tasks to *Sum Numbers* from one to N, to calculate the *Area of Square*, and *Check if Even*, the time in the AOI increased.

### 4.3.2 $RQ_2$: To what extent does the Extract Method refactoring affect the number of attempts?

Since the programs are somewhat simple, the median number of answer submissions does not provide useful information. Thus, we decided to use the mean instead. As seen in Table 4.2, with the Extract Method refactoring, the average number of submissions until getting the task solved reduced by 34.4% to determine the *Highest Grade* of a list, followed by 20% to determine the *Multiples of Three*, and 23.8% to calculate the *Number of Digits*. We observed statistically significant differences for these cases. The combination presented a reduction by 8.9% in the number of answer submissions, which was also significantly different.

Table 4.2: Results for **number of attempts of the answers** ($RQ_2$). I = Inline Method; E = Extract Method; PD = percentage difference; PV = *p*-value; ES = Cliff's Delta effect size. Columns I and E are based on the mean as a measure of central tendency.

| **Tasks** | **Submissions** | | | | |
|---|---|---|---|---|---|
| | I | E | PD % | PV | ES |
| Sum Numbers | 1.00 | 1.31 | ↑31.2 | 0.07 | n/a |
| Next Prime | 1.44 | 1.44 | 0.0 | n/a | n/a |
| Highest Grade | 1.81 | 1.19 | ↓34.4 | **0.01** | -0.42 |
| Factorial | 1.56 | 1.31 | ↓16.0 | 0.30 | n/a |
| Multiples of Three | 1.25 | 1.00 | ↓20.0 | **0.03** | -0.25 |
| Area of Square | 1.00 | 1.00 | 0.0 | n/a | n/a |
| Check If Even | 1.06 | 1.25 | ↑17.6 | 0.15 | n/a |
| Number of Digits | 1.31 | 1.00 | ↓23.8 | **0.03** | -0.25 |
| All Programs | 5.2 | 4.7 | ↓8.9 | **0.03** | -0.27 |

**Finding 2:** In our study, the subjects submit fewer answers to solve the tasks to determine the *Highest Grade* from a list, the *Multiples of Three*, and the *Number of Digits*.

### 4.3.3 $RQ_3$: To what extent does the Extract Method refactoring affect fixation duration?

For the visual metrics, we also distinguish between fixations in the AOI and in the complete code. With the Extract Method refactoring, the fixation duration was reduced by 78.9% in the AOI to calculate the *Factorial*, followed by 73.6% to determine the *Highest Grade*. On the other hand, the fixation duration increased by 130.1% in the AOI to *Sum Numbers*, followed by 121.1% to calculate the *Area of Square*, and 73.1% to *Check If Even*. These cases presented statistically significant differences.

In the complete code, the fixation duration was reduced by 74.5% to calculate the *Factorial*, followed by 65.8% to determine the *Highest Grade*, and 55.9% to determine the *Number of Digits*, and 53% to determine the *Next Prime*. On the other hand, the fixation duration increased by 110.5% to *Sum Numbers*. The combination did not present statistically significant

differences.

Table 4.3: Results for **fixation duration in AOI and in Code** ($RQ_3$). I = Inline Method; E = Extract Method; PD = percentage difference; PV = *p*-value; ES = Cliff's Delta effect size. Columns I and E are based on the median as a measure of central tendency.

| **Tasks** | **In AOI** | | | | | **In Code** | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | I sec | E sec | PD % | PV | ES | I sec | E sec | PD % | PV | ES |
| Sum Numbers | 6.2 | 14.2 | ↑130.1 | **0.002** | 0.60 | 8.7 | 18.4 | ↑110.5 | **0.007** | 0.53 |
| Next Prime | 63.7 | 35.3 | ↓44.5 | 0.11 | n/a | 65.6 | 30.2 | ↓53.9 | **0.04** | -0.39 |
| Highest Grade | 45.1 | 11.8 | ↓73.6 | **0.01** | -0.42 | 47.1 | 16.3 | ↓65.8 | **0.04** | -0.53 |
| Factorial | 40.2 | 8.4 | ↓78.9 | **0.003** | -0.64 | 46.0 | 11.7 | ↓74.5 | **0.03** | -0.51 |
| Multiples of Three | 11.7 | 19.9 | ↑69.1 | 0.38 | n/a | 18.1 | 28.9 | ↑60.0 | 0.29 | n/a |
| Area of Square | 1.4 | 3.0 | ↑121.1 | **0.001** | 0.65 | 4.1 | 6.5 | ↑59.8 | 0.08 | n/a |
| Check If Even | 2.9 | 5.0 | ↑73.1 | **0.02** | 0.46 | 14.5 | 19.6 | ↑35.1 | 0.06 | n/a |
| Number of Digits | 21.6 | 14.5 | ↓32.7 | 0.21 | n/a | 40.5 | 17.8 | ↓55.9 | **0.04** | -0.38 |
| All Programs | 70.9 | 59.2 | ↓16.4 | 0.18 | n/a | 102.4 | 84.5 | ↓17.4 | 0.09 | n/a |

**Finding 3:** In our study, the subjects exhibit a reduction in the fixation duration in the AOI with the Extract Method refactoring to determine the *Highest Grade* and to calculate the *Factorial*. However, for the tasks to *Sum Numbers* from one to N, to calculate the *Area of Square*, and *Check if Even*, the fixation duration in the AOI increased.

### 4.3.4 $RQ_4$: To what extent does the Extract Method refactoring affect fixations count?

With the Extract Method refactoring, the fixations count was reduced by 75.8% in the AOI to calculate the *Factorial*, followed by 67.7% to determine the *Highest Grade*. However, the fixations count increased by 194.2% in the AOI to *Sum Numbers*, followed by 138.8% to calculate the *Area of Square*, and 137.1% to *Check if Even*. For all these cases, we observed statistically significant differences.

In the complete code, the fixations count was reduced by 74.3% to calculate the *Factorial*, followed by 69.8% to determine the *Highest Grade*. However, the fixations count increased

by 102.1% to *Sum Numbers* and 44.8% to *Check if Even*. The combination did not present statistically significant differences.

Table 4.4: Results for **fixations count in AOI and in Code** ($RQ_4$). I = Inline Method; E = Extract Method; PD = percentage difference; PV = *p*-value; ES = Cliff's Delta effect size. Columns I and E are based on the median as a measure of central tendency.

| **Tasks** | **In AOI** | | | | | **In Code** | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | I | E | PD % | PV | ES | I | E | PD % | PV | ES |
| Sum Numbers | 17.5 | 51.5 | ↑194.2 | **0.004** | 0.62 | 23.5 | 47.5 | ↑102.1 | **0.003** | 0.56 |
| Next Prime | 186.0 | 108.0 | ↓41.9 | 0.19 | n/a | 191.5 | 111.0 | ↓42.0 | 0.17 | n/a |
| Highest Grade | 141.5 | 45.6 | ↓67.7 | **0.008** | -0.53 | 166.0 | 50.0 | ↓69.8 | **0.008** | -0.51 |
| Factorial | 141.0 | 34.0 | ↓75.8 | **0.02** | -0.44 | 156.0 | 40.0 | ↓74.3 | **0.03** | -0.40 |
| Multiples of Three | 38.5 | 62.5 | ↑62.3 | 0.37 | n/a | 47.5 | 80.5 | ↑69.4 | 0.26 | n/a |
| Area of Square | 4.6 | 11.0 | ↑138.8 | **0.004** | 0.59 | 12.0 | 20.0 | ↑66.6 | 0.07 | n/a |
| Check If Even | 8.5 | 20.1 | ↑137.1 | **0.003** | 0.58 | 39.0 | 56.5 | ↑44.8 | **0.04** | 0.42 |
| Number of Digits | 96.4 | 49.0 | ↓49.2 | 0.16 | n/a | 99.0 | 49.0 | ↓50.5 | 0.06 | n/a |
| All Programs | 252.0 | 189.5 | ↓24.8 | 0.26 | n/a | 288.0 | 282.5 | ↓1.9 | 0.24 | n/a |

**Finding 4:** In our study, the subjects exhibit a reduction in the fixations count in the AOI with the Extract Method refactoring to determine the *Highest Grade* and to calculate the *Factorial*. However, for the tasks to *Sum Numbers* from one to N, to calculate the *Area of Square*, and to *Check if Even*, the fixations count in the AOI increased.

### 4.3.5 $RQ_5$: To what extent does the Extract Method refactoring affect regressions count?

With the Extract Method refactoring, the regressions count was reduced by 84.6% in the AOI to calculate the *Factorial*, followed by 74.4% to determine the *Highest Grade*. However, the regressions count increased by 200% in the AOI to *Check if Even*, followed by 108.3% to *Sum Numbers*. For all these cases, we observed statistically significant differences.

In the complete code, the regressions count was reduced by 78% to calculate the *Factorial*, followed by 74.6% to determine the *Highest Grade*. However, the regressions count

increased by 89.4% to *Sum Numbers*. The combination did not present statistically significant differences.

Table 4.5: Results for **regressions count in AOI and in Code** ($RQ_5$). I = Inline Method; E = Extract Method; PD = percentage difference; PV = *p*-value; ES = Cliff's Delta effect size. Columns I and E are based on the median as a measure of central tendency.

| **Tasks** | **In AOI** | | | | | **In Code** | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | I | E | PD % | PV | ES | I | E | PD % | PV | ES |
| Sum Numbers | 6.0 | 12.5 | ↑108.3 | **0.005** | 0.57 | 9.5 | 18.0 | ↑89.4 | **0.01** | 0.52 |
| Next Prime | 80.5 | 41.0 | ↓49.0 | 0.09 | n/a | 84.5 | 44.0 | ↓47.9 | 0.08 | n/a |
| Highest Grade | 43.0 | 11.0 | ↓74.4 | **0.01** | -0.50 | 75.0 | 19.0 | ↓74.6 | **0.01** | -0.53 |
| Factorial | 52.0 | 8.0 | ↓84.6 | **0.003** | -0.67 | 70.5 | 15.5 | ↓78.0 | **0.01** | -0.49 |
| Multiples of Three | 13.5 | 19.0 | ↑40.7 | 0.39 | n/a | 21.5 | 28.5 | ↑32.5 | 0.53 | n/a |
| Area of Square | 1.0 | 2.0 | ↑100.0 | 0.05 | n/a | 5.0 | 7.0 | ↑40.0 | 0.10 | n/a |
| Check If Even | 1.0 | 3.0 | ↑200.0 | **0.03** | 0.43 | 18.5 | 24.5 | ↑32.4 | 0.14 | n/a |
| Number of Digits | 21.0 | 13.0 | ↓38.0 | 0.14 | n/a | 46.0 | 21.5 | ↓53.2 | 0.06 | n/a |
| All Programs | 75.0 | 54.0 | ↓28.0 | 0.10 | n/a | 125.5 | 114.0 | ↓9.16 | 0.09 | n/a |

**Finding 5:** In our study, the subjects exhibit a reduction in the regressions count in the AOI with the Extract Method refactoring to determine the *Highest Grade* and to calculate the *Factorial*. However, for the tasks to *Sum Numbers* from one to N and to *Check if Even*, the regressions count in the AOI increased.

## 4.4 Discussion

In this section, we discuss the interview with the subjects (Section 4.4.1), the perception of difficulties of the programs (Section 4.4.2), the positive impact of the Extract Method refactoring (Section 4.4.3), the negative impact of the Extract Method refactoring (Section 4.4.4), and a discussion on gaze transitions and heatmaps (Section 4.4.5).

### 4.4.1 Interview with the subjects

To analyze the qualitative data, aiming to support our quantitative results, we used the method of grounded theory proposed by Strauss and Corbin [113]. Grounded theory describes a phenomenon found in the data without preconceived theories. We used the following steps. First, we conduct a semi-structured interview with the subjects, break their answers into smaller chunks of data, and assign each chunk a code that emerges from it. We perform the coding in the first step. Second, we analyze all the codes and search for opportunities to group them into higher-level concepts. Third, we group similar concepts searching for opportunities to form higher-level categories. Fourth, we derive a theory through an inductive approach. All these steps are in Table 4.6 and are available in our material [108].

| **Code Step** | **Concept Step** | **Category Step** | **Theory Step** |
|---|---|---|---|
| Didn't pay attention to the function | Lack of attention | Attention | Code comprehension factors |
| Didn't see the equals operator | Missed details | | |
| Familiarity with factorial | Knowledge of Domain | Knowledge | |
| Not used to this iteration | Knowledge of Iterator | | |
| Confused the variable type | Knowledge of Language | | |
| Difficulties with modulo | Decision structure | Control Flow | |
| Difficulties with the repetition | Loop structure | | |
| Loop with decision is difficult | Nesting | | |
| Going back and forth takes time | Reading flow | | |
| Had to read line by line | Linear reading | | |
| The name helped to infer | Meaningfulness | Names | |
| Didn't understand the name | Lack of clearness | | |
| Confused in the multiplication | Math operation | Mathematics | |
| Confused the order of the operations | Precedence order | | |
| Difficulties in remembering the values | short-term memory | Memory load | |
| No difficulties | No difficulties | No difficulties | |

Table 4.6: Coding process of the subjects' answers.

In the code step, we found a set of 35 codes across 256 answers. The issues included

"Didn't pay attention to the function", "Difficulties with the repetition", and "The name helped to infer". We added more details in Table 4.6 and the complete process in our supplementary material [108].

In the concept step, we connected and grouped multiple codes. We found 16 concepts that emerged from the codes. For instance, we grouped "Names were suggestive" and "The name helped infer" in the same underlying concept, which is *"Meaningfulness"*.

In the category step, seven main categories emerged from the concepts, namely, "Attention", "Knowledge", "Control Flow", "Names", "Mathematics", "Memory load", and "No difficulties". In the theory step, a broad question emerged: what are the factors related to code comprehension found in the data? We did not set a research question a priori. Since the subjects reported mostly their difficulties and strategies when examining the code, we expected to find a theory related to this.

As the main lessons learned, we found a variety of factors playing important roles in the code comprehension of the subjects. Most of the subjects' answers (44) were related to the presence or the lack of suggestive methods' names. While interviewing the subjects, we observed a trend related to names in the data and asked the subjects why or how the names were important. The subjects mentioned that the names were suggestive and helped them infer the code behavior. The methods' names helped them to formulate a hypothesis about what the method does. One subject mentioned *"The name gives me a hint, but I check if the code does what it says"*. Another subject said *"To solve the task, I used only the name of the method and the input, otherwise it would take longer"*. Thus, the extracted method seemed to favor an up-bottom fashion code reading with hypothesis formulation and test of the hypothesis.

Another lesson was that the subjects tend to check if the method does what it says. While in three tasks, the subjects mentioned not needing to check the method, in 22 tasks, the subjects checked the method. Some of the reasons were *"Checked the code to see if the body matches the name"*, *"You cannot always trust the code others wrote"*, and *"To check if there is some kind of trick"*. Associating this feedback with eye tracking metrics, the strategy of formulating and testing hypothesis seemed to reduce time and visual effort.

On the hand, inlining the method favored a bottom-up fashion reading with no previous formulation of a hypothesis. In the category of Control Flow, we have concepts of decision

and repetition structures, with nesting and linear reading behaviors. In them, the subjects mentioned having difficulties with 31 tasks of which 25 had no extracted method. One subject mentioned "*Since I had no method to help me, I had to read the code line by line*". A more careful code examination may lead to more time and visual effort since the subjects have to fixate on more parts of the code.

The subjects reported difficulties in the category of Mathematics, namely math operations and precedence order. From 24 tasks, 16 had no method extracted. Since no hypotheses could be inferred from a name, the subjects had to formulate it from a more careful code examination, which seemed to be hampered by the operations. The most common difficulties were *"confusion in the math operation"*, particularly in the task to count the number of digits, in both extracted and inlined versions. This might explain the lack of statistical differences in the time and visual effort.

### 4.4.2 Subjective perception of difficulties

To investigate the perceptions of the subjects on the difficulties of the programs, we used a five-point scale. The subjects rated each program individually, whether they found it very easy, easy, neutral, difficult, or very difficult to solve. We compare their perceived difficulties between Inline and Extract Method in Figure 4.3.

Overall, the subjects perceived the tasks with the Extract Method easier to solve than those with the Inline Method. For the tasks to determine the *Highest Grade*, to calculate the *Factorial*, and to determine the *Number of Digits*, the differences were more evident.

### 4.4.3 A positive impact of the Extract Method refactoring

The subjects perceived the Extract Method versions of the four tasks *Next Prime*, *Highest Grade*, *Factorial*, *Multiples of Three*, and *Number of Digits* easier to solve in terms of difficulty, according to Figure 4.3(b), (c), (d), (e), and (h). With the Extract Method, the perception of difficulties (difficult or very difficult) for the *Next Prime* reduced from 18.7% to 12.5%, for the *Highest Grade* reduced from 12.5% to zero, for the *Factorial* reduced from 25% to 6.2%, and for the *Number of Digits* reduced from 31.2% to 6.2%. The perception for *Multiples of Three* remained the same. From the quantitative perspective, with

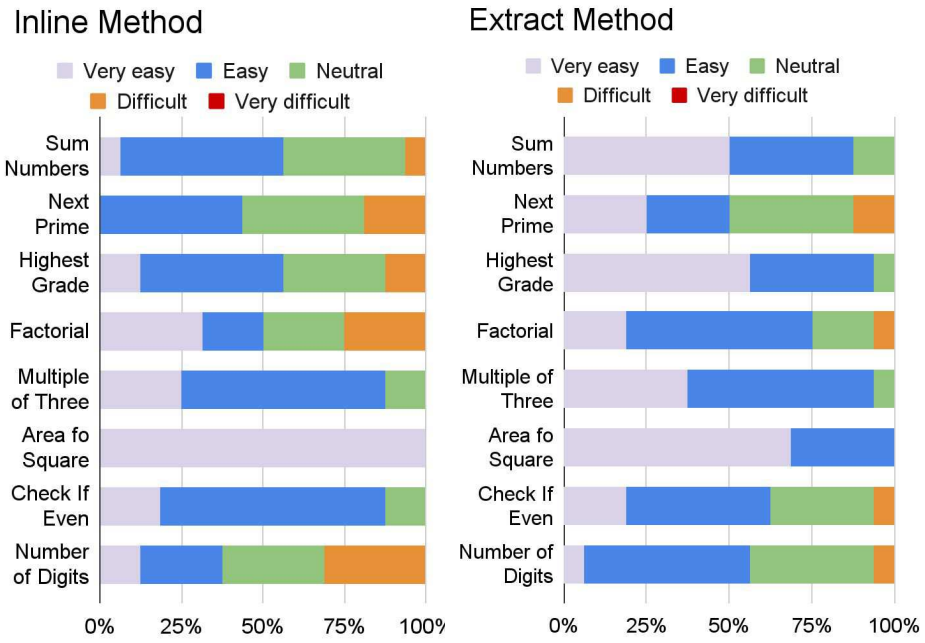



Figure 4.3: Perception of difficulties of the subjects with Inline Method and Extract Method of the tasks.

the Extract Method, the subjects needed from 70% to 78.8% more time, from 20% to 34.4% fewer submissions, fixation duration reduced from 73.6% to 78.9%, fixations count reduced from 67.7% to 75.8%, and regressions count reduced from 74.4% to 84.6%. It indicates an agreement between how subjects subjectively perceive the difficulty of the task and their performance on it. The tasks *Multiples of Three* and *Number of Digits* presented significant reductions in the number of attempts while *Number of Digits* and *Next Prime* in the fixation duration in the code.

It is important to mention that the obtained results are conservative and differences could potentially be positively higher for larger methods. On the other hand, we only examined the uses of the Extract Method in simple scenarios. One might use the Extract Method refactoring in more complex methods, usually in scenarios where a comment would need to explain part of that method. We discuss the threats related to this in Section 4.5.2.

In the tasks with the Inline Method, the subjects mainly mentioned the issues: *"difficulties with* `for` *loop and if together"*, *"trouble with control variable"*, *"confusion with iterator"*, *"math operation"*, *"integer division"*, and *"modulo"*. In the tasks with Extract

Method, the subjects mainly mentioned: *"name helped but checked the method"*,*"didn't remember factorial"* and *"difficulties with the math involved"*, *"integer division"*, *"suggestive name"*, and *"method checking"*. As a takeaway, in the Inline Method version, comprehension effort concentrates more on lower-level specific constructs, such as conditional and repetition structure, modulo, and math operations, while in the Extract Method version, similar difficulties are reported but it concentrates more on higher level components, such as the names, methods, and remembering concepts such as what a factorial does.

### 4.4.4 A negative impact of the Extract Method refactoring

The subjects perceived the Extract Method version of the task *Sum Numbers* easier to solve, while the tasks to calculate *Area of Square*, and *Check If Even* were less easy to solve, according to Figure 4.3. With the Extract Method, the perception of difficulties (difficult or very difficult) for *Sum Numbers* reduced from 6.2% to zero, for the *Area of Square*, the perception was the same, while for the *Check If Even*, it increased from zero to 6.2%. From the quantitative perspective, the subjects needed from 108.4% to 166.9% more time with the Extract Method, fixation duration increased from 73.1% to 130.1%, fixations count increased from 137.1% to 194.2%, and regressions count increased from 100% to 200%. It indicates a disagreement between how subjects perceive the difficulty of the task and their performance on the task to *Sum Numbers*.

In general, these tasks suggest that the Extracted Method impairs the subjects' performance. One possible explanation might be that, for small programs, extracting a method adds more elements and lines to be observed, which can influence the visual attention and effort of the subjects. In the Extracted Method versions, the attention of the subjects is redirected to another location of the code, because of the change in the control flow. In that way, extracting a method can influence where the subjects fixate their attention, which can lead to more effort.

In the tasks with Inline Method, the subjects mainly mentioned the issues: *"difficulties with multiplication"*, and *"order of operations"*, *"attention to parenthesis"*, *"attention to `for` loop"*. In the tasks with Extract Method, the subjects mainly mentioned: *"difficulties with multiplication"*, and *"order of operations"*, *"attention to `for` loop"*, *"suggestive name"*, and *"method easy to check"*. As a takeaway, in the Inline Method version, com-

prehension effort concentrates mainly on math operations and repetition structure. In the Extract Method version, the subjects presented similar difficulties. Even though the subjects reported that the name was suggestive, they analyze the method's body to check whether it matches the name.

### 4.4.5 Gaze Transitions and Heatmaps

To identify patterns in the eye gaze transitions in our data, we defined each line of code as a small region that could be analyzed independently. These regions were defined in pixels on the images of the tasks according to a previous guideline [55]. Their positioning was precisely defined considering the camera limitations so that we could have a margin between the regions, and the regions did not overlap. In addition, we defined the white-space as a region so that we could be aware of any threat to validity given the camera limitations. Using the chronological order of the fixations and their positions, we identified a sequence of visited regions for each subject. We then built a big picture of the sequences by simplifying repeated transitions to go from one region to the same region. We make the sequences and the images of the tasks with the regions identified available in our supplementary material [108].

With the method extracted, the code has more lines, and more elements to observe and go back into the code. To better understand how this impacts the visual effort, we distinguished between a regression to a previous line, or vertical regression, and a regression within the same line, or horizontal regression. With the Extract Method, the subjects make fewer regressions horizontally and vertically in the tasks *Next Prime*, *Highest Grade*, and *Factorial* tasks. It implies that the subject can solve the tasks with the Extract Method without going back visually in the code so often as in the Inline Method. For *Multiples of Three*, and *Number of Digits*, we did not observe significant differences.

#### Highest Grade

In Figure 4.4, we built two graphs to depict the distribution of the regressions for two subjects who examine a program to determine the *Highest Grade*. In the example, we selected two subjects whose individual results for the time, attempts, and visual metrics are coherent with the median results for all subjects. One subject examined the program with the inlined

version and the other examined the extracted method version. In the graph, each edge represents a regression with a direction to a previous line of code or to the same line (self-loop edge). Each node represents a line of code. The grayscale intensity of the edge represents the number of times such regression was repeated.

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 body of the method.

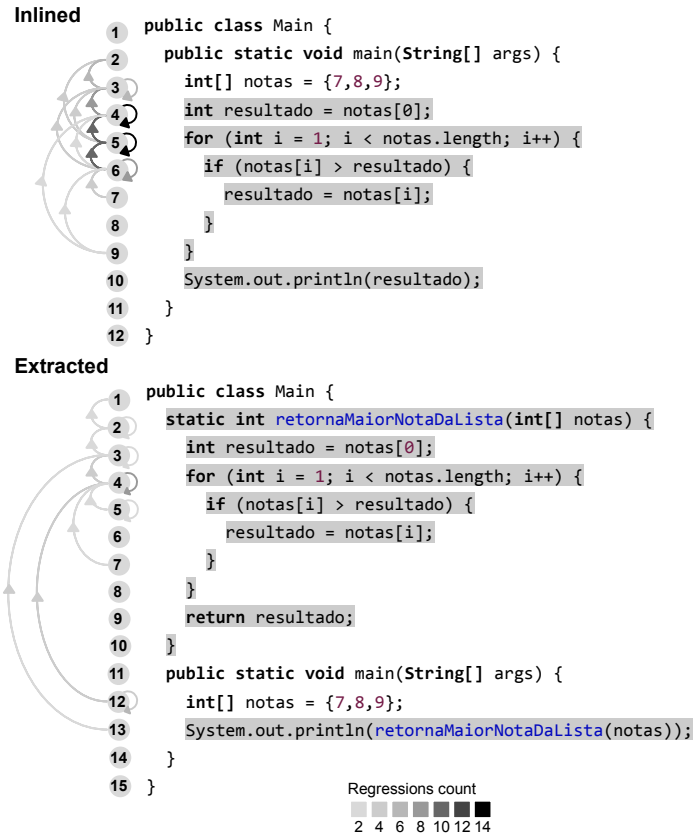



Figure 4.4: Eye movement regressions for the inlined and extracted method versions to determine the *Highest Grade*.

### Factorial

Consider the task to determine the *Factorial* in Figure 4.5. In the inlined version, the subjects make 91 switches between distinct regions on average, while in the extracted version, with the addition of three lines, the subjects make 59 switches, a 54% reduction. A common sequence for both versions is to go back and forth between the `for` control statement and the iterator inside the control, R5 and R6 (inlined), and R4 and R5 (extracted). For instance, in the inlined version, 100% of subjects do R5→R6 on average 11 times, and 93% do R6→R5 going back on average 12.8 times. Even though the loop only iterates four times, the subjects visually regress three times the expected. In the extracted version, 81% of subjects do R4→R5 on average 8 times, and 66% do R5→R4 back on average 10 times. In addition, the subjects examine R3 in the inlined version, where the variable is assigned, on average, 68% more times than R10 in the extracted version. Providing the subject with an idea of what the code is about, the subjects seem to check the calculation in the method's body with less visual effort.

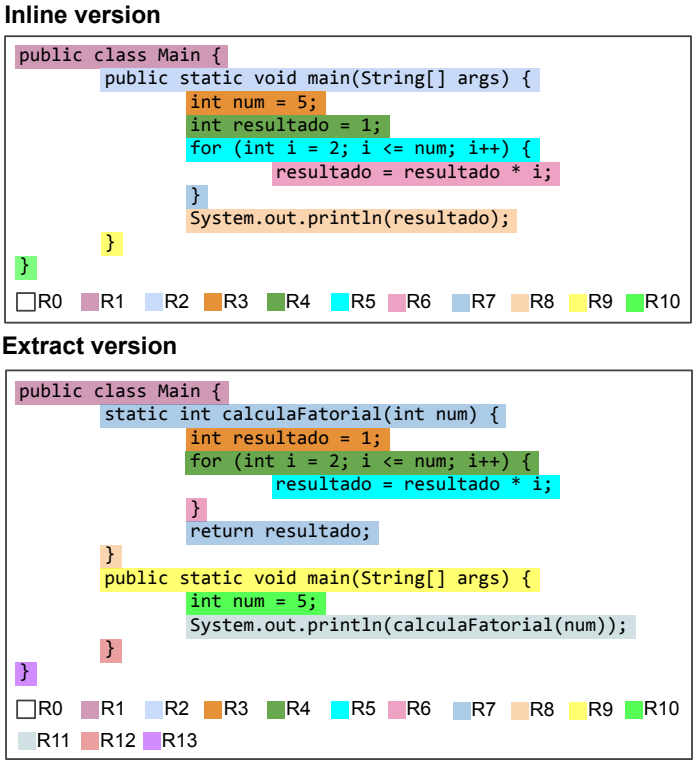



Figure 4.5: Set of regions of the inlined and extracted method code versions to determine the *Factorial* of a number.

Extracting the method was associated with an easier interpretation of the calculation that takes place within the method and this is reflected in the 16% reduction in the number of attempts. Five subjects mentioned difficulties with the multiplication operation in the inlined version, either by accumulating the iteration value or by the repetition structure itself, and two could not solve it. One of them exhibited repetitive visual behavior R5→R6 and R6→R5 for 37 times reporting inattentiveness in the loop stop condition. In the inlined version, three subjects mentioned that the name helped. However, domain knowledge is important. Two subjects reported difficulty for not knowing or remembering what a factorial would be in the extracted, which seemed to affect their visual effort. For instance, one of them who gave up the task exhibited the behavior of looking R10→R9 four times going up to the R3 or R4 method. When the subject has domain knowledge associating the name, extracted version led to less effort to compute the result.

### Multiples of Three

Consider the task to count the *Multiples of Three* in Figure 4.6. In the inlined version, the subjects make 38 switches between distinct regions on average, while in the extracted version, with the addition of three lines, the subjects make 46 switches, an increase of 27%. A common sequence for both versions is to go back and forth between the `for` control statement and the decision control, R5 and R6 (inlined), and R4 and R5 (extracted). For instance, in the inlined version, 93% of subjects do R5→R6 an average of 3.2 times and 87% do R6→R5 coming back an average of 2.7 times. In the extracted version, 87% of subjects do R4→R5 an average of 2.7 times and 75% do R5→R4 returning an average of 2.9 times. As the loop has three iterations, the round-trip average is close to what was expected in both versions. However, some differences were noticeable. For instance, in extracted version, subjects resort less to the array. The frequency with which they examined the array region in inlined is on average 7.5 times, while in extracted, 5.7, that is, 24% less. In inlined, 50% of the time, the subjects go back from the R5 or R6 regions. In extracted version, 50% of the time, the subjects go from the R11 or R13 region.

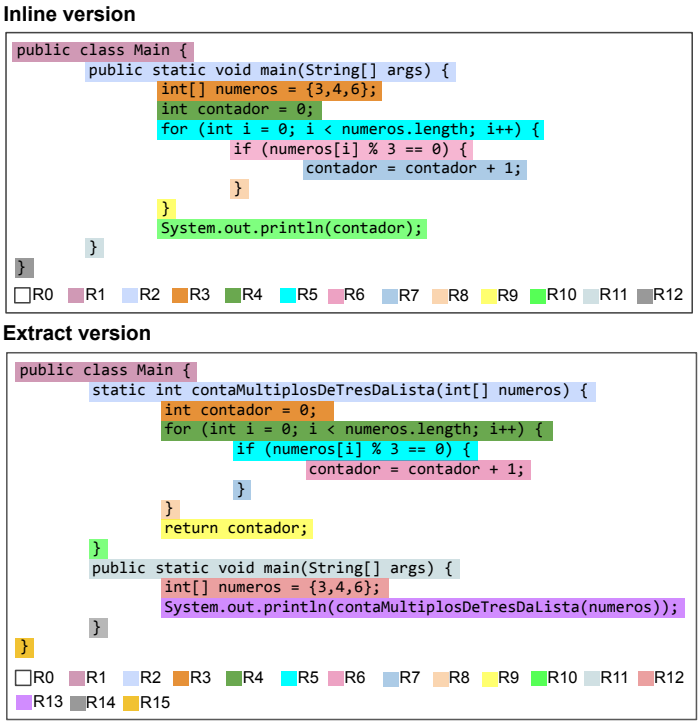



Figure 4.6: Set of regions of the inlined and extracted method code versions to count the *Multiples of Three* of a list.

The subjects' responses can help us understand this behavior. For instance, in inlined, two subjects mentioned "*difficulties with examining line by line because there is no function*", while in extracted, 12 subjects mentioned that the name helped them to infer what the answer was and 9 of them mentioned checking the function to see if it was doing what they inferred by the name. One subject mentioned: "*The name gave me a hint by I checked to see if the method's body corresponded to the name*". We observed a reduction in the number of attempts with extracted by 20%. In four subjects who needed more attempts in the inlined, they examined the array 13 times, majoritarily, returning from R6 where they check if it is divisible by three.

With the Extract Method, the subjects make more regressions horizontally and vertically in the code in both tasks. The horizontal regressions may relate to longer lines because of the methods' names. Specifically, in the task to *Sum Numbers* from one to N, subjects tend to go back and forth inside the method to check the math operation involving multiplication. The vertical regressions in the tasks may relate to going back and forth between caller and method.

### Sum Numbers

In Figure 4.7, we depict heatmaps for two programs, one with the inlined version and the other with the extracted version. Heatmaps provide a visual representation of the eye fixation data. The intensity of color varies according to the number of fixations, and their duration and heatmaps are useful for a big picture of visual attention. In the inlined version, the color is less intense, meaning fewer fixations with less duration. In the extracted version, two regions are more intense. The subjects make 16 switches between distinct regions on average, while in the extracted version, the subjects make 34 switches, which is an increase of 112%.

**Inline version**

```
public class Main {
        public static void main(String[] args) {
                int num = 3;
                int resultado = (num * (num + 1)) / 2;
                System.out.println(resultado);
        }
}
```

**Extract version**

```
public class Main {
        static int somaNumerosDeUmAteNum(int num) {
                return (num * (num + 1)) / 2;
        }
        public static void main(String[] args) {
                int num = 3;
                int resultado = somaNumerosDeUmAteNum(num);
                System.out.println(resultado);
        }
}
```

Figure 4.7: Heatmap of the inlined and extracted method code versions of *Sum Numbers* from one to N.

Three subjects mentioned difficulties in the order of arithmetic operations, while one mentioned difficulties associating an operation in return. A subject that took four tries to resolve mentioned the return as difficult. When dealing with an unknown formula, the inlined version may be better because it is more direct instead of taking this calculation to another region of the code and returning the result through a return. Not everyone is familiar with

the formula for the sum of the first $n$ positive integers. We designed that way to evaluate whether it gives different results compared to a more familiar formula, such as calculating the area of the square. In both cases, the impact was negative for the time and visual effort, except for the number of attempts. While calculating the area of the square, the number of attempts remained the same; to sum the numbers, there was an increase by 31.2%.

## 4.5 Threats to Validity

In this section, we describe the internal validity (Section 4.5.1), external validity (Section 4.5.2), and construct validity (Section 4.5.3).

### 4.5.1 Internal validity

We conducted the controlled experiment in two different locations for diversification purposes, which may have influenced the visual attention of the subjects. To mitigate it, we carefully arranged the rooms to have similar light, temperature, and quiet conditions.

The presence of a researcher in the room may have unintentionally influenced the visual attention or performance of the subjects. To mitigate it, we let the subjects feel comfortable and avoided any interaction while they were examining the programs.

The eye tracker equipment has limitations. Even after carefully calibrating and recalibrating it, we still needed an adjustment in the gaze points. The heatmap and plot of fixations revealed a red color for specific subjects and fixations over a blank area not touching the code. For these specific subjects, a small adjustment was sufficient to adjust it. All the fixations for a particular program received the same adjustment. The adjustment in the $y$-coordinate was in a median of 12.5 pixels. We did not adjust the $x$-coordinate. The adjustment in the points may influence their interpretation. We discussed these adjustments for each subject. However, we decided that the threat of adjusting the points would be preferable to the threat of analyzing the data with points not touching the code. The fixations and adjustment strategy are available in our replication package [108].

In pilot studies, a swivel chair impaired data points collected by the camera. To mitigate it, we used chairs without swiveling capability in the controlled experiment. We allocated one hour for each subject and assigned them 10 programs, which may have influenced the

visual effort. To minimize it, we designed simple and short programs.

With the Latin Square design, we blocked the set of programs to control noise. We analyzed the programs combined in the squares and individually. The analysis of individual programs may violate the design. However, analyzing them combined and individually can give a more nuanced and complete understanding of the effects of the refactorings.

### 4.5.2 External validity

We focused on novices in Java, which may restrict generalization to more experienced developers in Java. Other eye tracking studies have also focused on novices to understand code comprehension [15]. In the future, we intend to focus on experienced developers.

We focused on Java, which may restrict generalization to other programming languages. To mitigate it, we used constructions commonly employed in other languages. Most of our subjects reported some experience with other languages. Our programs were designed to contain identifiers in Portuguese, given that the subject were Brazilians.

We assigned the subjects a task to specify the correct output of the program, which was a numeric value. They answered the output aloud after reading the code with no syntax highlighting. This task may not generalize to other tasks, such as finding a bug or adding a feature.

We resorted to small programs aiming at fitting the code onto the screen, which may restrict generalization to larger programs. Thus, our results are conservative since we examined the uses of the Extract Method in simple scenarios. Developers might use the Extract Method refactoring in more complex methods, usually where a comment would be needed to explain part of that method. However, larger and more complex methods would require more lines of code, would take more time in the experiment, and could make the subjects more tired. These factors pose a challenge for controlled eye tracking studies in code comprehension.

Nevertheless, we need studies with larger code snippets. The number of methods in the program may influence the visual effort of the subjects. To minimize threats related to this, we consistently used only one method extracted in the program except for the main method.

The names of methods in the program may influence the comprehension and visual effort of the subjects. Confusing names can hamper comprehension and require more effort. To minimize this threat, we used lessons from previous studies and guidelines, refined the names

through a pilot study, and discussed the names. In addition, in our approach, the original method (version to be refactored) does not give hints about the goal of the code. One might argue that, in a real scenario, both the original method and the extracted method would have meaningful names, which would work as beacons and give hints to the developer about what the extracted method would do. This can be mitigated by the fact that we focus on novices, and beacons might not be so efficient in this context.

### 4.5.3 Construct validity

Code comprehension has often been measured through time and answer correctness [84]. Time, answer correctness, and visual effort have also been combined to investigate code comprehension [102; 33; 30; 77]. In particular, the visual effort has been measured before with fixation duration and fixation count [102; 8]. In addition, eye movement regressions have been associated with visual effort [99].

When we invite the subjects, we have to make them aware that their eyes are being tracked. This may influence where or how much they look at some regions of the code. To minimize this threat, we did not inform them about the precise goals of the study to avoid hypothesis guessing.

## 4.6 Conclusions

We report on a controlled experiment with eye tracking to evaluate the extent of the impact of the Extract Method on code comprehension. We compared the Extract and Inline Method versions of eight tasks measuring their impact on the time, the number of attempts, and the visual effort of 32 novices in Java. We triangulated the metrics of the objective performance with the subjective perceptions of the subjects.

With the Extract Method, there was a significant improvement in the time of two tasks, varying from 70% to 78.8%, and in the number of attempts of three tasks, varying from 20% to 34.4%. Improvements in the visual effort were observed for two tasks, varying from 73.6% to 78.9% in the duration of the fixations, from 67.7% to 75.8% in the fixations count, and from 74.4% to 84.6% in the regressions count. Negative effects were also observed. The time of three tasks increased, varying from 108.4% to 166.9%. The visual effort of two

tasks also increased, varying from 73.1% to 130.1% in the fixation duration, from 137.1% to 194.2% in the fixations count, and from 100% to 200% in the regressions count. While the eye tracking data are more sensitive to fine-grained changes in small snippets, such as the impact of adding one line, we have to be aware that in large code sources, extracting a method can actually reduce the number of lines of code. Thus, we cannot generalize these results to larger code sources. In the interview, the subjects found the tasks with the Extract Method easier to solve. In our data, for smaller snippets, the perception of developers may not always agree with their performance.

Our study contributes to raising educators' awareness about the Extract Method and its potential to ease or hinder code comprehension for novices in Java. Introductory courses should be more selective in choosing programs that do not negatively impact visual effort. For the researchers, our results show the potential of visual metrics to reveal an impact of refactorings that static code metrics cannot capture. For instance, a simple task to calculate the *Area of Square* exhibited 166.9% more time in the AOI and 138.8% more visual regressions. When we moved the calculation to somewhere else in the code, the subjects tended to keep coming back to those lines interrupting the flow. Other approaches, such as functional Magnetic Resonance Imaging (fMRI) during the experiments, could reveal other nuances. This impact revealed through eye tracking may raise the need for tools to assist the novices when applying the Extract Method.

In future work, we aim to evaluate other refactorings from Fowler's catalog [40]. We aim to conduct more controlled experiments with experienced developers, other programming languages, other types of tasks, and larger code.

# Chapter 5

# Study III: Refactoring Configurable Systems

In this chapter, we present an eye tracking study with novices to evaluate refactorings for disciplining `#ifdef` annotations. This chapter is organized as follows: Section 5.1 presents the study definition, and Section 5.2 presents the study methodology. Section 5.3 presents the obtained results, and Section 5.4 discusses a qualitative interview with the novices. Section 5.5 discusses the threats to validity.

## 5.1 Study Definition

In this section, we present the study definition following the Goal-Question-Metrics approach [5]. **We analyze** three refactorings for C programs that discipline `#ifdef` annotations **for the purpose of** understanding whether disciplined annotations associate with improvements **with respect to** code comprehension **from the point of view of** novices in the C programming language in the context of tasks extracted from real projects.

With this goal in mind, we address the following research questions:

- **$RQ_1$: To what extent do disciplined annotations affect task completion time?** To answer this question, we measure the total time duration novices need to solve a "specify the correct output" task with three evaluated refactorings. In addition, we measure the time the subjects spend in specific regions in the task. Our null hypothesis ($H_1$) is

that there is no difference between disciplined and undisciplined annotations regarding time.

- **$RQ_2$: To what extent do disciplined annotations affect the number of attempts?** To answer this question, we measure the number of answers novices submit until solving the task with three evaluated refactorings. Our null hypothesis ($H_2$) is that there is no difference between disciplined and undisciplined annotations regarding the number of answer submissions.

- **$RQ_3$: To what extent do disciplined annotations affect fixation duration?** In the code domain, longer fixations have been associated with a substantial increase in demands of attentiveness [16]. Crosby et al. [28] have shown that there is a correspondence between fixations and attention focus, suggesting the validity of immediacy and eye-mind theory, also in the code domain. The results of those studies imply that longer fixations indicate more attention and consequently more visual effort. To answer this question, we measure the fixation duration of the novices while solving the task with three evaluated refactorings. Our null hypothesis ($H_3$) is that there is no difference between disciplined and undisciplined annotations regarding fixation duration.

- **$RQ_4$: To what extent do disciplined annotations affect fixations count?** Another fixation-based metric, the fixation count ignores the fixation duration and considers only the total number of fixations in a particular area. This metric is also associated with cognitive and visual effort. For instance, a higher number of fixations indicates longer processing time to understand code phrases [8], more attention to complex code [28], and more visual effort to recall the name of identifiers [101]. To answer this question, we measure the fixation count of the novices while solving the task with three evaluated refactorings. Our null hypothesis ($H_4$) is that there is no difference between disciplined and undisciplined annotations regarding fixation count.

- **$RQ_5$: To what extent do disciplined annotations affect regressions count?** Regarding eye tracking metrics, we may have backward eye movements of any length over the stimuli called regressions [15]. According to Sharafi et al. [99], regressions can be used to measure visual effort. The linearity of natural language and code reading has

been measured before using the regression rate in code domain [15]. To answer this question, we measure the total number of regressions. Considering that the code writing follows a writing system represented by a left-to-right and top-to-bottom pattern, to measure regressions, we compute the number of gaze transitions with direction opposed to the writing system, from right to left and bottom to top. Our null hypothesis ($H_5$) is that there is no difference between disciplined and undisciplined annotations regarding regressions count.

## 5.2 Methodology

In this section, we present the methodology of our study. We present the pilot study (Section 5.2.1), experiment phases (Section 5.2.2), subjects (Section 5.2.3), treatments (Section 5.2.4), evaluated refactorings (Section 5.2.5), tasks (Section 5.2.6), eye tracking instrumentation (Section 5.2.7), and finally the analysis (Section 5.2.8).

### 5.2.1 Pilot Study

Before conducting the actual experiment (see Section 5.2.2), we conducted a pilot study with four subjects aiming at evaluating the experiment design, tasks to be used, and setup of the eye tracker. In the pilot study, we could test, adjust, and validate the material used, such as background form, code font size, font style, spaces between the lines of code, and indentation. We also adjusted environment settings, lights, and chair. For instance, fixing the chair allowed us to improve data capturing, eliminate noise, and improve data quality. We do not take the results of the pilot study into account in the analysis.

The pilot study allowed us to refine our experiment design, which consists of five phases: (1) Questionnaire, (2) Tutorial, (3) Warm-up, (4) Tasks, and (5) Interview. We then estimated an average of around 50 minutes for each subject to complete all phases. Next, we describe these phases in detail.

### 5.2.2 Experiment Phases

First, we chose a quiet room to minimize distractions and with typical indoor fluorescent bulbs for the experiment. As the subjects entered the room, we explained what data are captured by the camera. We then asked him/her to fill out a consent form and another form with questions related to programming background experience, experience with C language, and implementing variability with `#ifdef` annotations. We provided the subjects with chairs without wheels, leaning or swivel capability positioned 45–60 cm distant from the screen. In the experiment environment, we stayed close to the subject, but we did not encourage conversation while the subject was performing a task. Second, we presented a tutorial on variability implementation explaining how `#ifdef` annotations work and on basic concepts of conditional compilation. In addition, we explained basic concepts of the C programming language. We did not mention the words "disciplined" or "undisciplined" to the subjects.

Third, we illustrated the nature of the experiment through a simple warm-up task in which we asked the subjects to specify the output given the input. This task was not considered in the analysis of the experiment. We used the eye-tracking camera in the warm-up task so that the subjects got comfortable with the equipment and the study setup. We asked the subject to close their eyes for two seconds before and after solving the task. This allowed us to know exactly when the task started and ended by observing the timestamp. We asked the subjects to verbally provide the output of the code, which is an approach adopted by other studies as well [102; 103; 54]. We provided real-time feedback by emitting a distinct sound corresponding to whether the answer was correct. If the answer was incorrect, subjects could choose to keep trying to submit more answers until getting the correct one, if they felt free to do so. They also had the option of quitting at any time without having to provide any reasons for that.

Fourth, we ran the actual experiment with six tasks. We used the Latin Square approach [12] to ensure that every subject was exposed to each treatment only once and to ensure that the same subject answered the same task only once, avoiding a learning effect. Thus, we randomly assigned subjects to treatments in the cells of each square as depicted in Figure 5.1. The comparison in further analysis occurs across the squares by gathering all subjects who answered the same task.

Fifth, once a subject finished all the tasks, we conducted a semi-structured interview to

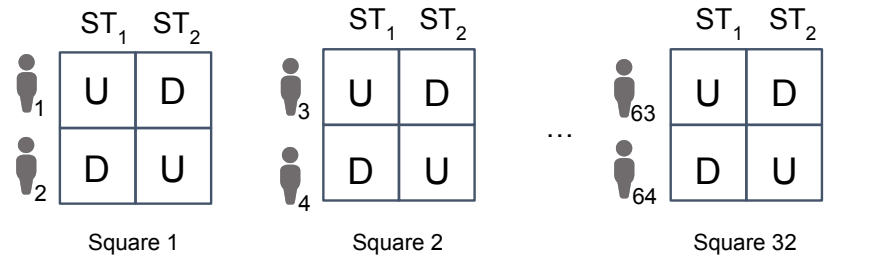



Figure 5.1: Design of experiment with Latin Squares with 64 subjects with projects $P_1$, $P_2$, $P_3$, and $P_4$. U and D refer to undisciplined and disciplined annotation tasks, respectively .

obtain qualitative feedback on how they approached the tasks. We asked the subjects three questions:

- How did you find the output? What strategy did you use?
- How difficult was it to find the output: very easy, easy, neuter, difficult, or very difficult?
- What were the difficulties, if any?

When answering the third question, we asked the subjects to point out the code locations where they had difficulties. This strategy helped us to collect qualitative feedback, and we could observe whether their difficulties matched the fixation duration, the fixation count, and the regressions count.

In some cases, we had to calibrate the camera twice or thrice until we gained confidence that the data captured by the camera could be reliable/useful or that we could get the data corrected. Camera calibration consists of an automatic procedure in which the subject is asked to look at specific locations on the screen and, during that, the camera's integrated system customizes captured data according to each subject's eye characteristics. The camera indicates when calibration is successfully done.

In addition, we were careful with environmental aspects and the swivel function of the chair, so that subjects' eyes could remain calibrated, and the data could not suffer from external noise. Despite these measures, it was still difficult to obtain perfect data given

camera limitations and some subjects' aspects. Thus, we had to perform data correction by slightly shifting chunks of fixations up or down. We discuss this strategy and its effects in the threat to validity section (see Section 5.5.1). We provide a replication package with the data collected, tasks, and other materials [2].

### 5.2.3 Subjects

We performed the study with 64 subjects. In total, we had 42 undergraduates, 11 MSc. students, 8 PhD. students, and 3 postdocs. Regarding the experience with programming languages, 40 subjects reported having experience with C for less than six months. In addition, 14 reported one year or less, 9 from one year to three years, and 1 with more than three years. Regarding their experience, we consider "novices" all the subjects who know how to program but have little experience specifically with C programming language, which corresponds to all subjects in the study except for 10. All subjects reported having experience with another programming language, such as Java. On a scale from very inexperienced (1) to very experienced (5), the median answer was experienced (4). We asked about Java because it is a common practice to teach Java in computer science courses where the study was conducted, however, it could be any other procedural language. Subjects were invited mainly through e-mails and text messages that suggested them to respond by communicating their availability. In addition, we also met some subjects in person and invited them.

### 5.2.4 Treatments

We expose each subject to three disciplined (D) and three undisciplined (U) annotated tasks as seen in Figure 5.2, which results in six tasks ($T_1$-$T_6$) for each subject. The same subject does not solve the same disciplined and undisciplined annotated task to avoid a learning effect. For that, we have two projects ($P_1$ and $P_2$) that comprehend similar but distinct tasks with the same refactorings instantiated or, in other words, a distinct version of the same task with similar structure but involving distinct variables, arithmetic operations, and outputs. We consider this study as within-subjects in the sense that the same subject is exposed to both treatments but does not solve the same task in both disciplined and undisciplined annotations [21]. In Figure 5.3, we present an example of the distribution of the subjects, tasks, and

refactorings according to the projects.

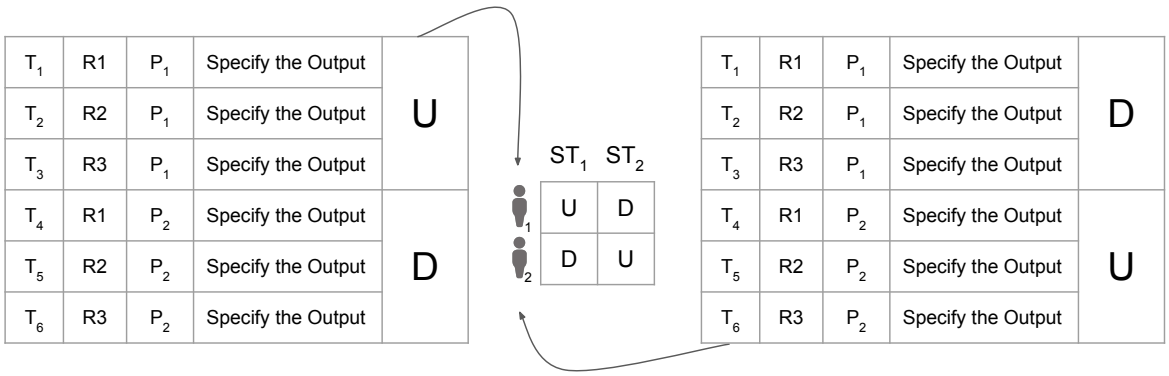



Figure 5.2: Structure of the experiment in terms of experimental units of the study. There are six tasks ($T_1$–$T_6$) distributed in two sets of tasks ($ST_1$ and $ST_2$), with R1 ⟨wrapping function call⟩, R2 ⟨undisciplined `if` conditions⟩, and R3 ⟨alternative `if` statements⟩, and two projects ($P_1$ and $P_2$).

In total, 32 subjects solved three tasks with disciplined annotations from three distinct refactorings of $P_1$, and three tasks with undisciplined annotations from $P_2$. Thus, each subject solves two distinct tasks, one without applying the refactoring (undisciplined version), and another with the refactoring applied (disciplined version). In addition, 32 subjects solved six tasks with three refactorings, three with undisciplined and three with disciplined annotations, but from $P_3$ and $P_4$. In all of them, the subjects had the task of specifying the correct output. We present an open-ended question so that the subject could read the entire code and find the output for themselves. The undisciplined versions are our baseline group, and the disciplined ones are the treatment group.

### 5.2.5 Evaluated Refactorings

In Figure 5.4, we present three refactorings to discipline annotations proposed by Medeiros et al. [75] and evaluated in our study. Each refactoring is a unidirectional transformation that consists of two templates of C code snippets: the left-hand side and the right-hand side. The left-hand side defines a template of C code that contains undisciplined preprocessor usage. The right-hand side is a corresponding template for the refactored code removing undisciplined preprocessor usage. We can apply a refactoring whenever the left-hand side template is matched by a piece of C code and when it satisfies the preconditions ($\rightarrow$). A

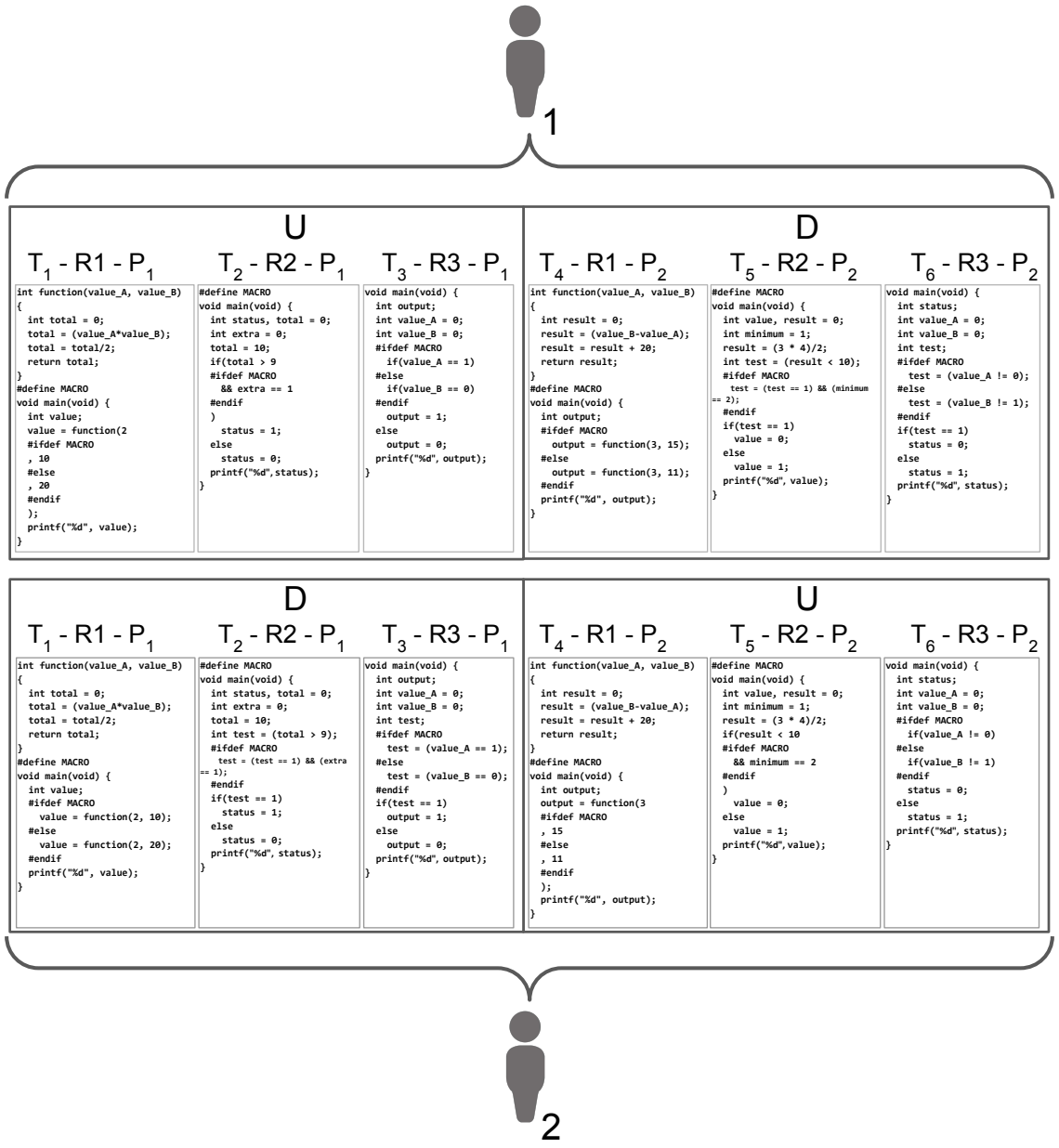



Figure 5.3: Distribution of subjects, tasks, and refactorings in two projects of the study. The structure of the tasks ($T_1$–$T_6$) in projects $P_1$ and $P_2$, before and after applying the refactoring, is similar but involves distinct elements. R1, R2, and R3 refer to Refactoring 1 (Undisciplined returns), Refactoring 2 (Undisciplined `if` conditions), and Refactoring 3 (Alternative `if` statements).

matching is an assignment of all meta-variables in the left-hand side and right-hand side templates to concrete values from the source code. We highlight meta-variables using capital letters, and we use the symbol $\oplus$ to represent arbitrary binary operators. Any element not mentioned in both C code snippets remains unchanged, so the refactoring templates only show the differences among pieces of code.

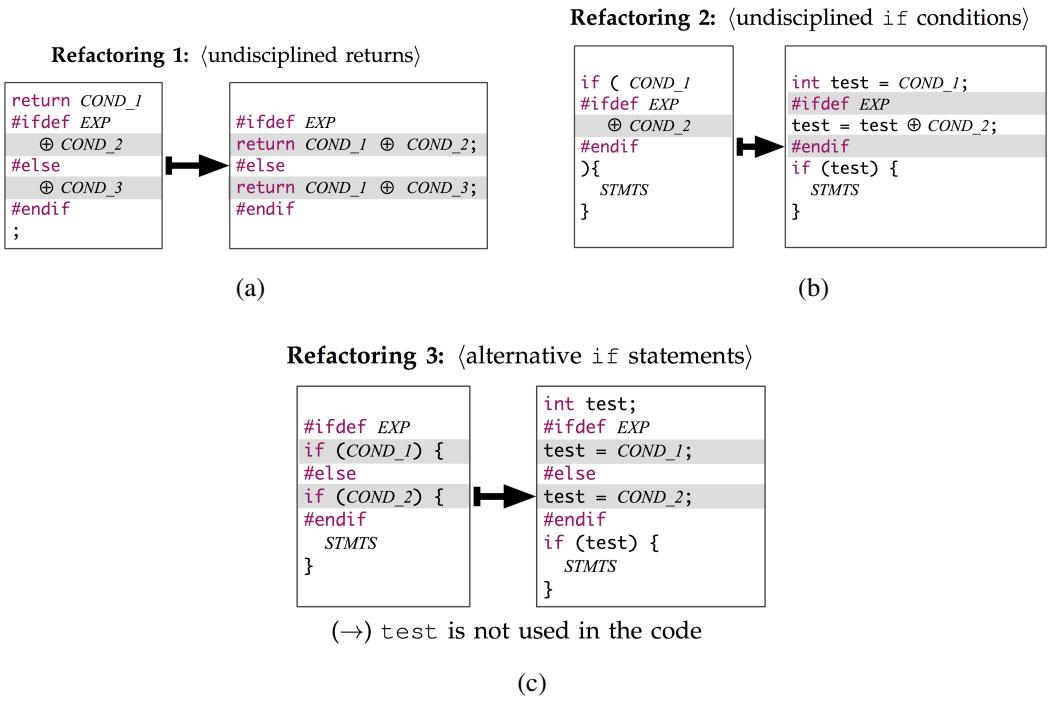



Figure 5.4: Refactorings R1, R2, and R3 to discipline `#ifdef` annotations evaluated in this study.

Medeiros et al. [75] surveyed 246 developers to access their preference regarding disciplined or undisciplined annotated code. We selected the top three refactorings that developers most preferred to discipline annotations. Moreover, Medeiros et al. [75] showed that there are more than 2,200 opportunities to apply the three refactorings in 57 out of 63 code repositories, and 27 out of 63 projects contain possibilities of applying all three refactorings, reaching up to 2,101 opportunities. Furthermore, the three selected refactorings show a relevant acceptance in practice. For instance, Medeiros et al. [75] submitted six patches using R1 and all patches were accepted; five patches were submitted using R2, and 80% of them were accepted; five patches were submitted using R3, and 80% of them were accepted. In addition, Malaquias et al. [70] submitted 31 patches using R2, and 61.2%

were accepted; 63 patches using R3 and 63.4% were accepted. We preferred to focus on evaluating a limited, well-studied set of three refactoring types to gain more confidence about the results instead of evaluating more refactoring types.

### 5.2.6 Tasks

The code snippets were selected as a result of mining code repositories for commits that showed an opportunity to apply the evaluated refactorings [75]. Thus, all tasks have a template associated with real projects, which means an assignment of the meta-variables in the tasks to concrete values in the source code in the projects. For instance, R1 was applied to a task with a template associated with *Vim's* source code, R2 was applied to a task with a template associated with *Libpng's* source code, and R3 was applied to a task with template associated with *OpenSSL*. Malaquias et al. [70] have also used similar tasks in their study. We decided to use simple constructions commonly occurring in many programming languages. The difference in lines of code between both disciplined and undisciplined versions is two lines for R1 and R3. R2 remains with the same number of lines of code. Even though R1 in Figure 5.4(a) involves undisciplined returns, according to Medeiros et al. [75], the return statement is only an example. They handle other statements with subexpressions in the same way, such as a function call as we have used.

There are several types of maintenance tasks, such as applying refactorings, fixing bugs, and adding functionality. Our study focused on a type of task that focuses on code comprehension. We assume that to add functionality, refactor code, and fix bugs, developers will need to at least understand the code. For this reason and for time constraints, we only focused on this type of task. The tasks also involved answering open-ended questions, which subjects could answer by saying out loud the resulting output without any multiple choices.

Moreover, a systematic literature review on code comprehension conducted by Oliveira et al. [84] revealed that the majority of the studies (70%) involve asking subjects to provide information about a program, such as to specify the output. In addition, 83% of the retrieved studies use correctness as a response variable, and 50% use time and correctness together. We then followed a commonly adopted approach, being aligned with the literature. Following Bloom's taxonomy described in their work [84], "understanding" consists of one level of the

**Refactoring 1:** ⟨ wrapping function call ⟩

Undisciplined

```
int function(value_A, value_B) {
        int total = 0;
        total = (value_A*value_B);
        total = total/2;
        return total;
}
#define MACRO
void main(void) {
        int value;
AOI     value = function(2
        #ifdef MACRO
AOI Activated       , 10
        #else
AOI Deactivated     , 20
        #endif
        );
        printf("%d", value);
}
```

Disciplined

```
int function(value_A, value_B) {
        int total = 0;
        total = (value_A*value_B);
        total = total/2;
        return total;
}
#define MACRO
void main(void) {
        int value;
AOI     #ifdef MACRO
AOI Activated       value = function(2, 10);
        #else
AOI Deactivated     value = function(2, 20);
        #endif
        printf("%d", value);
}
```

(a) R1 - $P_1$ - Undisciplined to disciplined annotations

**Refactoring 2:** ⟨ undisciplined `if` conditions ⟩

Undisciplined

```
#define MACRO
void main(void) {
        int value, result = 0;
        int minimum = 1;
        result = (3 * 4)/2;
        if(result < 10
        #ifdef MACRO
                && minimum == 2
AOI     #endif
        )
AOI Deactivated     value = 0;
        else
AOI Activated       value = 1;
        printf("%d",value);
}
```

Disciplined

```
#define MACRO
void main(void) {
        int value, result = 0;
        int minimum = 1;
        result = (3 * 4)/2;
        int test = (result < 10);
        #ifdef MACRO
AOI             test = (test == 1) && (minimum == 2);
        #endif
        if(test == 1)
AOI Deactivated     value = 0;
        else
AOI Activated       value = 1;
        printf("%d", value);
}
```

(b) R2 - $P_2$ - Undisciplined to disciplined annotations

**Refactoring 3:** ⟨ alternative `if` statements ⟩

Undisciplined

```
void main(void) {
        int status;
        int value_A = 0;
        int value_B = 20;
AOI     #ifdef MACRO
AOI Deactivated     if(value_A > 0)
        #else
AOI Activated       if(value_B > 10)
        #endif
AOI Activated       status = 0;
        else
AOI Deactivated     status = 1;
        printf("%d",  status);
}
```

Disciplined

```
void main(void) {
        int status;
        int value_A = 0;
        int value_B = 20;
AOI     int test;
        #ifdef MACRO
AOI Deactivated     test = (value_A > 0);
        #else
AOI Activated       test = (value_B > 10);
        #endif
        if(test == 1)
AOI Activated       status = 0;
        else
AOI Deactivated     status = 1;
        printf("%d",  status);
}
```

(c) R3 - $P_4$ - Undisciplined to disciplined annotations

Figure 5.5: Examples of six tasks from projects $P_1$, $P_2$, and $P_4$, before and after applying R1, R2, and R3.

dimension of interpretation. Most activities such as code trace and inspections performed by subjects occur at the "understanding" level followed by the "analysis" level. Therefore, we align with them in the sense that, to evaluate code comprehension, we elaborated code tasks to be inspected and traced for providing the correct output.

In general, we used tasks with less than 20 lines to fit the screen size. The eye tracker was mounted on a laptop screen with a resolution of 1366 x 720 pixels, a width of 30.9 cm, and a height of 17.4 cm, at a distance of 50-60 cm from the subject. The code tasks were displayed as an image in the full-screen mode, but no Integrated Development Environment (IDE) was used, nor number for the lines. All the products of the configurable system could be compiled with no syntactic errors. We had tasks with macro enabled and tasks with macro disabled, and they presented distinct outputs depending on whether macro was defined or not. For instance, when the macro is defined, the macro region gets activated and the statement inside the macro is exercised. When the macro is not defined, the macro region is not activated. We made sure that each task of the same refactoring and same project, whether disciplined or undisciplined version, presented the same output. Program style followed Consolas font style, font size 18, no spaces between lines, and eight white spaces of indentation with four white spaces from y-axis.

In Figure 5.5, we present three undisciplined annotated tasks and their refactored versions. For instance, in Figure 5.5(a), AOI defines the area in which both code versions differ. It encompasses two sub-areas, namely, AOI Activated and AOI Deactivated. The main distinction between these two sub-areas relies on the fact that, when macro is enabled, only one sub-area of the AOI gets exercised, which is the AOI Activated, because it contains a statement that is activated only when macro is enabled. When macro is disabled, only one sub-area gets activated, which is the AOI Activated. This approach allows us to measure time and fixations inside those areas. For instance, we can observe how much time subjects spend looking at the activated area when macro is enabled, how many times they fixate on it and for how long. Accordingly, we can do that for the deactivated area when subjects are looking at the opposite statement when macro is enabled.

### 5.2.7 Fixation Instrumentation

Fixations can be defined as the stabilization of the eye on part of a stimulus for a period of time [93; 55]. The duration threshold typically depends on the tasks processing demands. According to Salvucci and Goldberg [93], the duration threshold can be between 100 and 200 ms, while according to Rayner [91], our eyes remain relatively still during fixations for about 200–300 ms. Commonly applied in practice, we applied the Dispersion-Threshold Identification (I-DT) algorithm to classify gaze samples into fixations [93]. It classifies gaze samples as belonging to a fixation if the samples are located within a spatial region of approximately 0.5 degrees [81]. The I-DT algorithm requires two parameters: the dispersion threshold and the duration threshold [81]. We used a dispersion threshold of approximately 0.5 degrees, which corresponded to 25 pixels in our screen. For the duration threshold, we used 200 ms based on the study of Salvucci and Goldberg [93]. The classification of data points into relevant eye movements reduces the amount of eye tracking data to process and allows the researcher to focus on the measures relevant to the research question.

### 5.2.8 Analysis

Of all 64 subjects, resulting in 384 tasks, one subject opted for not completing two out of six tasks and another opted for not completing one out of six tasks, resulting in three tasks not being completed, which corresponds to less than 1% of the total of tasks. We included those two subjects and we used Multivariate Imputation by Chained Equations (MICE) implemented as a mice package in R for a multiple imputation method namely Predictive Mean Matching (PMM) for the three tasks. The PMM method imputes univariate missing data using predictive mean matching [58]. This approach performs better when the sample size is sufficiently large [64], which was our case.

After data collection, we performed a statistical analysis to test our null hypotheses. In our analysis, when the $p$-value was inferior to 0.05, we rejected the null hypothesis that there was no difference between the median of the treatments and conclude that a significant difference did exist. We tested data distribution for normality using Shapiro-Wilk's test [97]. Whenever the data were normally distributed or we could normalize it, we performed the parametric $t$ test for two independent samples. The $t$ test consists of an analysis method

to test two groups to see if there is a statistically significant difference between them [104; 100]. Before performing the *t* test, we tested whether the data satisfied another condition besides normality of distribution of the data, which is whether the variances of the two groups were equal [104]. For the data that could not be normalized, we used the non-parametric test Mann-Whitney, also known as Wilcoxon test, which compares two independent groups of samples that do not follow a normal distribution [104; 100]. In addition, since the mean value might not be appropriate to characterize values of fixation duration or count because the description of the central tendency might be dependent on some very high values [43], we computed and based our analysis on the median. Both the analysis of the individual and combined refactorings were analyzed using the median as a measure of central tendency.

We also used Cliff's Delta [25] to yield the effect size. Since in most cases our data does not follow a normal distribution, Cohen's effect size would not be appropriate. Cohen has made widely accepted suggestions on what constitutes small and large effects [26]. For instance, according to Cohen's description, the effect size of 0.2 suggests a small effect, 0.5 a medium effect, and 0.8 a large effect. The negative sign of the effects implies that the values of the treatment group (disciplined annotations) are greater than the control group (undisciplined annotations).

## 5.3 Results

In Sections 5.3.1–5.3.5, we present the results for our research questions. In each of these sections, when we mention statistically significant differences, we mean that we can reject the null hypothesis for the research question being analyzed. In Section 5.3.6, we summarize the results for all research questions.

### 5.3.1 $RQ_1$: To what extent do disciplined annotations affect task completion time?

After applying R1 ⟨wrapping function call⟩ or R3 ⟨alternative `if` statements⟩, novices exhibited faster task completion (see Table 5.1). We observed statistically significant reductions by 23.8% and 46.9% in the time they spent in AOI, respectively. Thus, they spend less

time in AOI in Figure 5.5(a) and Figure 5.5(c) after applying R1 or R3. After applying R3, we observed a statistically significant reduction by 42.4% in the time they spent in the whole code. The application of R3 was associated with a reduction in the time novices spent in both activated and deactivated areas by 51.1% and 59.4%, respectively. They spend less time in AOI Activated and AOI Deactivated areas in Figure 5.5(b), both right and left-hand sides. On the other hand, we observed a statistically significant increase by 47.6% in time novices spent in AOI after applying R2 ⟨undisciplined `if` conditions⟩, which means that they spent more time in AOI in Figure 5.5(b), right-hand side. It is also associated with a slowdown in their task completion by increasing the time they spent on the whole code by 24.6% with R2 applied. Thus, applying R1 or R3 is associated with improvements in task completion time for novices. However, after applying R2, we cannot observe the same effect.

Medeiros et al. [75] found that 27 out of 63 projects evaluated contain possibilities of applying all three refactorings together. Combining all refactorings follows the Latin Square methodology. We assign two subjects to each square and each subject solves three undisciplined and three disciplined versions. We can combine the refactorings in two ways. We can perform combinations of the individual refactorings, but not pairing the subjects in the squares, and we can combine the refactorings by pairing the subjects.

After applying R1, R2, and R3, in combination, the novices exhibited faster task completion (see Table 5.1). We observed a statistically significant reduction by 20% in the time they spent in AOI after applying R1, R2, and R3 combined. Combined, the application of refactorings is also associated with a reduction in the time they spent in the whole code by 12.5%. Thus, applying R1, R2, and R3 combined is associated with improvements in task completion time for novices.

We also analyzed the time outside AOI and we found a statistically significant difference only after applying R3. Since R3 showed differences in time both inside and outside AOI, we analyzed the whole code. After applying R3, we observed a statistically significant reduction in time spent in the whole code. Therefore, we focus on presenting first the analysis of the AOI followed by the analysis of the whole code.

Table 5.1: Summarizing the results for time completion ($RQ_1$). Bold font represents statistically significant differences. U = undisciplined annotations; D = disciplined annotations; PD = percentage difference; PV = *p*-value; ES = effect size. Columns U and D are based on the median as a measure of central tendency.

| Task | In Code | | | | | In AOI | | | | | In Activated Areas | | | | | In Deactivated Areas | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | U (sec) | D (sec) | PD % | PV | ES | U (sec) | D (sec) | PD % | PV | ES | U (sec) | D (sec) | PD % | PV | ES | U (sec) | D (sec) | PD % | PV | ES |
| R1 | 34.1 | 32.5 | ↓5.2 | 0.14 | n/a | 12.7 | 9.7 | ↓23.8 | **0.004** | -0.29 | 1.9 | 2.7 | ↑42.0 | 0.08 | 0.29 | 1.7 | 2.0 | ↑11.8 | 0.24 | n/a |
| R2 | 41.3 | 51.4 | ↑24.6 | **0.01** | 0.23 | 25.2 | 36.9 | ↑47.6 | **0.01** | 0.24 | 3.4 | 3.1 | ↓8.4 | 0.65 | n/a | 1.7 | 1.4 | ↓16.5 | 0.41 | n/a |
| R3 | 39.4 | 22.5 | ↓42.4 | $\mathbf{10^{-5}}$ | -0.44 | 29.1 | 15.4 | ↓46.9 | $\mathbf{10^{-5}}$ | -0.44 | 6.3 | 3.1 | ↓51.1 | $\mathbf{7x10^{-7}}$ | -0.49 | 3.1 | 1.2 | ↓59.4 | $\mathbf{2x10^{-9}}$ | -0.60 |
| All | 38.1 | 33.7 | ↓12.5 | **0.01** | -0.14 | 20.6 | 16.9 | ↓20.0 | **0.02** | -0.13 | 3.4 | 3.0 | ↓13.7 | 0.18 | n/a | 2.2 | 1.6 | ↓23.4 | **0.001** | -0.18 |

**Finding 1:** In our study, after applying R1 or R3 in isolation, the novices exhibit faster task completion. Faster task completion is also exhibited by the novices after applying R1, R2, and R3 in combination.

### 5.3.2 $RQ_2$: To what extent do disciplined annotations affect the number of attempts?

After applying R3, novices provide more correct answers. Although the median number of submissions remained the same, we realize that, by observing the box-plot in Figure 5.6(c), the data is less spread when R3 was applied, which can explain the observed differences. While they both present the same median number of submissions, after applying R3 (see Table 5.2), the mean number of submissions decreased from 1.25 to 1.20. It means that, on average, they need 1.2 attempts to solve the task. Thus, applying R3 associated with improvements in the attempts of the answers submitted by the novices. Combined, we did not find differences in attempts after applying R1, R2, and R3.

**Finding 2:** In our study, after applying R3 in isolation, the novices provide more correct answers. No differences were observed after applying R1, R2, and R3 in combination.

Table 5.2: Summarizing the results for attempts ($RQ_2$). Bold font represents statistically significant differences. U = undisciplined annotations; D = disciplined annotations; PD = percentage difference; PV = *p*-value; ES = effect size. Columns U and D are based on the median as a measure of central tendency.

| Task | Submissions | | | | |
|---|---|---|---|---|---|
| | U | D | PD % | PV | ES |
| R1 | 1.0 | 1.0 | n/a | 0.37 | n/a |
| R2 | 1.0 | 1.0 | n/a | 0.18 | n/a |
| R3 | 1.0 | 1.0 | n/a | **0.03** | -0.15 |
| All | 1.0 | 1.0 | n/a | 0.43 | n/a |

### 5.3.3 $RQ_3$: To what extent do disciplined annotations affect fixation duration?

After applying R1 or R3, novices exhibit a reduction in the fixation duration in AOI (see Table 5.3). We observed statistically significant reductions by 25% and 44.7% in the duration of the fixations in AOI after applying R1 and R3, respectively. This correlation implies that novices make shorter fixations in AOI in Figure 5.5(a) and Figure 5.5(c) after applying R1 or R3. In the whole code, novices also exhibit a reduction in the fixation duration after applying R3. We observed a statistically significant reduction by 37.2% in the duration of the fixations. Thus, applying R1 or R3 associated with a reduction in the fixation duration in the AOI for novices.

After applying R1, R2, and R3, novices also exhibit a reduction in the fixation duration in AOI. We observed a statistically significant reduction by 28.5% in the duration of the fixations in AOI after applying R1, R2, and R3. Combined, the application of refactorings also associated with a reduction in the duration of the fixations in the whole code by 20.8%. Thus, applying them combined associated with a reduction in the fixation duration both in the AOI and in the whole code for novices.

Table 5.3: Summarizing the results for duration of fixations ($RQ_3$). Bold font represents statistically significant differences. U = undisciplined annotations; D = disciplined annotations; PD = percentage difference; PV = *p*-value; ES = effect size. Columns U and D are based on the median as a measure of central tendency.

| Task | In Code | | | | | In AOI | | | | | In Activated Areas | | | | | In Deactivated Areas | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | U (sec) | D (sec) | PD % | PV | ES | U (sec) | D (sec) | PD % | PV | ES | U (sec) | D (sec) | PD % | PV | ES | U (sec) | D (sec) | PD % | PV | ES |
| R1 | 15.8 | 14.8 | ↓11.2 | 0.15 | n/a | 6.3 | 4.7 | ↓25.0 | **4x10$^{-3}$** | -0.27 | 1.0 | 1.4 | ↑39.8 | 0.17 | n/a | 0.8 | 0.9 | ↑17.4 | 0.70 | n/a |
| R2 | 23.6 | 28.6 | ↑22.6 | 0.20 | n/a | 16.0 | 20.6 | ↑28.2 | 0.20 | n/a | 1.9 | 1.7 | ↓7.2 | 0.75 | n/a | 0.9 | 1.0 | ↑8.4 | 0.29 | n/a |
| R3 | 19.6 | 12.3 | ↓37.2 | **10$^{-4}$** | -0.41 | 15.4 | 8.5 | ↓44.7 | **6x10$^{-5}$** | -0.42 | 3.7 | 1.7 | ↓53.2 | **6x10$^{-6}$** | -0.46 | 1.9 | 0.6 | ↓65.6 | **10$^{-8}$** | -0.57 |
| All | 20.2 | 16.1 | ↓20.8 | **0.01** | -0.15 | 12.2 | 8.6 | ↓28.5 | **10$^{-3}$** | -0.16 | 1.8 | 1.5 | ↓16.4 | **0.02** | -0.13 | 1.1 | 0.7 | ↓31.5 | **10$^{-3}$** | -0.18 |

**Finding 3:** In our study, after applying R1 or R3 in isolation, the novices exhibit a reduction in the fixation duration in the AOI. A reduction in the fixation duration in the AOI is also exhibited by the novices after applying R1, R2, and R3 in combination.

### 5.3.4 $RQ_4$: To what extent do disciplined annotations affect fixation count?

After applying R1 or R3, novices exhibit a reduction in the fixation count in AOI (see Table 5.4). We observed statistically significant reductions by 17.5% and 48.4% in the number of the fixations in AOI after applying R1 and R3, respectively. This correlation implies that novices make fewer fixations in AOI in Figure 5.5(a) and Figure 5.5(c) after applying R1 or R3. In the whole code, novices also exhibit a reduction in the fixation count after applying R3. We observed a statistically significant reduction by 39.1% in the number of the fixations. Thus, applying R1 or R3 associated with a reduction in the fixation count in the AOI for novices.

After applying R1, R2, and R3, novices also exhibit a reduction in the fixation count in AOI. We observed a statistically significant reduction by 26.7% in the number of fixations in AOI after applying R1, R2, and R3. Combined, the application of refactorings also associated with a reduction in the number of fixations in the whole code by 22.4%. Thus, applying them combined associated with a reduction in the fixation count both in the AOI

Table 5.4: Summarizing the results for fixation count ($RQ_4$). Bold font represents statistically significant differences. U = undisciplined annotations; D = disciplined annotations; PD = percentage difference; PV = *p*-value; ES = effect size. Columns U and D are based on the median as a measure of central tendency.

| Task | In Code | | | | | In AOI | | | | | In Activated Areas | | | | | In Deactivated Areas | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | U | D | PD % | PV | ES | U | D | PD % | PV | ES | U | D | PD % | PV | ES | U | D | PD % | PV | ES |
| R1 | 49.0 | 45.0 | ↓11.2 | 0.12 | n/a | 20.0 | 16.5 | ↓25.0 | **0.004** | -0.28 | 3.0 | 4.0 | ↑39.8 | **0.03** | -0.14 | 3.0 | 3.5 | ↑17.4 | 0.48 | n/a |
| R2 | 68.5 | 85.0 | ↑24.0 | 0.15 | n/a | 46.0 | 59.0 | ↑28.2 | 0.12 | n/a | 5.5 | 4.5 | ↓7.2 | 0.14 | n/a | 3.0 | 2.5 | ↑8.42 | 0.83 | n/a |
| R3 | 60.0 | 36.5 | ↓37.2 | **5x10$^{-5}$** | -0.42 | 47.5 | 24.5 | ↓44.7 | **10$^{-5}$** | -0.43 | 11.0 | 5.0 | ↓53.2 | **6x10$^{-7}$** | 0.24 | 6.0 | 2.0 | ↓65.6 | **5x10$^{-9}$** | -0.09 |
| All | 61.5 | 48.0 | ↓20.8 | **9x10$^{-3}$** | -0.15 | 34.5 | 25.5 | ↓28.5 | **4x10$^{-3}$** | -0.16 | 6.0 | 5.0 | ↓16.4 | **0.03** | 0.05 | 4.0 | 2.5 | ↓31.5 | **10$^{-3}$** | -0.05 |

and in the whole code for novices.

**Finding 4:** In our study, after applying R1 or R3 in isolation, the novices exhibit a reduction in the fixation count in the AOI. A reduction in the fixation count in the AOI is also exhibited by the novices after applying R1, R2, and R3 in combination.

### 5.3.5 $RQ_5$: To what extent do disciplined annotations affect regressions count?

Since we are interested in transitions, we focused this analysis on the AOI, which comprises a few lines of code together, and on the whole code, comprising all lines of code together, leaving out activated and deactivated areas. Notice that our tasks follow the left-to-right and top-to-bottom writing system and have no loops. Thus, a regression is a saccade with an opposed direction in this writing system. After applying R1 or R3, novices exhibit a reduction in the regressions count in AOI (see Table 5.5). We observed statistically significant reductions by 33.3% and 60.5% in the number of regressions in AOI after applying R1 and R3, respectively. It associates with improvements in the number of regressions in the AOI in Figure 5.5(a) after applying R1, and in Figure 5.5(c) after applying R3. In other words, the novices read the code 33.3% and 60.5% more often against the writing system before R1 and R3 were applied, respectively. In the whole code, novices also exhibit a reduction

Table 5.5: Summarizing the results for regressions count ($RQ_5$). Bold font represents statistically significant differences. U = undisciplined annotations; D = disciplined annotations; PD = percentage difference; PV = *p*-value; ES = effect size. Columns U and D are based on the median as a measure of central tendency.

| Task | In Code | | | | | In AOI | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | U | D | PD % | PV | ES | U | D | PD % | PV | ES |
| R1 | 21.0 | 18.0 | ↓14.2 | 0.09 | n/a | 6.0 | 4.0 | ↓33.3 | **6x10$^{-4}$** | -0.34 |
| R2 | 30.0 | 36.0 | ↑20.0 | 0.25 | n/a | 18.0 | 20.0 | ↑11.1 | 0.20 | n/a |
| R3 | 26.0 | 13.0 | ↓50.0 | **9x10$^{-6}$** | -0.46 | 19.0 | 7.5 | ↓60.5 | **10$^{-6}$** | -0.49 |
| All | 25.0 | 19.0 | ↓24.0 | **10$^{-3}$** | -0.18 | 12.5 | 8.0 | ↓36.0 | **4x10$^{-4}$** | -0.20 |

in the regressions count after applying R3. We observed a statistically significant reduction by 50% in the number of regressions. Thus, applying R3 associated with alleviating the need of going back to the same or to previous lines of the code in AOI for novices from the regressions count perspective.

After applying R1, R2, and R3, novices exhibit a reduction in the regressions count in the AOI. We observed a statistically significant reduction by 36% in the number the regressions in the AOI after applying R1, R2, and R3. Combined, the application of the refactorings also associated with a reduction in the number of regressions in the whole code by 24%. Thus, applying them combined associated with alleviating the need of going back to the same or to previous lines of code in the whole code.

**Finding 5:** In our study, after applying R1 or R3 in isolation, the novices exhibit a reduction in the regressions count in the AOI. A reduction in the regressions count in the AOI is also exhibited by the novices after applying R1, R2, and R3 in combination.

Table 5.6: Summary of the null-hypotheses' statuses in isolated refactorings in the AOIs.

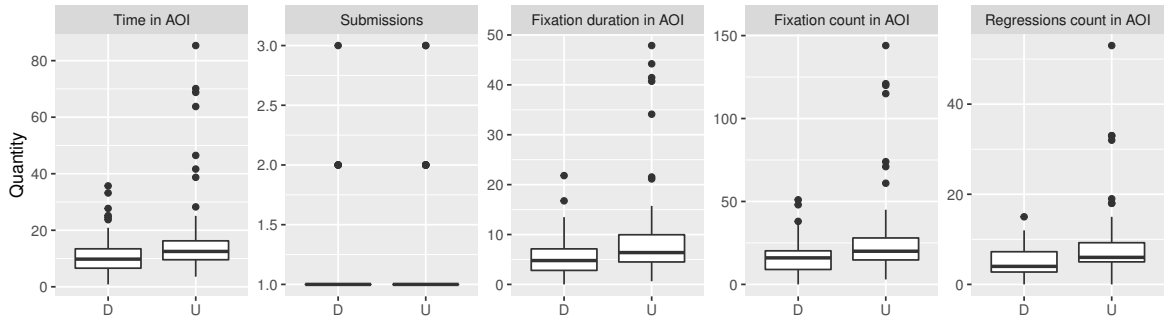
| RQs | Refact. | Null-Hypothesis | *p*-value | Status | Effect size |
|---|---|---|---|---|---|
| $RQ_1$ | R1 | No difference in time between treatments | < 0.05 | Rejected | Small |
| $RQ_2$ | R1 | No difference in attempts between treatments | > 0.05 | Not Rejected | — |
| $RQ_3$ | R1 | No difference in fixation duration between treatments | < 0.05 | Rejected | Small |
| $RQ_4$ | R1 | No difference in fixation count between treatments | < 0.05 | Rejected | Small |
| $RQ_5$ | R1 | No difference in regressions count between treatments | < 0.05 | Rejected | Small |
| $RQ_1$ | R2 | No difference in time between treatments | < 0.05 | Rejected | Small |
| $RQ_2$ | R2 | No difference in attempts between treatments | > 0.05 | Not Rejected | — |
| $RQ_3$ | R2 | No difference in fixation duration between treatments | > 0.05 | Not Rejected | — |
| $RQ_4$ | R2 | No difference in fixation count between treatments | > 0.05 | Not Rejected | — |
| $RQ_5$ | R2 | No difference in regressions count between treatments | > 0.05 | Not Rejected | — |
| $RQ_1$ | R3 | No difference in time between treatments | < 0.05 | Rejected | Medium |
| $RQ_2$ | R3 | No difference in attempts between treatments | < 0.05 | Rejected | Small |
| $RQ_3$ | R3 | No difference in fixation duration between treatments | < 0.05 | Rejected | Medium |
| $RQ_4$ | R3 | No difference in fixation count between treatments | < 0.05 | Rejected | Medium |
| $RQ_5$ | R3 | No difference in regressions count between treatments | < 0.05 | Rejected | Medium |

### 5.3.6 Summary

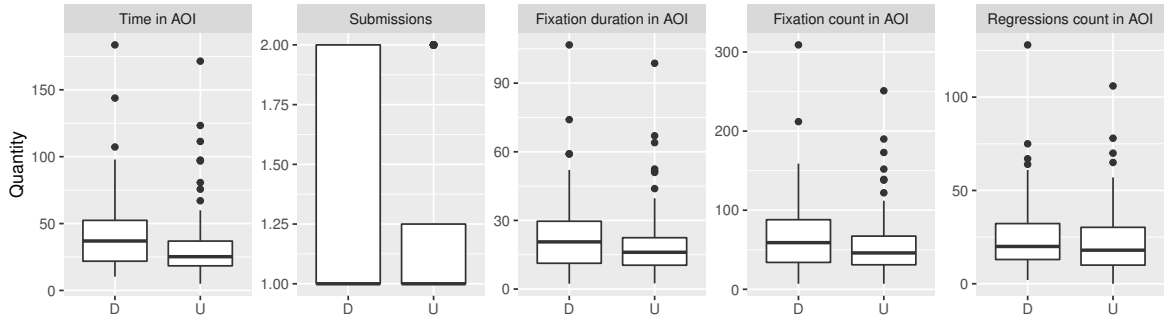
In Table 5.6, we present the confirmation/rejection of the original null-hypotheses. Figures 5.6(a)–(c) summarize total time in the AOI, number of answer submissions, fixation duration in the AOI, fixation count in the AOI, and regressions count in the AOI for R1, R2, and R3, respectively, for $P_1$–$P_4$ combined. Figure 5.6(d) summarizes the results for the mentioned metrics, however, from a combined perspective, instead of analyzing each refactoring individually.

The greatest effects were observed after applying R3. For instance, the effect size for R3 in $RQ_1$, $RQ_3$, $RQ_4$, $RQ_5$ is close to medium in AOI (Cliff's delta ranges from -0.42 to -0.49). In other words, the effects of applying R3 are noticeable. The effects after applying R1 were also noticeable but to a smaller degree.

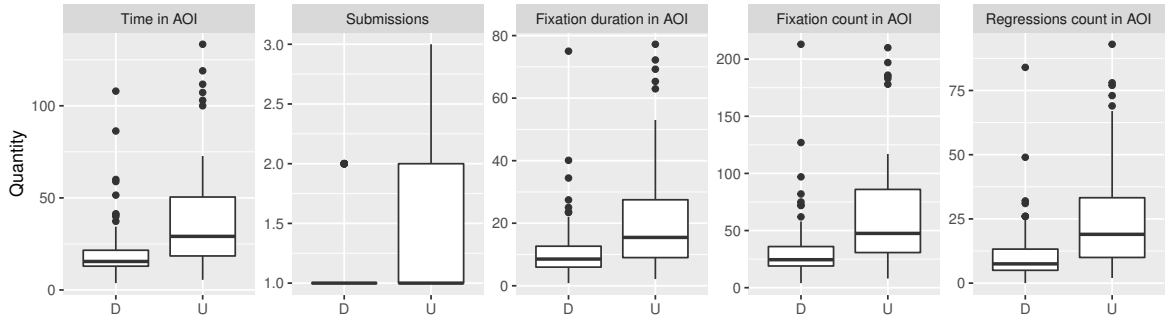
After applying the refactorings R1, R2, and R3, we have an addition of 40.6%, 57.3%, and 37.7% in the median number of characters in AOI, respectively. After applying the refac-



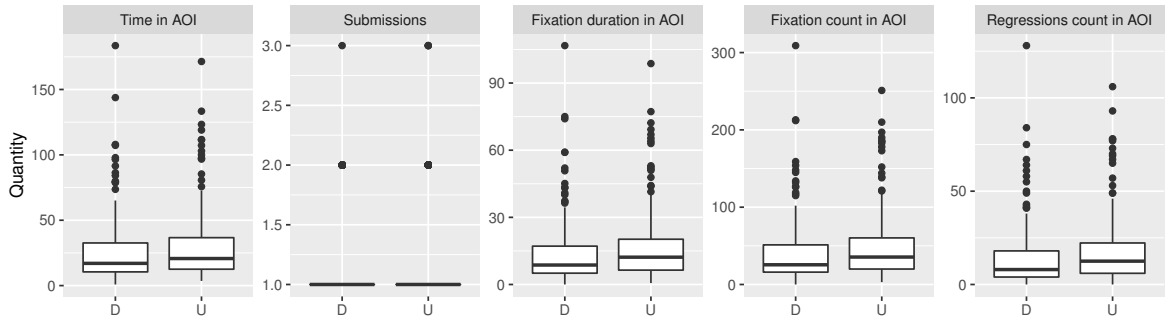

(a) Comparing Statistics for R1




(b) Comparing Statistics for R2




(c) Comparing Statistics for R3




(d) Comparing Statistics for R1, R2, and R3 Combined

Figure 5.6: Comparison between: Disciplined (D) and Undisciplined (U) annotations for R1, R2, and R3 from an isolated and combined perspective involving $P_1$–$P_4$ together.

torings, all tasks from all projects had more characters. Even with more characters, we observed that applying R1 or R3 statistically significantly reduced the time, fixations duration, and fixations count in AOI. In addition, after applying R3, there is a statistically significant increase in the number of correct answers submitted. We did not observe differences in these metrics after R2 was applied.

## 5.4 Discussion of the Interview

In this section, we discuss the qualitative interview with the novices. Besides analyzing their performance in solving code comprehension tasks, we analyze how the answers provided in the interview can help us to validate, understand quantitative results, and complement our discussion on the research questions. In the interview, we asked the subjects to describe 1) their approach used to solve the tasks, 2) their perception of how difficult the tasks were, and 3) what difficulties they had if any. The subjects provided a general approach used in all solved tasks and they were free to share any particular approach used in any specific situation. The same applied to the difficulties, where they were encouraged to point out in the code any area where they had difficulties with the task. With this qualitative feedback, we aimed to better understand how time, attempts, fixation duration, fixation count and regressions count could be better explained through a triangulation of the data.

Based on Corbin and Strauss [113], we adopted the following approach to qualitatively code the interview: In Step 1, we analyzed each whole sentence spoken by the subject during the interview and taking note of the major idea conveyed by this sentence, giving a name to it. In Step 2, we read these names searching for opportunities to group them into distinct categories. In Step 3, we categorized the names by discussing how similar they were according to their properties, for instance, "`#ifdef`" and "directive" could be in the same category, since both refer to "`#ifdef`." In Step 4, we searched for opportunities to link the categories. Given the lack of clear connections between the categories in Step 4, we did not delve deep into them on how they could be used to interpret our results. Thus, we based our results and interpretation on the resulting categories.

When answering the first question, with respect to the used strategy to solve the tasks, we observed that the most common approach adopted consisted of first checking whether the

macro was defined in the program (37 subjects). In addition, 13 subjects mentioned looking at `#ifdef` directives, and, by doing so, 6 subjects mentioned that they could ignore unnecessary parts of the code. This relates to what we call activated and deactivated areas, where, given a macro declared, only one part of the AOI (activated area) gets exercised. Moreover, 16 subjects mentioned also first looking at the function that was on the top of the code, 8 subjects mentioned starting to read the code from the beginning, and 9 subjects mentioned reading the code in a top-down fashion. Furthermore, 9 subjects mentioned a sequential reading pattern, whereas 7 subjects mentioned looking at the end of the code, specifically to the output, and 2 subjects mentioned a bottom-up fashion. Regardless of the order in which they were mentioned, subjects mentioned looking at key parts such as variables (14 subjects) and their assigned values (7 subjects). To summarize, the most frequent terms such as macro, function, variables, and `#ifdef` worked as key elements that guided them in the execution flow of the code. Their influence on time, fixation duration, fixation count, and regressions count should be taken in consideration when investigating code comprehension in the presence of disciplined and undisciplined annotations.

Regarding the second question of the interview, 78% of the subjects found the tasks very easy or easy to solve on a scale of five options, namely, very easy, easy, neuter, difficult, and very difficult. Since the tasks were somewhat simple, these results were not surprising and confirmed the results we had in our $RQ_2$, regarding the number of submissions, which did not present much variation. Even though the majority mentioned that the tasks were easy, they also mentioned having difficulties with some specific tasks.

Regarding our third question of the interview related to their possible difficulties, the most frequent ones related to specific code elements were the following: `if` inside `#ifdef` (18 subjects), boolean expressions (11 subjects), broken lines (9 subjects), confusion regarding the interaction between commands from directives and language constructs (7 subjects), and confusion with `#ifdef` directives specifically (5 subjects). These terms were mentioned when referring to R1 ⟨wrapping function call⟩, R2 ⟨undisciplined `if` conditions⟩, and R3 ⟨alternative `if` statements⟩ in all projects. Other more general terms were frequently evoked. For instance, 11 subjects mentioned that they had difficulties resulting from the fact that they did not pay as much attention as they should, skipping important details, and 9 subjects mentioned confusion disregarding a certain pattern specifically.

According to $RQ_1$, after applying R1 or R3, novices were able to complete tasks faster. Approaches used to solve the tasks and difficulties reported in the interview may shed light on some of the reasons underlying these results. For instance, "broken lines" was reported by the novices as a factor that caused them to face difficulties in completing the tasks. However, by applying R1, the method's parameters that were separated or broken became wrapped as depicted in Figure 5.5(a), which may have helped them to solve tasks faster. On subject mentioned "*I had difficulties in completing the reasoning because of the broken condition*". Another subject mentioned *"When the directive breaks the code, it becomes more difficult*". Similarly, the subjects reported difficulties in reasoning about `if` inside `#ifdef` and about the interaction between commands from directives and language constructs such as `if` and `#ifdef`, `#endif` and `else`. However, by applying R3, the `if` statement is moved from the `#ifdef` body as in Figure 5.5(c), which may have contributed to faster task completion.

According to $RQ_2$, after applying R3, novices provided more correct answers. Even though after applying R3 we have the same median number of submitted answers, whether correct or incorrect, we observe that the data seem more scattered in the box-plot regarding the number of submissions in Figure 5.6(c). Separating an `if` statement with `#ifdef` annotations and placing `else` close to the `#else` seemed to confuse the subjects. Separating these terms by adding an alternative `if` statement, subjects seemed less confused about the correct output.

According to $RQ_3$ and $RQ_4$, after applying R1 or R3, novices exhibited alleviated attention in AOI and in the whole code by reducing both the number of fixations and their duration. These reductions imply less effort in jumping from one place to another in the code, which translates to less visual effort. For instance, subjects mentioned difficulties in dealing with `if` statement inside `#ifdef`, from which we may infer an effort jumping back and forth between those code elements, which translates to a higher number of fixations. They also mentioned confusion with `else` and `#else`. One subject mentioned that *´´it is difficult to look and tell to whom belongs the* `else`", which may have contributed with more fixations, but also longer ones focusing on those elements specifically to make sense of them. Applying R3 separates the `if` body from the `#ifdef` body, which may have impacted the way they concentrated their attention on the AOI. Similarly, after applying R1, the parameters of the function that were separate become wrapped, which impacts the number of jumps

from distinct parts of the same statement distributed over distinct lines.

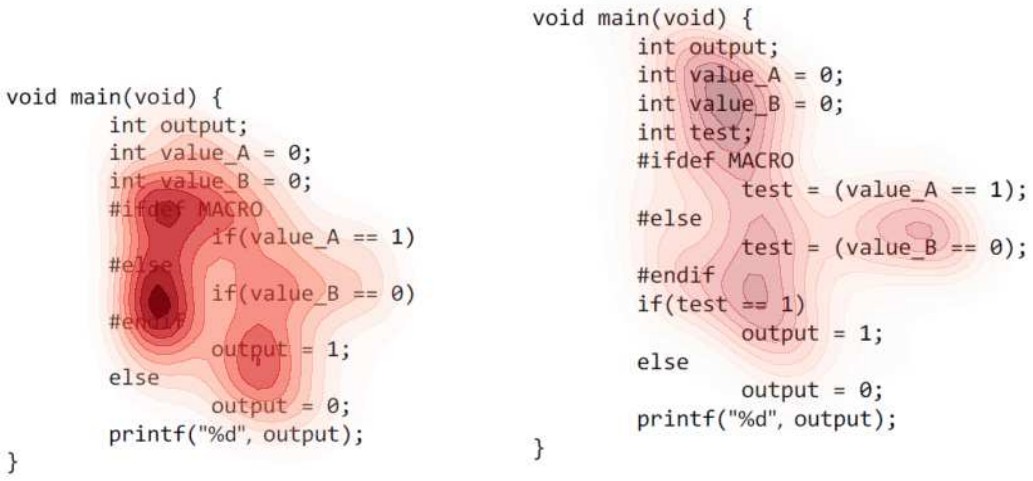
With respect to R3, we observed more pronounced results, with medium effect sizes for four out of five tests. Based on quantitative results supported by the answers of the subjects in the interview, the interaction between `if` and `#ifdef`, `#endif` and `else` might have higher effects. One subject mentioned "confused `else` statement with `#else` statement". This confusion impacts the attention of the subjects, making them look at these statements for a longer time and with more effort.

With respect to R2, we opted for the use of the declaration `(text == 1)` to compare boolean values to `True` or `False` given that, for a novice, it might be simpler to understand instead of making it implicit. With these decisions, we aimed to facilitate the understanding in both undisciplined and disciplined versions. However, it might have an additional impact on the visual behavior, since it adds more elements to pay attention, to compare, and to go back, which can explain why we did not observe clear differences. Another reason might be that, in the refactored version, we have more variables. One subject mentioned having "*difficulties with boolean results such as* `test`". The addition of more elements might indicate more effort in remembering or making sense of their role in the program. In addition, the disciplined version allows one to interpret the code in a more indirect way, which can make it more difficult to understand. This approach was proposed in the catalog to discipline the annotation which passes the result of the boolean expression to a new variable.

### Heatmaps

To complement the analysis and discussion of the qualitative interview, we also investigated the heatmaps for each refactoring, which relate to the attention and visual effort of the subjects. In the heatmaps, the shades of color used represent the relative concentration of gaze points in one area, thus, an area with dark red color reveals a high concentration of fixations, and the longer they are, the darker it gets. To ease the comparison, we have adjusted the colors of the heatmaps by controlling the lightness and saturation of red, thus, normalizing their intensity, and putting the data of the subjects on the same scale. In Figure 5.7, we present two examples of heatmaps. They are based on 16 novices aggregated who solved the same task before and after applying R3 from $P_1$. On the left-hand side in Figure 5.7(a), the subjects solve the task before R3 is applied. On the right-hand side in Figure 5.7(b), other

subjects solve the task after R3 is applied.

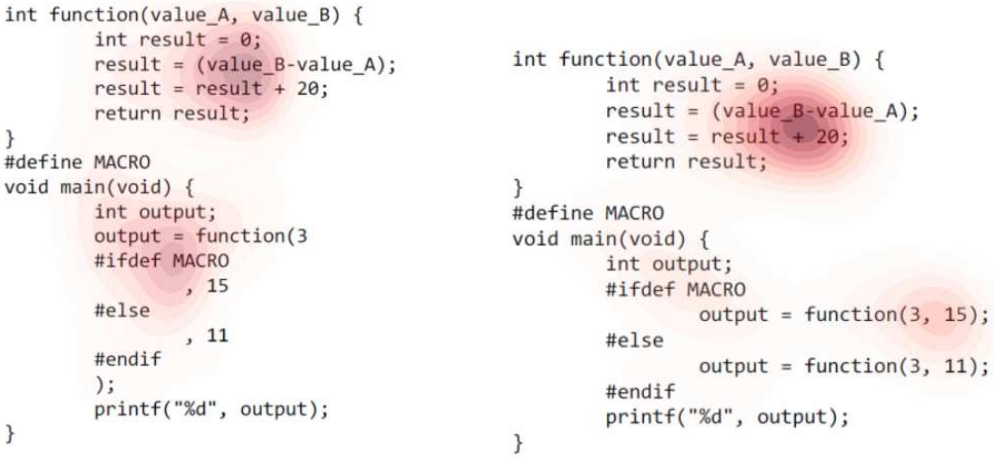



(a) R3 - $P_1$ - Undisciplined

(b) R3 - $P_1$ - Disciplined

Figure 5.7: Comparison between disciplined and undisciplined annotations of R3, $P_1$, indicating distribution and concentration of attention of all subjects who performed it in an aggregated way.

After applying R1 or R2, we could not see clear differences through a visual and manual analysis of the heatmaps. For instance, consider Figure 5.8. In Figure 5.8(a), the fixations are concentrated over two main regions, which are similar to Figure 5.8(b), in the sense that, the subjects pay attention to the correct declarations since the macro was enabled.

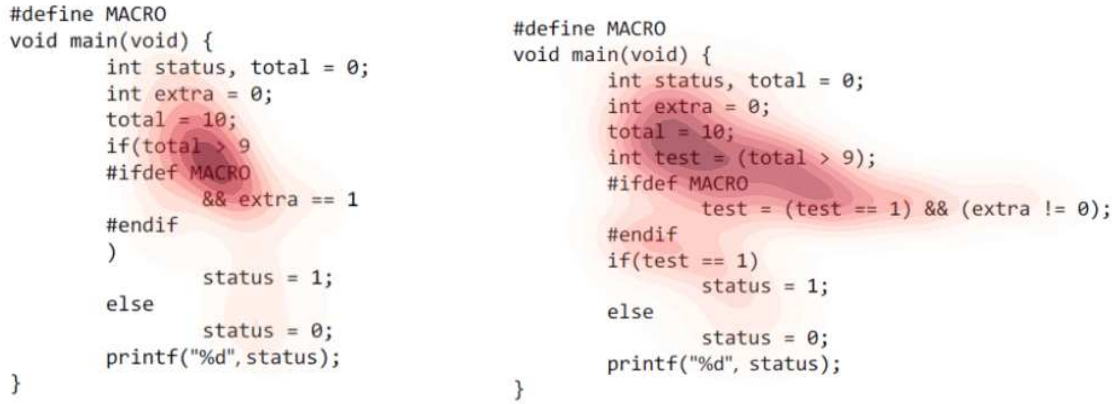



(a) R1 - $P_2$ - Undisciplined

(b) R1 - $P_2$ - Disciplined

Figure 5.8: Comparison between disciplined and undisciplined annotations of R1, $P_2$, indicating distribution and concentration of attention of all subjects who performed it in an aggregated way.

In Figure 5.9(a), the fixations are concentrated over one main region, where the macro is enabled. In Figure 5.9(b), the fixations are more spread over one main broader region, with basically the same intensity of color. We have more elements to look at and be analyzed which might explain why the region is broader.




(a) R2 - $P_1$ - Undisciplined

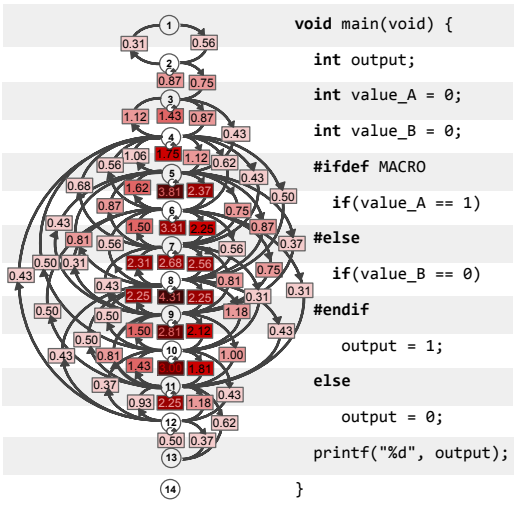
(b) R2 - $P_1$ - Disciplined

Figure 5.9: Comparison between disciplined and undisciplined annotations of R2, $P_1$, indicating distribution and concentration of attention of all subjects who performed it in an aggregated way.

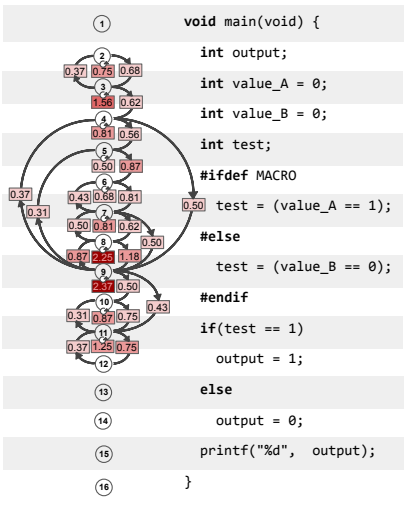
However, for R3, we observed clear differences between before and after applying the refactoring for all tasks, individually and aggregated, for all subjects. For instance, in the heatmap in Figure 5.7(a), in the AOI before applying R3, the colors are darker, which implies more concentrated and longer fixations, especially over `#ifdef`, `#else`, and `#endif`. In future work, we intend to explore the heatmaps in more depth.

### 5.4.1 Analyzing the Saccades

The movement of the eyes from one fixation to another is called a *saccade* [93; 90]. In this sense, they can be useful by pointing to the directions of the fixations in the order in which they occurred. Thus, we mapped the chronological sequences of saccade between the code lines of the tasks performed in order to visualize and better understand the dynamics of the saccade between distinct elements of code before and after applying the refactoring. Then, we built a graph with each node representing a line of code and each edge representing a unidirectional saccade. We present two examples of saccades in Figure 5.10 for all subjects who performed the same task. On the left-hand side in Figure 5.10(a), the subjects solve the task before R3 is applied. On the right-hand side in Figure 5.10(b), subjects solve the task after R3 is applied.




(a) Saccades for R3 - $P_1$ - Undisciplined




(b) Saccades for R3 - $P_1$ - Disciplined

Figure 5.10: Comparison of Graphs with saccades for R3 for $P_1$. The saccades correspond to 16 subjects who performed the same tasks.

The edge weight is based on the idea that many subjects can make the same saccades from one specific line to another while performing the same task, which means that they follow a pattern. We considered only saccades made by at least 1/3 of the subjects for the purpose of simplifying the number of saccades in the graph, making the visualization of the saccades less polluted. We tested with a lower threshold, but the graph visualization became polluted. This strategy also allows us to find patterns since it considers only repetitive saccades across distinct subjects. In Figure 5.10, we observe the gaze saccades for subjects who performed R3 for $P_1$. Each graph represents 16 subjects who performed the same task. For instance, consider the edge from node 1 to node 2 in Figure 5.11 which is the same node 1 and node 2 in the task in Figure 5.10(a). This edge has a weight 0.56. The edge weight consists of the sum of all the same saccades of all subjects who performed it (total of 9 saccades) divided by the total number of subjects who performed the task (16 subjects). So, in this example, six subjects performed at least one saccade each, which satisfied the threshold condition for $1/3$ of the subjects. In addition, subjects 4, 7, and 12 performed the same saccades twice. The edges on the right-hand side of the node in Figure 5.10 represent a progression or a saccade going forward. There are also saccades from one node to itself. The saccades on the left-hand side of the node represent the regressions, which are the saccades going backward.

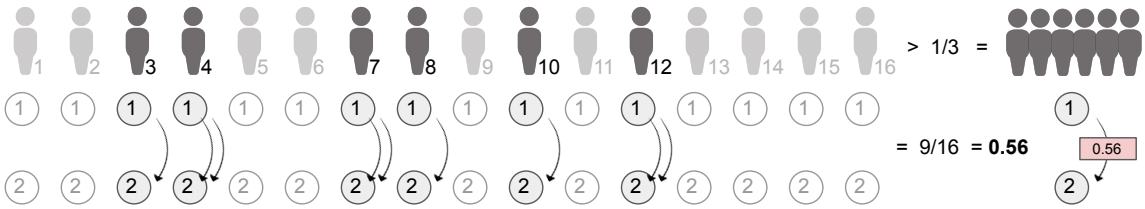
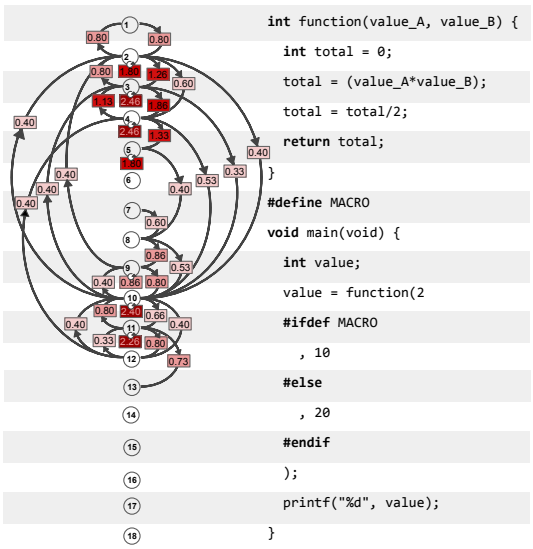



Figure 5.11: Edge weight computation from node 1 to node 2 from all subjects in the task before R3 is applied in Figure 5.10(a). The subjects in darker shades represent all subjects who performed the saccade. The number of arrows from node 1 to node 2 represents the number of saccades.

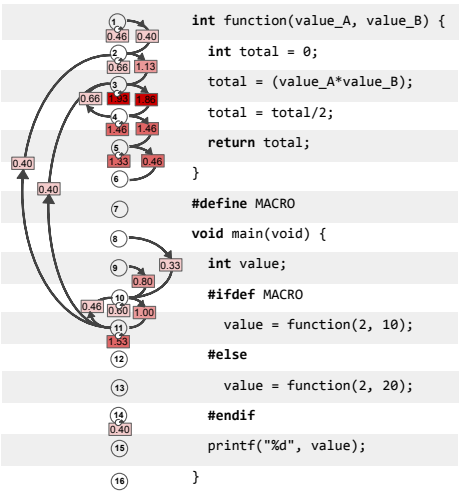
To analyze and compare the graphs in Figure 5.10, we focus on their complexity. Visually, the graph on the left-hand side looks more complex with a higher number of saccades and heavier weights on the edges. After applying R3, the number of saccades was reduced in Figure 5.10. In Figure 5.10(a), we observe considerably more regressions. For instance,

before applying R3, there are 25 regressions, whereas there are 8 regressions after applying R3. We thus recommend applying R3 to reduce the complexity of the graph.

For R1 in Figure 5.12 and R2 Figure 5.13, we did not observe clear differences. In Figure 5.12(b), the saccades seem more concentrated over two regions without so many connections between them as seen in Figure 5.12(a). In Figure 5.13(b), the saccades do not seem to exhibit distinct patterns over the regions compared to in Figure 5.13(a).

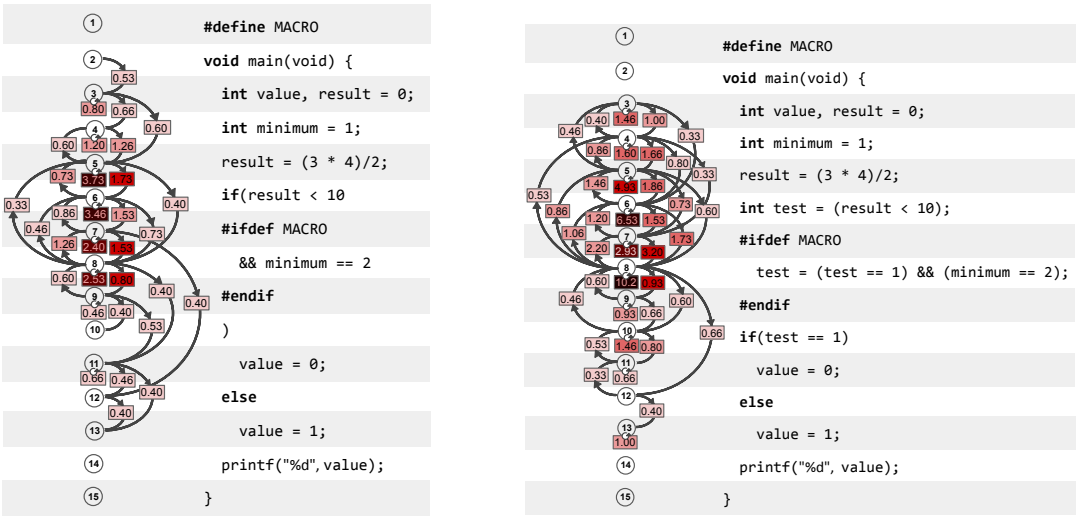



(a) Saccades for R1 - $P_1$ - Undisciplined




(b) Saccades for R1 - $P_1$ - Disciplined

Figure 5.12: Comparison of Graphs with saccades for R1 for $P_1$. The saccades correspond to 16 subjects who performed the same tasks.




(a) Saccades for R2 - $P_2$ - Undisciplined

(b) Saccades for R2 - $P_2$ - Disciplined

Figure 5.13: Comparison of Graphs with saccades for R2 for $P_2$. The saccades correspond to 16 subjects who performed the same tasks.

## 5.5 Threats to Validity

We discuss potential threats to validity: internal validity (Section 5.5.1), external validity (Section 5.5.2), and construct validity (Section 5.5.3).

### 5.5.1 Internal Validity

The environmental location may have influenced the subjects' attention. Given the difficulties in getting more people, we performed the experiment in seven different rooms. However, these environments were similar in terms of being quiet places with minimum distraction and similar lighting and temperatures.

The author's presence may have unintentionally influenced the data because subjects may have felt being observed. The author may also have influenced the subjects to achieve certain outcomes. To mitigate these threats, the author minimized the interaction with the subjects to let them feel free to act and be concentrated on the tasks.

Our camera has limitations. Even carefully calibrating and re-calibrating it, we observed that the fixation data needed some adjustments, which is a threat resulting from the equip-

ment. For instance, we saw parts of the heatmaps with red color over a blank area not touching the code, but a small adjustment was sufficient to correct these cases. These small errors were systematic meaning that all the fixations for the task needed the same adjustment. Right after solving the tasks, we plotted a heatmap of one task, and only when necessary, we shifted the sample points and showed it to the subjects in order to verify whether it best matched their visual intent. In addition, we later discussed this strategy. To minimize the threat of the equipment, we performed a correction of the eye tracking data in some cases, which generated another threat. The correction may influence the position of the gaze points, which may influence our interpretations. We chose to correct the data of some subjects because data pointing to a blank area of the code would influence our interpretations leading to misunderstandings. It is worth mentioning that the median number of pixels that we have used to correct the fixations in $y$-coordinate was 10 pixels and the maximum value was 70 pixels. We did not correct $x$-coordinate. In addition, the generated fixations are available in our replication package [2].

A chair with swiveling capabilities can impair the camera of collecting data or reduce the camera's accuracy. To reduce this threat, we used chairs without swiveling capability. Given the difficulties in arranging the setup in some locations, seven subjects used still chairs with swiveling capabilities.

The duration of the experiment may have influenced the visual effort of the subjects. The six tasks for each subject plus one for warming up have to be taken in consideration. To minimize this threat, we have designed simple tasks so that they could also be solved faster. The maximum amount of time a subject spent on a task was 5 min and 2 seconds, and the median time for all the tasks of all subjects was 36 seconds.

We gave the option to the subjects to keep trying until they answered correctly, but they had the option of quitting at any time without having to provide any reasons for that. We thus compared the number of trials until they answered correctly. However, using this approach, if a subject gives an incorrect answer on the first attempt, she can make more fixations or even longer ones, with more regressions. An alternative approach would be conducting the study such that these metrics were analyzed before they kept trying over and over. However, in our study, this threat is minimized by the fact that 77.6% of the 384 tasks were answered correctly on the first try.

From the total of 384 tasks, three of them were not answered, which corresponds to less than 1%. We used PMM method that imputes missing data using predictive mean matching [58]. Regarding the reliability of this method, PMM generally performed better when the sample size was sufficiently large [64], which we have confidence in our sample with more than 99% of the data.

Following the Latin Square experiment design, we have blocked the set of tasks to control noise. In addition to performing an analysis under a combined perspective of the tasks in all evaluated refactorings, we analyzed the refactorings from an individual perspective. The extent of the impact of such violation of the Latin Square design is not estimated. However, analyzing the data from both perspectives, combined and individual, provides a more nuanced and complete approach to understanding the effects of the evaluated refactorings.

### 5.5.2 External Validity

We had to resort to small tasks for the purpose of fitting the code snippet of each task onto the screen without compromising the accuracy of the data. This may restrict the capacity of generalizing to more complex or larger tasks. However, even in more simple code snippets, we have shown several opportunities to apply the evaluated refactorings. In addition, all our tasks have a template associated with real projects. Medeiros et al. [75] found that 27 out of 63 C real projects contain possibilities of applying the three evaluated refactorings together. However, we need to conduct more studies with more complex tasks to provide evidence regarding those tasks.

Since the majority of subjects in our study were novices in the C programming language, we cannot generalize our results to more experienced developers in C. Other studies have also investigated code comprehension from the perspective of novices as well, revealing an interesting field to be explored [15]. We had a total of 64 subjects in our study, out of which only 10 were experienced subjects. For our analysis and reporting results, we did not filter out these 10 experienced subjects. To ensure that this does not affect the validity of our results, we did a separate analysis where we compared the results of considering all 64 subjects to the results of considering only the 54 novice subjects. We found that the results from both groups of 64 (all subjects) and 54 (only novices) subjects are the same. In the future, we need to conduct further studies with more experienced subjects to better

understand if there are any differences compared to novices.

We have used code snippets written in the C language, which may restrict the generalization capacity to other languages. To limit this threat, we have used constructions that commonly occur in other languages, and all the subjects reported some experience with other languages, which minimized the effect of syntax constructions.

We have performed a "specify the correct output" type of task, in which the subject reads the code and says out loud the correct answer. Thus, it may not generalize to other types of tasks, such as finding bugs or adding features. The font size or font style may have influenced the subject's attention. To reduce this threat, we chose a common font style as well as a size that fitted the screen. All snippets were displayed in the same font size, black colored, and no bold font. The number of macros may also have influenced the visual effort of the subjects, in which they had to reason about enabled and disabled macros to understand which conditions were valid. To minimize this threat, we used only one macro in all the tasks.

### 5.5.3 Construct Validity

Eye tracking metrics similar to the ones employed in our study have been used in other studies for both similar and distinct purposes [77; 15; 103; 7]. For the purpose of investigating code comprehension, time and attempts have been used in isolation [95; 70] and in combination with visual effort [102]. The visual effort has been measured before by separate eye-tracking-based metrics such as fixation duration and fixation count [102; 8]. In addition to fixation-based metrics, regressions have been associated with visual effort [99].

We tried to not influence our subjects' decisions on where to look or for how long, but we may have done so nevertheless, which is a side effect of inviting people to participate in an eye tracking study. We did not inform the subjects about the precise goals of the study to avoid hypothesis guessing, but we informed them that their eyes were being tracked, which may have influenced where or how much they looked at some regions of the code.

## 5.6 Conclusions

In this chapter, we reported on a controlled experiment using an eye tracker camera with 64 subjects who were majoritarily novices in the C language to evaluate the influence of three refactorings that discipline `#ifdef` annotations. There is no consensus on whether developers should use exclusively disciplined annotations. Thus, this study with eye tracking contributes to filling these knowledge gaps showing that even small changes such as adding one or two lines of code, fine-grained refactorings to discipline `#ifdef` annotations associate with differences in code comprehension, and visual effort.

In our results, applying R1 ⟨wrapping function call⟩ or R3 ⟨alternative `if` statements⟩ associated with improvements in the time and visual effort. In addition, applying R3, specifically, associated with improvements in the accuracy. We do not observe statistically significant improvements in time, accuracy, and visual effort in our code comprehension tasks after applying R2 ⟨undisciplined `if` conditions⟩, in isolation. Instead, we observed an increase in time for R2 in both AOIs and the whole code. We also found that applying R1, R2, and R3 in a composite perspective associated with reductions in the total time and visual effort. There are a number of opportunities to apply them in a composite manner in real projects [75].

As future work, we aim to evaluate other refactorings proposed by Medeiros et al. [75]. We aim at performing experiments considering more participants, experienced developers, other types of tasks that add functionalities to the code and fix bugs, a higher number of macros, and other types of annotations. We also intend to explore larger source code files, which can be studied with eye tracking with the addition of a proper tool such as iTrace [50]. This tool allows scrolling or navigation of the content overcoming the limitation of short code snippets for the tasks. Finally, we will consider other eye tracking metrics, such number of blinks, scans, and other metrics based on gaze transitions.

# Chapter 6

# Related Work

In this chapter, we provide an overview of the related work. In Section 6.1 we present the works related to atoms of confusion, in Section 6.2 we present the works related to Extract Method refactoring, and in Section In Section 6.3, we present the works related to `#ifdefs` annotations.

## 6.1 Atoms of Confusion

Gopstein et al. [48] introduced the term "atom of confusion" as the smallest code pattern that can cause misunderstanding in the programmer. They proposed a set of 15 atoms they extracted from the International Obfuscated C Code Contest. They hypothesized that these atoms could cause programmers to misunderstand code. They performed two empirical experiments, one with 73 subjects and the other with 43 mostly students, aiming to find which atoms caused confusion and how much confusion they could reduce by clarifying the atoms. They measured the time it took for programmers to answer correctly and the accuracy of their answers. They found that small C code snippets including atoms of confusion are more difficult to understand than their functionally equivalent clarified versions. Extending their prior work, Gopstein et al. [49] investigated the prevalence of atoms of confusion in the real-world setting. They performed a study involving 14 open-source projects in the C language and found that atoms of confusion are prevalent in real and successful projects. In addition, the presence of these atoms has a correlation with bug-fixing commits and long code comments. We performed a controlled experiment to observe the impact of the obfuscated code

containing atoms of confusion and the clarified code on the novices' code comprehension. However, we focused on Python programming language and, besides time and accuracy, we investigated the eye tracking metrics. In addition, we were more conservative in our programs than in their studies, using more meaningful names for the variables. This approach arguably closer to a practical scenario.

In a more recent work, Gopstein et al. [47] performed a study with 14 human subjects, including both professionals and students, aiming to better understand and scrutinize their prior studies on atoms of confusion. According to them, the precision and accuracy can only tell the outcome of programmers' performance, but not how or why programmers behaved in a certain way. They used a think-aloud methodology to collect data and then performed a qualitative analysis. They found that correct hand-evaluations do not imply understanding, which means that a subject can answer correctly and still be confused. Similarly, incorrect evaluations do not imply misunderstanding. With the sole use of the accuracy, these sources of confusion would otherwise go unnoticed. Going beyond accuracy, we used an eye tracker to assess the visual effort of the subjects. Eye tracking allowed us to better understand their visual behavior while solving the tasks, which could give insights into how or why programmers behaved in a certain way. In addition, we also performed a qualitative interview to get personal feedback.

Medeiros et al. [73] conducted an investigation to better understand the relevance and prevalence of atoms of confusion in C open-source projects. They used a mixed research method approach, which comprised mining repositories of 50 C open-source projects followed by a survey with 97 developers with experience in the C language. They found that atoms of confusion are indeed prevalent in open-source projects, with more than 109K occurrences of the 12 atoms. In addition, according to developers' opinions, only some atoms are perceived to cause misunderstandings. Instead of basing on the opinions of experienced developers in C, we conducted a controlled experiment to quantitatively and qualitatively assess the performance of the human subjects who were novices in Python. In addition to time and accuracy, we collect eye tracking metrics to have a better understanding of the effect of the atoms. To complement our quantitative data, additionally, we perform interviews to get feedback from the subjects.

Yeh et al. [121] conducted an experiment using an EEG device to measure the cogni-

tive 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. Still, in the domain of physiological data, we focused on changes in eye movements instead of brain activity. Eye movements have also been used to investigate cognitive load, however, we explored in more depth the visual effort regarding atoms in Python language.

Langhout and Aniche [66] replicated the work of Gopstein et al. [48], however, in the Java programming language. After deriving a set of atoms of confusion for Java, they performed an experiment with 132 computer science novices. They found that atoms of confusion can cause confusion among novice software developers. Extending this idea, Mendes et al. [78] proposed a tool named BOHR (The Atoms of Confusion Hunter) to detect atoms of confusion in Java systems. The tool was able to detect eight out of 13 types of atoms pointed out as confusing by Langhout and Aniche [66]. We also investigated the potential of atoms to negatively influence the code comprehension of novices. However, we did so in Python language and from the perspective of the eye tracking measures.

Castor [18] proposed a structured definition of atoms of confusion, examined factors that make them confusing, and presented a preliminary catalog of atoms of confusion for the Swift programming language. Based on the prior studies [48; 49], Castor defined an atom as precisely identifiable, likely to cause confusion, replaceable by another pattern that is less like to cause confusion, and indivisible. He also identified sources that make atoms confusing such as little-known and less common constructs, which include *Conditional Operator* and *Assignment as Value*. We used the definition already proposed [48; 49] and investigated empirically the effects of obfuscated and clarified programs on code comprehension in Python language.

Schröter et al. [94] conducted a literature review to investigate how researchers address code comprehension in their studies. Among their findings, they found that the source code and program behavior are the mostly addressed parts of code comprehension in their empirical studies. Our work consists of an empirical study that comprises a comparison of programs following distinct styles.

Stefik and Siebert [111] studied the influence of programming language syntax on the

novices' comprehension. As their tasks, the novices had to rate the intuitiveness of several programming language constructs. Among the findings of the study, syntactic choices made in commercial programming languages are more intuitive to novices than others, and variations in syntax influence novice accuracy rates when they are starting to program. We also explored the context of novices, however, we considered just one language, and we used objective metrics while novices solved the code task.

Obaidellah et al. [82] conducted a systematic mapping study on eye tracking experiments focusing on code scenario. They found that the main areas of research include program comprehension and debugging, non-code comprehension, collaborative programming, and requirements traceability research. In addition, they found that most of the subjects in the experiments were students and faculty members from institutions. In our controlled experiment, we focus on code comprehension involving novices.

## 6.2 Extract Method

Cedrim et al. [19] studied the impact of refactorings, such as Extract and Inline Method, on code smells through static code metrics. They performed a longitudinal study observing how refactorings impacted 13 types of code smells along the version histories of 23 projects. They found that 57% of the refactorings, including the Inline Method, was neutral in the sense that they did not reduce the occurrences of smells, and 33.3%, including the Extract Method, were negative, meaning that they induced the introduction of new smells instead. We studied the impact of Extract Method from a dynamic perspective. We consider the human visual effort of novices in Java through a controlled experiment. In our study, the Extract Method presented positive results in four tasks and negative results in two tasks.

Hora and Robbes [56] conducted a study to characterize the Extract Method refactoring. They mined 124 Java systems and investigated 70K instances of the Extract Method focusing on the aspects magnitude, content, transformation, size, and degree. They found that (i) the Extract Method is the third most frequent refactoring; (ii) Extracted Methods concentrate on operations related to the creation, validation, and setup; (iii) methods that are targets of the extractions are 2.2x longer than the average ones, and they are reduced by one statement after the extraction; and (iv) single method extraction represents most of the cases. We also

investigated the Extract Method, however, focusing on the extent of its impact on the code comprehension of novices in Java.

Silva et al. [107] investigated why developers refactor their code. They monitored Java projects to detect recently applied refactorings and asked the developers their reasons to refactor the code. They found the Extract Method to be the most frequently applied. The three most common reasons were to reuse code, introduce an alternative method signature, and improve readability. We performed a controlled experiment on the impact of two refactorings and interview the subjects on their perceptions. Their answers helped us to better understand the code comprehension factors that affect their understanding of the tasks.

Sharafi et al. [103] studied the impact of two code styles on code comprehension, namely, camel case and underscore. They measured time, accuracy, and visual effort of the subjects through eye tracking. They found significantly less time and lower visual effort with the underscore style. Sharafi et al. [101] investigated the same two styles on code comprehension, however, measuring the impact of the subjects' gender on the time, accuracy, and visual effort. Overall, no differences were observed. In our study, we compared two code styles with similar metrics—time, number of attempts, and visual effort—however in a different context.

## 6.3 `#ifdef` Annotations

Medeiros et al. [75] conducted a survey with 246 experienced developers to access their perception of the proposed refactorings. The majority of the developers reported having at least five years of experience with C preprocessors. They sent a questionnaire to the subjects with six templates presented as pairs: on the left-hand side, they presented the original code from a real C project and, on the right-hand side, the refactored version of the original code. They asked the subjects which version they preferred, whether the original or the refactored one. Among the refactorings, they evaluated R1 ⟨wrapping function call⟩, R2 ⟨undisciplined `if` conditions⟩, and R3 ⟨alternative `if` statements⟩ of our study. In their study, the rate of preferences for R1, R2, and R3 were 90.3%, 70.4%, and 64.8%, respectively. In contrast, in our work, we have focused on novices rather than on experienced developers. The majority of the subjects reported having one year or less of experience with C programming language.

In addition, we have conducted a controlled experiment in which the novices had to solve a set of proposed tasks. We investigated eye tracking metrics to evaluate R1, R2, and R3 with respect to time, accuracy, fixation duration, fixation count, and regressions count.

Our study and the one conducted by Medeiros et al. [75] are distinct in the following characteristics: research questions, the experience of the developers, tasks used, empirical method, metrics, and threats to validity. These differences are summarized in Table 6.1. The differences shown in Table 6.1 may explain the differences in the conclusions. However, we need to conduct further studies to better understand the reasons for some differences.

Table 6.1: Summarizing the comparison between the study conducted by Medeiros et al. and our study.

| | **Medeiros et al. [75]** | **Our study** |
|---|---|---|
| Common RQs | — | — |
| Distinct RQs | What is the number of possibilities to apply the refactorings in practice?<br>What opinion do developers have on the catalog of refactorings in practice?<br>Do the refactorings of the catalog preserve program behavior? | To what extent do disciplined annotations affect task completion time?<br>To what extent do disciplined annotations affect task accuracy?<br>To what extent do disciplined annotations affect visual effort? |
| Common Findings | Developers prefer applying R1<br>Developers prefer applying R3 | Applying R1 associated with improvements in time and visual effort<br>Applying R3 associated with improvements in time, accuracy, and visual effort |
| Distinct Findings | Developers prefer applying R2 | Applying R2 did not associate with improvements in time, accuracy, or visual effort |
| Experience | Experienced developers in C programming language | Novices in C programming language |
| Tasks | Non-executable code templates | Executable code snippets |
| Empirical Method | Online survey with subjects not being observed | Controlled experiment with subjects being observed |
| Metrics | Subjective opinions and preferences | Objective metrics: time, accuracy, fixation duration, fixation count, and regressions count |

| | | |
|---|---|---|
| Threats to Validity | Simple code snippets, incompleteness of catalog, programming language, some undisciplined directives different from the practice | Environment location, camera limitations, chair setup, time for the experiment, answers' submission, simple tasks, developers' experience, programming language, type of task, eye tracking metrics' representativeness |

In addition to the survey, Medeiros et al. [75], submitted patches with the evaluated refactorings. Six patches using R1 were submitted, and all patches were accepted. Five patches were submitted using R2 and 80% of them were accepted, and five patches were submitted using R3 and 80% of them were accepted. These results indicate a higher rate of acceptance of R1. Applying R1 or R3 in isolation associated with improvements in time and visual effort in our study.

Medeiros et al. [72] interviewed 40 developers with at least five years of experience and conducted a survey with 202 developers with different levels of experience regarding conditional directives usage, to understand common problems with the C preprocessor such as code understanding, maintainability, and error proneness. Developers affirmed that they checked only a few configurations of the source code when they were testing their implementations. The study showed that C preprocessor had problems, such as faults, inconsistencies, code quality, and incomplete testing, making it a "hell." The survey and interview focused on the perception of the developers, which included experienced subjects. Differently, we conducted an eye tracking study and focused on analyzing the performance, code comprehension, and visual attention of novices. From this perspective, even in simple tasks, we observed that novices had difficulties comprehending code with undisciplined annotations, mentioning terms such as broken lines, statements, and syntax. We observed that tasks were easier to comprehend using the disciplined version by the correlation with the improvements in accuracy after applying R3.

Fenske et al. [39] have conducted a controlled study involving both an experiment and questionnaires with 521 experienced developers to understand the impact of refactoring C preprocessor directives. The evaluated refactorings were called discipline directive, extract alternative function, and unify compile-time and runtime-time variability. They evaluate coarse-grained transformations converting from undisciplined to disciplined annotations instead of evaluating a single fine-grained transformation, such as the ones we evaluated in our work (see Figure 5.5). Their comprehension tasks are distinct from ours comprising larger

snippets with more directives. In addition, multiple choices are presented to the subjects. For instance, among multiple statements about the code, the subjects had to select the correct one. Moreover, the subjects had to configure their selection so that a certain line would be executed. They mainly investigated how the perception of the developers aligned with their objective of comprehension performance. According to their results, comprehension performance worsened in terms of correctness when the subjects worked on code with refactored directives. However, in their perception, the refactored code was more comprehensible and easier to work. In contrast, we have presented smaller snippets with one directive to the subjects. We have configured the directive by enabling or disabling the macro. Then, we asked the subjects an open-ended question regarding the correct output of the snippet. In addition, we have performed a controlled experiment using eye tracking with novices. In our results, we recommend applying R1 or R3 for novices.

Thus, our study and the one conducted by Fenske et al. [75] are distinct in the following characteristics: experience of the developers, tasks used, answer submissions method, empirical method, and metrics. These differences are summarized in Table 6.2 and may explain the differences in the conclusions. However, we need to conduct further studies to better understand the reasons for some differences.

Table 6.2: Summarizing the comparison between the study conducted by Fenske et al. and our study.

| | **Fenske et al. [39]** | **Our study** |
|---|---|---|
| Experience | Experienced developers in C programming language | Novices in C programming language |
| Tasks | Larger snippets with more directives | Short snippets with one `#ifdef` |
| Answer submission | Multiple options | Open-ended without multiple options |
| Empirical Method | Online survey and experiment | Controlled experiment |
| Metrics | Subjective preferences, time, and accuracy | Objective metrics: time, accuracy, fixation duration, fixation count, and regressions count |

Schulze et al. [95] conducted a controlled experiment to analyze the effect of disciplined and undisciplined annotations on program comprehension. The subjects were undergradu-

ates with less programming experience than experienced developers. The study addressed this topic by measuring correctness and response time for solving a set of tasks. The results of the study did not reveal any statistically significant differences between disciplined and undisciplined annotations from a program comprehension perspective. In addition to time and accuracy, but distinctly from their study, we have measured the fixation duration, fixation count, and regressions count, which allowed us to access the subject's visual effort in solving tasks. The eye tracker allowed us to understand code comprehension from the analysis of these additional dimensions. Furthermore, similar to their study, the majority of the subjects of our study consisted of undergraduates. Differently from their results, we have shown statistically significant differences for the evaluated refactorings with disciplined annotations, indicating that the composition of refactorings evaluated associated with improvements in time, fixation duration, fixation count, and regressions count.

Malaquias et al. [70] compared undisciplined and disciplined annotations by investigating the influence of disciplined annotations on maintenance tasks. They performed the study with undergraduates with three to five semesters of experience with programming. Their results showed that undisciplined annotations are more time-consuming and error-prone, disagreeing with Schulze et al. [95]. For R1 or R3, the results of Malaquias et al. [70] align with ours in the sense that disciplined annotations associate with improvements in task completion time. In addition, disciplined annotations associate with improvements in accuracy after applying R3 in the context of novices. For the composition of three evaluated refactorings, their results also align with ours for the time, fixation duration, fixation count and regressions count perspective. Regarding accuracy, we did not reject the null hypothesis for the number of submissions. Notice that our tasks are simple. In our study with an eye tracker camera, we are able to explore other dimensions besides time and accuracy, and quantify developer's difficulties by measuring time in specific areas, as well as effort with visual attention.

Melo et al. [76] presented a controlled experiment predominantly with graduate students. All subjects had Java programming experience, and several of them had industrial experience. They aimed to quantify the impact of variability on the time and accuracy in finding bugs in configurable systems. They only considered disciplined annotations. By exploring these dimensions, they found that the time of bug finding decreases linearly with the degree of variability. In addition, it is harder to identify the exact set of affected configurations than

finding the bug in the first place. They mentioned difficulties in reasoning about several configurations. In our study, we explore time, accuracy, and other additional dimensions, but in another type of task. We did not present many configuration options, only one macro enabled and disabled, and simple tasks. Even in simple tasks, we observed that it became easier to find the correct output after applying R3, removing undisciplined annotations. In the qualitative feedback, novices mentioned difficulties in reasoning about broken statements, which were removed by applying refactorings.

Aiming to understand how developers debug code in the presence of code variability, Melo et al. [77] carried out an experiment by using an eye tracker predominantly with graduate students. All subjects had Java programming experience, and several of them had industrial experience. The main results indicate that variability increases debugging time for code fragments with variability. Besides performing a distinct type of task, so-called "find the bug," they have focused only on disciplined annotations. They observed that variability prolongs the initial scan in the task of finding defects. We focused on refactorings to discipline annotations to understand how novices specify the correct output. We put extra effort into minimizing potential threats regarding eye tracker camera usage. For instance, we systematized the program style with fewer lines and larger size, for easy reproducibility, and avoided chairs with swiveling capability, which showed potential to impact data quality.

The use of an eye tracker camera has been traditionally applied in the context of cognitive psychology for the purpose of studying the reading and information processing at the cognitive level [90]. For instance, using an eye tracker, Crosby and Stelovsky [29] observed differences between reading source code and reading prose. However, they did not investigate refactorings. We analyzed how disciplined annotations affect the way novices read and comprehend code.

In the programming language context, eye tracking allowed researchers to understand a variety of tasks, such as code comprehension and code debugging [82]. For instance, Sharafi et al. [103] investigated the influence of identifier styles (camel case and underscore) on the speed and accuracy of comprehending source code. No differences regarding accuracy were observed in this context. Nevertheless, results indicate a significant improvement in time and lower visual effort with the underscore style. In our study, we considered similar metrics—time, accuracy, fixation duration and fixation count—but in another context. Binkley et al. [8]

also studied the influence of identifier styles on code reading and comprehension. With an eye tracker, they found that camel case shows to be more advantageous. Likewise, we perform a comparison between two types of code styles, namely disciplined and undisciplined annotations, aiming to find which one is more advantageous. However, to analyze visual effort, in addition to regressions count, we explored fixation duration and fixation count inside AOI, in the whole code, and in specific areas such as activated and deactivated ones.

Turner et al. [115] presented a study to analyze the effect of the choice of the programming language, namely C++ and Python, on code comprehension. The metrics they used consisted of accuracy, time, and visual effort. The former metric concerns the rate one looks at buggy lines. In our work, we also cover accuracy, time, and visual effort, which we relate to fixation duration, fixation count, and regressions count. Binkley et al. [9] studied the effect of identifier length on the ability of programmers to recall. Their results suggested that longer names reduce correctness and take more time to recall from memory. In our domain, the eye tracking metrics gave us additional insights. Table 6.3 summarizes these works.

Table 6.3: Related works. In column "Eye," we refer to whether eye tracking was used or not. In column "Ann." we specify the annotations in which U refers to Undisciplined and D refers to Disciplined. In column "Exp." we specify whether the subjects were experienced or not, in which "Yes" refers to experienced and "No" refers to not experienced. In column Goal, the symbol (∗) refers to a survey while (†) refers to a controlled experiment.

| Study | Eye | Ann. | Exp. | Metrics | Goal | Finding |
|---|---|---|---|---|---|---|
| Medeiros et al. [72] | No | — | Yes | — | Access developers' perception on C preprocessor usage (∗) | Despite the criticism of C preprocessor, they use it nonetheless |
| Schulze et al. [95] | No | U, D | No | Time and accuracy | Analyze the effect of annotations on program comprehension (†) | No differences between disciplined and undisciplined annotations |
| Malaquias et al. [70] | No | U, D | No | Time and accuracy | Analyze whether annotation influences maintenance tasks (†) | Undisciplined annotations are more time-consuming and error prone |
| Melo et al. [77] | Yes | D | Yes | Time, accuracy, fixations and saccades | Study how developers debug with variability (†) | Debugging time increases with variability |
| Melo et al. [76] | No | D | Yes | Time and accuracy | Analyze the impact of variability on metrics (†) | Time of bug finding decreases with the degree of variability |

| | | | | | | |
|---|---|---|---|---|---|---|
| Ours | Yes | U, D | No | Time, accuracy, fixation duration, fixation count, and regressions count | Evaluate whether R1, R2, and R3 affect comprehension and visual attention (†) | Applying R1 or R3 reduce time, fixation duration and count. |

# Chapter 7

# Conclusions

This dissertation addressed knowledge gaps in the understanding of the impact of refactorings on code comprehension with eye tracking (Chapter 1); provided a background on the underlying subject (Chapter 2); presented a controlled experiment on the impact of clarified atoms of confusion with 32 novices in Python (Chapter 3), a controlled experiment on the impact of Extract Method refactoring with 32 novices in Java (Chapter 4), and a controlled experiment on the impact of `#ifdef` annotations with 64 majoritarialy novices in the C language (Chapter 5); and a relationship with related works (Chapter 6).

We learned that code comprehension consists of a complex and nuanced phenomenon, and several factors and details have to be considered. Controlled experiments are central to understanding this phenomenon and usually investigate several factors, such as the code, tasks, metrics, and subjects. Particularly, one has to select a set of suitable metrics to measure the code understanding, and in this field, time and accuracy are commonly employed. Despite being easy to assess, their potential is limited. For instance, they cannot provide information about processes happening during the execution of the task, whether from the visual or brain perspective of the developer. In this work, we have explored the potential of adding eye tracking metrics to reveal nuances and provide insights distinct from other code metrics approaches.

In the context of novices dealing with undisciplined `#ifdef` annotations, the sole use of time and accuracy makes it difficult to evaluate whether the code comprehension has improved after applying fine-grained transformations. However, with an eye tracker, besides the effects on time and accuracy, 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 the 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. 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 to the extent of 200% in the visual metrics.

Besides providing insights into the visual effort, we discuss code comprehension from a distinct perspective. With an eye tracker, we approach comprehension by triangulating time, attempts, and visual effort. We can infer potential areas with bottlenecks in the code tasks by adding eye tracking. 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.

With an eye tracker, our study revealed an impact of the refactorings that could not be captured by static code metrics, such as:

- How much time the subjects spent in the area of interest – time spent specifically in the region that contains the atom, extracted method, or `#ifdef`. No other tool or approach can allow us to measure how much time a subject spends in specific regions of the code while reading the entire code. For instance, we found that the clarified version of the code containing *Operator Precedence* reduced the time spent in the AOI by 38.6%. On the other hand, the *Multiple Var. Assignment* increased the time in AOI by 30.1%.

- How many times subjects fixate in the code and/or an area of interest – the number of fixations specifically in the region that contains the atom, extracted method, or `#ifdef`.

- For how long the subjects fixate in the code and/or an area of interest – the duration of the fixations performed while focusing specifically in the region that contains the atom, extracted method, or `#ifdef`. The longer the fixation, the more time processing the information, which relates to code comprehension.

- How many times the subjects need to go back in the code and/or an area of interest – the number of eye movement regressions performed in the code or specifically in the region that contains the atom, extracted method, or `#ifdef`.

- Study *how* and measure the *extent* of 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. For instance, with the clarified version of the atom *Operator Precedence*, the fixation duration in the AOI was reduced by 34.1% while the fixations count was reduced by 32.3%. The clarified version of the code containing the *True or False Evaluation* was associated with reductions in the regressions count by 47.3%. On the other hand, the clarified version of the code containing the *Multiple Variable Assignment* was associated with an increase of 60% in the number of regressions.

- Study *how* and measure the *extent* of the impact of clarified atoms, extracted methods, and disciplined #ifdef annotations on the code reading. In the clarified version of

the code containing the *Multiple Var. Assignment*: Splitting the assignments between two lines leads the subjects to make 60% more regressions inside the area of interest, 38.4% more fixations with 22.9% more duration. In addition, subjects make more eye movement transitions between the two split lines containing the assignments and the lines of code that later use them.

Additional contributions of this dissertation include a set of learned lessons that work as a framework for conducting eye tracking studies in the context of behavior-preserving code changes on the code comprehension of novice programmers. These lessons include knowledge about the design of programs such as the size and style, the use of specific eye tracking metrics, and parameters for the fixations detection, among others. In addition, we contribute with visualizations for the visual effort in terms of transitions between code regions and horizontal/vertical regressions in the code.

## 7.1 Future Work

In this section, we present other empirical studies to be performed after the final defense of this dissertation. We aim to conduct studies on code comprehension similar to the ones presented here but on distinct scenarios, namely, with experienced developers (Section 7.1.1), with a higher number of code changes (Section 7.1.2), with other types of tasks (Section 7.1.3), with other types of metrics (Section 7.1.4), and more reading patterns (Section 7.1.5).

### 7.1.1 Experienced Developers

We aim to investigate the same behavior-preserving code transformations presented in the three chapters for the purpose of understanding whether the clarified atoms, extracted methods, and disciplined #ifdef annotations improve the code comprehension from the point of view of experienced developers.

When solving the same tasks, we may expect to see distinct results from the ones obtained in our study with novices. Experienced developers may exhibit less completion time, higher accuracy, and less visual effort when compared to novices in the context. However, we aim

to contribute by investigating the extent of those possible differences.

### 7.1.2 Higher Number of Code Changes

We have investigated the use of disciplined annotations with only one macro, whether enable or disabled. However, increasing the number of macros, we expect the increase the complexity of the task, once the subjects have to reason about how multiple macros can interact with one another. For instance, in Figure 7.1(a) we have three macros. If all macros get defined, all the code blocks get activated. But we can define just some of them resulting in more possibilities. The more possibilities we have, the higher the impact on the code comprehension we can expect.

```
#define M1
#define M2
#define M3
int main() {

#ifdef M1
   int x;
#endif

#ifdef M2
   int y;
#endif

#ifdef M3
   int z;
#endif
}
```

(a) Three macros enabled

```
public class Main {
  static int m1(int num) {
    //code here..
    return num
  }
  static int m2(int num) {
    //code here..
    return num
  }
  static int m3(int num) {
    //code here..
    return num
  }
  public static void main
                  (String[] args) {
    int number = 3;
    int a = m1(number)
    int b = m2(number)
    int c = m3(number);
  }
}
```

(b) Three methods extracted

```
itens = 5
total = 0

price = itens * 10 if itens == 5
                    else itens * 2

if (price % 2):
  value = price
else:
  value = price * 2

if(False or True and True):
   total = value

print(total)
```

(c) Three atoms of confusion

Figure 7.1: Programs with more than one code change.

In addition, we have investigated code comprehension using only one method extracted and one atom of confusion in each program. We aim to investigate the impact of the combination of more code changes such as more method extractions (Figure 7.1(b)) and more atoms of confusion (Figure 7.1(c)). We aim to investigate it from the perspective or novices as well as experienced subjects in programming.

Increasing the number of code changes requires using larger code. We intend to explore the `iTrace` [50], a tool that allows scrolling or navigation of the content overcoming the

limitation of short code snippets in eye tracking. In addition, to overcome the limitations of static images of code, we intend to use `iTrace-Atom` tool [35]. The tool allows tracking gaze and editing information over source code, accompanied by `gazel` (gaze edit evolution), a Python data processing library to analyze the data collected by `iTrace-Atom`.

### 7.1.3 Other Types of Tasks

We investigated code comprehension based on tasks in which the subjects had to read the code and specify its correct output. However, we are aware that there are other types of maintenance tasks, such as finding bugs, fixing bugs, and adding a functionality to the code.

In this way, we aim to investigate the clarified atoms, extracted methods, and disciplined #ifdef annotations presented in the three chapters to improve code comprehension from the point of view of novices in the context of tasks extracted from real projects. However, the accuracy is based not on specifying the correct output, but on finding the bug or adding the functionality correctly.

The type of task may have an influence on visual effort. For instance, on a task where the subject has to find the bug, we might expect a more thorough search looking specifically for undeclared variables, missing elements, or other uncommon patterns. We aim to investigate if there is any impact and the extent of that possible impact.

### 7.1.4 Other Metrics

We resorted to the most popular metrics in eye tracking to investigate comprehension, however, with the appropriate equipment, we could investigate other visual metrics such as pupil dilation or blink rate. Eye blinks can be distinguished into three types: reflexive, voluntary, and endogenous. Reflexive blinks can occur in a response to external stimuli, are designed to protect the eye, and are involuntary. Voluntary blinks are blinks invoked voluntarily. Blinks that occur in absence of any physical stimulus or intent are called endogenous blinks [112].

The endogenous blinks are influenced by perceptual and information processing. When more visual attention is required by a task, the endogenous eye blinks are inhibited and delayed to a moment where the visual demand is reduced [112; 31]. For instance, during reading, short bursts or individual blinks are likely to occur at the attention breaks such as at

the end of a sentence or the end of a line [51]. Another study demonstrated that spontaneous blinks were significantly reduced during a stimulus-processing period of high attention and were facilitated immediately after the end of the stimulus [83].

We aim to investigate how the clarified atoms, extracted methods, and disciplined #ifdef annotations affect the number of endogenous blinks. This might indicate that certain types of behavior-preserving code changes may be associated with more visual demands. Since tasks that take longer to solve correlate with more blinks, we normalize by computing the number of blinks per minute, which makes a fair comparison between the types of annotations.

We have other eye tracking metrics to indicate visual effort such as the ones based on the saccades. For instance, the number of saccades and saccade duration are metrics whose definitions are identical to the corresponding fixations-based metrics and have similar interpretations in relation to the visual effort [99]. Thus, we opted for the use of fixations-based metrics in the studies. The average saccade amplitude could be promising to reveal nuances in the extraction of a method, for instance. It indicates the angular distance that the eye travels by summing the distances between consecutive fixations. However, average saccade amplitude must be used with care, because the amplitude is completely dependent on the size of the stimulus and of its elements [99]. In the method extraction scenario, It can reveal how the change of focus between the original method and the extracted impacts the amplitude of the saccades, which is a relevant nuance to be investigated. However, we need better understand what we can learn from this.

### 7.1.5 Reading Patterns

We have investigated reading patterns in our data to complement the quantitative analysis of the performance of the subjects. We used the chronological order of the fixations and their positions to identifying a sequence of visited regions for each program of each subject. Based on the sequences of visited regions, we mined and manually unidentified common sequences. However, we need to better understand whether or how these patterns can be used to characterize the subjects' experience, whether we can compare reading patterns, and how they can be generalized.

# Bibliography


[1] Alfred Vaino Aho, Ravi Sethi, and Jeffrey David Ullman. Compiladores: Princıpios, técnicas e ferramentas. *LTC, Rio de Janeiro, Brasil*, 1995.

[2] José Aldo, Rohit Gheyi, Márcio Ribeiro, Sven Apel, Baldoino Fonseca, Vander Alves, Flávio Medeiros, and Alessandro Garcia. Evaluating refactorings for disciplining #ifdef annotations using eye tracking with novices (artifacts). At https://github.com/josealdo/EMSE21-ifdefs-with-eye-tracking, 2021.

[3] Venera Arnaoudova, Massimiliano Di Penta, and Giuliano Antoniol. Linguistic Antipatterns: What They are and How Developers Perceive Them. *Empirical Software Engineering*, 21(1):104–158, 2016.

[4] Eran Avidan and Dror G Feitelson. Effects of Variable Names on Comprehension: An Empirical Study. In *Proceedings of the International Conference on Program Comprehension*, ICPC'17, pages 55–65. IEEE, 2017.

[5] Victor Basili, G. Caldiera, and H. Rombach. The Goal Question Metric Approach. *Encyclopedia of software engineering*, pages 528–532, 1994.

[6] Jackson Beatty. Task-evoked pupillary responses, processing load, and the structure of processing resources. *Psychological bulletin*, 91(2):276, 1982.

[7] Roman Bednarik and Markku Tukiainen. An Eye-tracking Methodology for Characterizing Program Comprehension Processes. In *Proceedings of the Symposium on Eye Tracking Research & Applications*, ETRA'06, pages 125–132, 2006.

[8] Dave Binkley, Marcia Davis, Dawn Lawrie, Jonathan Maletic, Christopher Morrell,

and Bonita Sharif. The Impact of Identifier Style on Effort and Comprehension. *Empirical Software Engineering*, 18(2):219–276, 2013.

[9] Dave Binkley, Dawn Lawrie, Steve Maex, and Christopher Morrell. Identifier length and limited programmer memory. *Science of Computer Programming*, 74(7):430–445, 2009.

[10] Tanja Blascheck, Kuno Kurzhals, Michael Raschke, Michael Burch, Daniel Weiskopf, and Thomas Ertl. State-of-the-art of visualization for eye tracking data. In *EuroVis - STARs*. The Eurographics Association, 2014.

[11] Agnieszka Aga Bojko. Informative or misleading? heatmaps deconstructed. In *International conference on human-computer interaction*, pages 30–39. Springer, 2009.

[12] George Box, J. Stuart Hunter, and William G. Hunter. *Statistics for Experimenters*. Wiley-Interscience, 2005.

[13] Larissa Braz, Rohit Gheyi, Melina Mongiovi, Márcio Ribeiro, Flávio Medeiros, and Leopoldo Teixeira. A change-centric approach to compile configurable systems with #ifdefs. In *Proceedings of the International Conference on Generative Programming: Concepts and Experiences*, GPCE'16, pages 109–119, 2016.

[14] Ruven Brooks. Towards a theory of the comprehension of computer programs. *International Journal of Man-machine Studies*, 18(6):543–554, 1983.

[15] Teresa Busjahn, Roman Bednarik, Andrew Begel, Martha Crosby, James H Paterson, Carsten Schulte, Bonita Sharif, and Sascha Tamm. Eye Movements in Code Reading: Relaxing the Linear Order. In *Proceedings of the International Conference on Program Comprehension*, ICPC'15, pages 255–265. IEEE, 2015.

[16] Teresa Busjahn, Carsten Schulte, and Andreas Busjahn. Analysis of Code Reading to Gain More Insight in Program Comprehension. In *Proceedings of the Koli Calling International Conference on Computing Education Research*, Koli Calling'11, pages 1–9, 2011.

[17] Simon Butler, Michel Wermelinger, Yijun Yu, and Helen Sharp. Exploring the Influence of Identifier Names on Code Quality: An Empirical Study. In *Proceedings of the European Conference on Software Maintenance and Reengineering*, CSMR'10, pages 156–165. IEEE, 2010.

[18] Fernando Castor. Identifying Confusing Code in Swift Programs. In *Proceedings of the CBSoft Workshop on Visualization, Evolution, and Maintenance*, VEM'18, 2018.

[19] Diego Cedrim, Alessandro Garcia, Melina Mongiovi, Rohit Gheyi, Leonardo Sousa, Rafael de Mello, Baldoino Fonseca, Márcio Ribeiro, and Alexander Chávez. Understanding the Impact of Refactoring on Smells: A Longitudinal Study of 23 Software Projects. In *Proceedings of the Joint Meeting on Foundations of Software Engineering*, ESEC/FSE'17, page 465–475, 2017.

[20] Diego Cedrim, Leonardo Sousa, Alessandro Garcia, and Rohit Gheyi. Does refactoring improve software structural quality? a longitudinal study of 25 projects. In *Proceedings of the Brazilian Symposium on Software Engineering*, SBES'16, pages 73–82, 2016.

[21] Gary Charness, Uri Gneezy, and Michael A Kuhn. Experimental methods: Between-subject and within-subject design. *Journal of Economic Behavior & Organization*, 81(1):1–8, 2012.

[22] Shyam R Chidamber and Chris F Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[23] Paul Clements. Being proactive pays off. *IEEE Software*, 19(4):28–, 2002.

[24] Paul Clements and Linda Northrop. Software product lines: Practice and patterns, addison wesley. *Reading, MA*, 2001.

[25] Norman Cliff. Dominance Statistics: Ordinal Analyses to Answer Ordinal Questions. *Psychological bulletin*, 114(3):494, 1993.

[26] Jacob Cohen. *Statistical Power Analysis for the Behavioral Sciences*. Academic press, 2013.

[27] Myra B Cohen, Matthew B Dwyer, and Jiangfan Shi. Interaction testing of highly-configurable systems in the presence of constraints. In *Proceedings of the International Symposium on Software Testing and Analysis*, ISSTA'07, pages 129–139, 2007.

[28] Martha Crosby, Jean Scholtz, and Susan Wiedenbeck. The Roles Beacons Play in Comprehension for Novice and Expert Programmers. In *Workshop of the Psychology of Programming Interest Group*, PPIG'02, page 5, 2002.

[29] Martha Crosby and Jan Stelovsky. How do we read algorithms? A case study. *Computer*, 23(1):25–35, 1990.

[30] José Aldo Silva da Costa, Rohit Gheyi, Márcio Ribeiro, Sven Apel, Vander Alves, Baldoino Fonseca, Flávio Medeiros, and Alessandro Garcia. Evaluating Refactorings for Disciplining #ifdef Annotations: An Eye Tracking Study with Novices. *Empirical Software Engineering*, 26(5):1–35, 2021.

[31] Peter J De Jong and Harald Merckelbach. Eyeblink frequency, rehearsal activity, and sympathetic arousal. *International Journal of Neuroscience*, 51(1-2):89–94, 1990.

[32] Rafael de Mello, José Aldo da Costa, Benedito de Oliveira, Márcio Ribeiro, Baldoino Fonseca, Rohit Gheyi, Alessandro Garcia, and Willy Tiengo. Decoding confusing code: Social representations among developers. In *Proceedings of the International Workshop on Cooperative and Human Aspects of Software Engineering*, volume 1 of *CHASE'21*, pages 11–20, 2021.

[33] Benedito de Oliveira, Márcio Ribeiro, José Aldo Silva da Costa, Rohit Gheyi, Guilherme Amaral, Rafael de Mello, Anderson Oliveira, Alessandro Garcia, Rodrigo Bonifácio, and Baldoino Fonseca. Atoms of Confusion: The Eyes Do Not Lie. In *Proceedings of the Brazilian Symposium on Software Engineering*, SBES'20, pages 243–252, 2020.

[34] Michael Ernst, Greg Badros, and David Notkin. An empirical analysis of C preprocessor use. *IEEE Transactions on Software Engineering*, 28(12):1146–1170, 2002.

[35] Sarah Fakhoury, Devjeet Roy, Harry Pines, Tyler Cleveland, Cole S Peterson, Venera Arnaoudova, Bonita Sharif, and Jonathan Maletic. gazel: supporting source code

edits in eye-tracking studies. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 69–72. IEEE, 2021.

[36] Jean-Marie Favre. Preprocessors from an abstract point of view. In *Proceedings of the Working Conference on Reverse Engineering*, WCRE'96, pages 287–296. IEEE, 1996.

[37] Jean-Marie Favre. Understanding-in-the-large. In *Proceedings of the International Workshop on Program Comprehension*, IWPC'97, pages 29–38. IEEE, 1997.

[38] Dror G Feitelson. Considerations and pitfalls in controlled experiments on code comprehension. In *Proceedings of the International Conference on Program Comprehension*, ICPC'21, pages 106–117. IEEE Computer Society, 2021.

[39] Wolfram Fenske, Jacob Krüger, Maria Kanyshkova, and Sandro Schulze. #ifdef Directives and Program Comprehension: The Dilemma between Correctness and Preference. In *Proceedings of the International Conference on Software Maintenance and Evolution*, ICSME'20, 2020.

[40] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 2018.

[41] Thomas Fritz, Andrew Begel, Sebastian Müller, Serap Yigit-Elliott, and Manuela Züger. Using psycho-physiological measures to assess task difficulty in software development. In *Proceedings of the International Conference on Software Engineering*, ICSE'14, pages 402–413, 2014.

[42] Critina Gacek and Michalis Anastasopoules. Implementing product line variabilities. In *Proceedings of the Symposium on Software Reusability: Putting Software Reuse in Context*, SSR'11, pages 109–117. ACM, 2001.

[43] Niels Galley, sDirk Betz, and Claudia Biniossek. Fixation Durations: Why are They so Highly Variable. *Advances in Visual Perception Research*, pages 83–106, 2015.

[44] Alejandra Garrido and Ralph Johnson. Refactoring C with conditional compilation. In *Proceedings of the International Conference on Automated Software Engineering*, ASE'03, pages 323–326. IEEE, 2003.

[45] Alejandra Garrido and Ralph Johnson. Embracing the C preprocessor during refactoring. *Journal of Software: Evolution and Process*, 25(12):1285–1304, 2013.

[46] Joseph H Goldberg and Xerxes P Kotval. Computer interface evaluation using eye movements: methods and constructs. *International Journal of Industrial Ergonomics*, 24(6):631–645, 1999.

[47] Dan Gopstein, Anne-Laure Fayard, Sven Apel, and Justin Cappos. Thinking Aloud about Confusing Code: A Qualitative Investigation of Program Comprehension and Atoms of Confusion. In *Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE'20, pages 605–616, 2020.

[48] Dan Gopstein, Jake Iannacone, Yu Yan, Lois DeLong, Yanyan Zhuang, Martin K.-C. Yeh, and Justin Cappos. Understanding Misunderstandings in Source Code. In *Proceedings of the Joint Meeting on Foundations of Software Engineering*, ESEC/FSE'17, pages 129–139, 2017.

[49] Dan Gopstein, Hongwei Henry Zhou, Phyllis Frankl, and Justin Cappos. Prevalence of Confusing Code in Software Projects: Atoms of Confusion in the Wild. In *Proceedings of the International Conference on Mining Software Repositories*, ICSMR'18, pages 281–291, 2018.

[50] Drew Guarnera, Corey Bryant, Ashwin Mishra, Jonathan Maletic, and Bonita Sharif. iTrace: Eye tracking infrastructure for development environments. In *Proceedings of the Symposium on Eye Tracking Research & Applications*, ETRA'18. ACM, 2018.

[51] Arthur Hall. The origin and purposes of blinking. *The British journal of ophthalmology*, 29(9):445, 1945.

[52] Dan Hansen and Qiang Ji. In the eye of the beholder: A survey of models for eyes and

gaze. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 32(3):478–500, 2009.

[53] Felienne Hermans. Peter Hilton on Naming. *IEEE Software*, 34(3):117–120, 2017.

[54] Frouke Hermens and Sunčica Zdravković. Information extraction from shadowed regions in images: An eye movement study. *Vision research*, 113:87–96, 2015.

[55] Kenneth Holmqvist, Marcus Nyström, Richard Andersson, Richard Dewhurst, Halszka Jarodzka, and Joost Van de Weijer. *Eye Tracking: A Comprehensive Guide to Methods and Measures*. OUP Oxford, 2011.

[56] Andre Hora and Romain Robbes. Characteristics of Method Extractions in Java: A Large Scale Empirical Study. *Empirical Software Engineering*, 25(3):1798–1833, 2020.

[57] Robert JK Jacob and Keith S Karn. Eye tracking in human-computer interaction and usability research: Ready to deliver the promises. In *The Mind's Eye*, pages 573–605. Elsevier, 2003.

[58] Anil Jadhav, Dhanya Pramod, and Krishnan Ramanathan. Comparison of Performance of Data Imputation Methods for Numeric Dataset. *Applied Artificial Intelligence*, 33(10):913–933, 2019.

[59] José Aldo Silva da Costa, Rohit Gheyi, Fernando Castor, Pablo Roberto, Márcio Ribeiro, Baldoino Fonseca. "Seeing Confusion Through a New Lens: On the Impact of Atoms on Novices' Code Comprehension (Artifacts)". At https://github.com/josealdo/atoms-of-confusion-with-eye-tracking, 2022.

[60] Marcel A Just and Patricia A Carpenter. A Theory of Reading: From Eye Fixations to Comprehension. *Psychological review*, 87(4):329, 1980.

[61] Christian Kästner, Paolo G Giarrusso, Tillmann Rendel, Sebastian Erdweg, Klaus Ostermann, and Thorsten Berger. Variability-aware parsing in the presence of lexical macros and conditional compilation. In *Proceedings of the International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA'11, pages 805–824, 2011.

[62] Yoshio Kataoka, Takeo Imai, Hiroki Andou, and Tetsuji Fukaya. A quantitative evaluation of maintainability enhancement by refactoring. In *International Conference on Software Maintenance*, ICSME'02, pages 576–585. IEEE, 2002.

[63] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. An empirical study of refactoring challenges and benefits at microsoft. *IEEE Transactions on Software Engineering*, 40(7):633–649, 2014.

[64] Kristian Kleinke. Multiple Imputation under Violated Distributional Assumptions: A Systematic Evaluation of the Assumed Robustness of Predictive Mean Matching. *Journal of Educational and Behavioral Statistics*, 42(4):371–404, 2017.

[65] Maren Krone and Gregor Snelting. On the inference of configuration structures from source code. In *Proceedings of the International Conference on Software Engineering*, ICSE'94, pages 49–57. IEEE, 1994.

[66] Chris Langhout and Maurício Aniche. Atoms of Confusion in Java. In *Proceedings of the International Conference on Program Comprehension*, ICPC'21, pages 25–35. IEEE, 2021.

[67] Dawn Lawrie, Henry Feild, and David Binkley. Quantifying Identifier Quality: an Analysis of Trends. *Empirical Software Engineering*, 12(4):359–388, 2007.

[68] Dawn Lawrie, Christopher Morrell, Henry Feild, and David Binkley. What's in a Name? a Study of Identifiers. In *Proceedings of the International Conference on Program Comprehension*, ICPC'06, pages 3–12. IEEE, 2006.

[69] Jörg Liebig, Christian Kästner, and Sven Apel. Analyzing the discipline of preprocessor annotations in 30 million lines of C code. In *Proceedings of the International Conference on Aspect-Oriented Software Development*, AOSD'11, pages 191–202, 2011.

[70] Romero Malaquias, Márcio Ribeiro, Rodrigo Bonifácio, Eduardo Monteiro, Flávio Medeiros, Alessandro Garcia, and Rohit Gheyi. The Discipline of Preprocessor-Based Annotations – Does #ifdef TAG n't #endif Matter. In *Proceedings of the International Conference on Program Comprehension*, ICPC'17, pages 297–307. IEEE, 2017.

[71] Thomas J McCabe. A complexity measure. *IEEE Transactions on software Engineering*, (4):308–320, 1976.

[72] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Sarah Nadi, and Rohit Gheyi. The Love/hate Relationship with the C Preprocessor: An Interview Study. In *Proceedings of the European Conference on Object-Oriented Programming*, ECOOP'15, pages 999–1022. ACM, 2015.

[73] Flávio Medeiros, Gabriel Lima, Guilherme Amaral, Sven Apel, Christian Kästner, Márcio Ribeiro, and Rohit Gheyi. An Investigation of Misunderstanding Code Patterns in C Open-source Software Projects. *Empirical Software Engineering*, 24(4):1693–1726, 2019.

[74] Flávio Medeiros, Márcio Ribeiro, and Rohit Gheyi. Investigating preprocessor-based syntax errors. In *Proceedings of the International Conference on Generative Programming: Concepts & Experiences*, GPCE'13, pages 75–84. ACM, 2013.

[75] Flávio Medeiros, Márcio Ribeiro, Rohit Gheyi, Sven Apel, Christian Kästner, Bruno Ferreira, Luiz Carvalho, and Baldoino Fonseca. Discipline matters: Refactoring of preprocessor directives in the #ifdef hell. *IEEE Transactions on Software Engineering*, 44(5):453–469, 2018.

[76] Jean Melo, Claus Brabrand, and Andrzej Wasowski. How Does the Degree of Variability Affect Bug Finding? In *Proceedings of the International Conference on Software Engineering*, ICSE'16, pages 679–690. ACM, 2016.

[77] Jean Melo, Fabricio Batista Narcizo, Dan Witzner Hansen, Claus Brabrand, and Andrzej Wasowski. Variability Through the Eyes of the Programmer. In *Proceedings of the International Conference on Program Comprehension*, ICPC'17, pages 34–44, Piscataway, NJ, USA, 2017. IEEE Press.

[78] Wendell Mendes, Windson Viana, and Lincoln Rocha. BOHR - Uma Ferramenta para a Identificação de Átomos de Confusão em Códigos Java. In *Workshop de Visualização, Evolução e Manutenção de Software*, VEM'21, pages 41–45. SBC, 2021.

[79] Gail C Murphy, Mik Kersten, and Leah Findlater. How are Java software developers using the Elipse IDE? *IEEE Software*, 23(4):76–83, 2006.

[80] Emerson Murphy-Hill, Chris Parnin, and Andrew P Black. How We Refactor, and How We Know It. *IEEE Transactions on Software Engineering*, 38(1):5–18, 2011.

[81] Marcus Nyström and Kenneth Holmqvist. An Adaptive Algorithm for Fixation, Saccade, and Glissade Detection in Eyetracking Data. *Behavior research methods*, 42(1):188–204, 2010.

[82] Unaizah Obaidellah, Mohammed Al Haek, and Peter C-H Cheng. A Survey on the Usage of Eye-tracking in Computer Programming. *ACM Computing Surveys (CSUR)*, 51(1):1–58, 2018.

[83] Jihoon Oh, So-Yeong Jeong, and Jaeseung Jeong. The timing and temporal patterns of eye blinking are dynamically modulated by attention. *Human movement science*, 31(6):1353–1365, 2012.

[84] Delano Oliveira, Reydne Bruno, Fernanda Madeiral, and Fernando Castor. Evaluating Code Readability and Legibility: An Examination of Human-centric Studies. In *Proceedings of the International Conference on Software Maintenance and Evolution*, ICSME'20, 2020.

[85] Jonhnanthan Oliveira, Rohit Gheyi, Melina Mongiovi, Gustavo Soares, Márcio Ribeiro, and Alessandro Garcia. Revisiting the refactoring mechanics. *Information and Software Technology*, 110:136–138, 2019.

[86] William F. Opdyke. Refactoring Object-oriented Frameworks. Ph.D. thesis, University of Illinois at Urbana-Champaign Champaign, IL, USA, 1992.

[87] David Lorge Parnas. On the design and development of program families. *IEEE Transactions on software engineering*, (1):1–9, 1976.

[88] Troy Pearse and Paul Oman. Experiences developing and maintaining software in a multi-platform environment. In *Proceedings of the International Conference on Software Maintenance*, ICSME'97, pages 270–277. IEEE, 1997.

[89] Klaus Pohl, Günter Böckle, and Frank J van Der Linden. *Software product line engineering: foundations, principles and techniques*. Springer Science & Business Media, 2005.

[90] Keith Rayner. Eye movements in reading and information processing. *Psychological bulletin*, 85(3):618, 1978.

[91] Keith Rayner. Eye Movements in Reading and Information Processing: 20 Years of Research. *Psychological bulletin*, 124(3):372, 1998.

[92] Erica Sadun and Doug Gregor. Remove c-style for-loops with conditions and incrementers. Swift Programming Language Evolution, proposal SE-0007, "At https://github.com/apple/swift-evolution/blob/main/proposals/0007-remove-c-style-for-loops.md".

[93] Dario Salvucci and Joseph Goldberg. Identifying Fixations and Saccades in Eye-tracking Protocols. In *Proceedings of the Symposium on Eye Tracking Research & Applications*, ETRA'00, pages 71–78, 2000.

[94] Ivonne Schröter, Jacob Krüger, Janet Siegmund, and Thomas Leich. Comprehending Studies on Program Comprehension. In *Proceedings of the International Conference on Program Comprehension*, ICPC'20, pages 308–311. IEEE, 2017.

[95] Sandro Schulze, Jörg Liebig, Janet Siegmund, and Sven Apel. Does the Discipline of Preprocessor Annotations Matter?: A Controlled Experiment. In *Proceedings of the International Conference on Generative Programming: Concepts & Experiences*, GPCE '13, pages 65–74, 2013.

[96] Teresa M Shaft and Iris Vessey. The relevance of application domain knowledge: The case of computer program comprehension. *Information Systems Research*, 6(3):286–299, 1995.

[97] Samuel Sanford Shapiro and Martin B. Wilk. An Analysis of Variance Test for Normality (Complete Samples). *Biometrika*, 52(3/4):591–611, 1965.

[98] Zohreh Sharafi, Alessandro Marchetto, Angelo Susi, Giuliano Antoniol, and Yann-Gaël Guéhéneuc. An empirical study on the efficiency of graphical vs. textual representations in requirements comprehension. In *Proceedings of the International Conference on Program Comprehension*, ICPC'13, pages 33–42. IEEE, 2013.

[99] Zohreh Sharafi, Timothy Shaffer, Bonita Sharif, and Yann-Gaël Guéhéneuc. Eye-tracking Metrics in Software Engineering. In *Proceedings of the Asia-Pacific Software Engineering Conference*, APSEC'15, pages 96–103. IEEE, 2015.

[100] Zohreh Sharafi, Bonita Sharif, Yann-Gaël Guéhéneuc, Andrew Begel, Roman Bednarik, and Martha Crosby. A Practical Guide on Conducting Eye Tracking Studies in Software Engineering. *Empirical Software Engineering*, 25(5):3128–3174, 2020.

[101] Zohreh Sharafi, Zéphyrin Soh, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. Women and Men—Different but Equal: On the Impact of Identifier Style on Source Code Reading. In *Proceedings of the International Conference on Program Comprehension*, ICPC'12, pages 27–36. IEEE, 2012.

[102] Bonita Sharif, Michael Falcone, and Jonathan Maletic. 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, pages 381–384. ACM, 2012.

[103] Bonita Sharif and Jonathan Maletic. An Eye Tracking Study on Camelcase and Under_score Identifier Styles. In *Proceedings of the International Conference on Program Comprehension*, ICPC'10, pages 196–205. IEEE, 2010.

[104] David J Sheskin. *Handbook of Parametric and Nonparametric Statistical Procedures*. crc Press, 2020.

[105] Janet Siegmund. Program comprehension: Past, present, and future. In *Proceedings of the International Conference on Software Analysis, Evolution, and Reengineering*, volume 5 of *SANER'16*, pages 13–20. IEEE, 2016.

[106] Janet Siegmund, Christian Kästner, Sven Apel, Chris Parnin, Anja Bethmann, Thomas Leich, Gunter Saake, and André Brechmann. Understanding understanding source

code with functional magnetic resonance imaging. In *Proceedings of the International Conference on Software Engineering*, ICSE'14, pages 378–389. ACM, 2014.

[107] Danilo Silva, Nikolaos Tsantalis, and Marco Tulio Valente. Why We Refactor? Confessions of GitHub Contributors. In *Proceedings of the International Symposium on Foundations of Software Engineering*, FSE'16, page 858–870, New York, NY, USA, 2016. Association for Computing Machinery.

[108] José Aldo Silva da Costa, Rohit Gheyi, and Márcio Ribeiro. The Impact of the Extract Method Revealed through the Eyes of Novices in Java. At https://github.com/josealdo/refactorings-with-eye-tracking, 2022.

[109] Henry Spencer and Geoff Collyer. #ifdef considered harmful, or portability experience with C news. In *USENIX Summer. USENIX Association*, pages 185–197, 1992.

[110] Thomas A Standish. An essay on software reuse. *IEEE Transactions on Software Engineering*, (5):494–497, 1984.

[111] Andreas Stefik and Susanna Siebert. An Empirical Investigation into Programming Language Syntax. *Transactions on Computing Education*, 13(4):1–40, 2013.

[112] John A Stern, Larry C Walrath, and Robert Goldstein. The endogenous eyeblink. *Psychophysiology*, 21(1):22–33, 1984.

[113] Anselm Strauss and Juliet Corbin. *Basics of Qualitative Research Techniques*. Thousand Oaks, CA: Sage publications, 1998.

[114] Alexander Strukelj and Diederick C Niehorster. One page of text: Eye movements during regular and thorough reading, skimming, and spell checking. *Journal of Eye Movement Research*, 11(1), 2018.

[115] Rachel Turner, Michael Falcone, Bonita Sharif, and Alina Lazar. An eye-tracking study assessing the comprehension of C++ and Python source code. In *Proceedings of the Symposium on Eye Tracking Research and Applications*, ETRA'14, pages 231–234. ACM, 2014.

[116] Hidetake Uwano, Masahide Nakamura, Akito Monden, and Ken-ichi Matsumoto. Analyzing individual performance of source code review using reviewers' eye movement. In *Proceedings of the Symposium on Eye Tracking Research and Applications*, ETRA'06, pages 133–140. ACM, 2006.

[117] Frank J Van der Linden, Klaus Schmid, and Eelco Rommes. *Software product lines in action: the best industrial practice in product line engineering*. Springer Science & Business Media, 2007.

[118] Jilles Van Gurp, Jan Bosch, and Mikael Svahnberg. On the notion of variability in software product lines. In *Proceedings Working IEEE/IFIP Conference on Software Architecture*, WICSA'05, pages 45–54. IEEE, 2001.

[119] Anneliese Von Mayrhauser and A Marie Vans. From program comprehension to tool requirements for an industrial environment. In *Proceedings of the Workshop on Program Comprehension*, WPC'93, pages 78–86. IEEE, 1993.

[120] Susan Wiedenbeck. Beacons in Computer Program Comprehension. *International Journal of Man-Machine Studies*, 25(6):697–709, 1986.

[121] Martin K-C Yeh, Yu Yan, Yanyan Zhuang, and Lois Anne DeLong. Identifying Program Confusion Using Electroencephalogram Measurements. *Behaviour & Information Technology*, pages 1–18, 2021.

# Appendix A

# Ethics Committee Approval

## PARECER CONSUBSTANCIADO DO CEP

**DADOS DO PROJETO DE PESQUISA**

**Título da Pesquisa:** Uma Avaliação sobre Refatoramentos para #ifdefs com Rastreamento Ocular
**Pesquisador:** José Aldo Silva da Costa
**Área Temática:**
**Versão:** 1
**CAAE:** 24042719.8.0000.5182
**Instituição Proponente:** Universidade Federal de Campina Grande
**Patrocinador Principal:** Financiamento Próprio

**DADOS DO PARECER**

**Número do Parecer:** 3.781.870

**Apresentação do Projeto:**
Trata-se de projeto que tem como instituição proponente a Universidade Federal de Campina Grande, com duas instituições coparticipantes: Universidade Federal de Alagoas e Universidade Estadual da Paraíba. É um estudo experimental que visa avaliar se refatoramentos para disciplinar as anotações #ifdefs melhoram a qualidade e compreensão do código usando o rastreamento ocular. Ainda no contexto de variabilidade, o projeto também objetiva investigar a dificuldade dos desenvolvedores para encontrar defeitos ou fraquezas no código.
Os sujeitos do estudo serão 200 pessoas com conhecimento de programação.

**Objetivo da Pesquisa:**
Objetivo Primário: analisar programas em linguagem C com #ifdefs usando rastreamento ocular com o objetivo de avaliar a qualidade de refatoramentos, dificuldade de se encontrar defeitos em anotações indisciplinadas ou disciplinadas, e dificuldade de se encontrar fraquezas no código. Para tal, será medido o tempo, quantidade de varreduras, e número de tentativas na execução de tarefas de evolução de software do ponto de vista dos desenvolvedores no contexto da compreensão do código.
Objetivo Secundário: (i) Identificar qual tipo de anotações proporciona ao desenvolvedor identificar mais saídas corretas, encontrar mais erros, e apontar mais defeitos; (ii) Investigar qual dos tipos de anotações proporciona ao desenvolvedor identificar mais fraquezas no código; (iii) Apontar

quais os refatoramentos que proporcionam ao desenvolvedor realizar mais tarefas de evolução em menor tempo e com maior acurácia; (iv) Averiguar que tipo de anotações proporciona mais produtividade na adição de novas funcionalidades no código.

**Avaliação dos Riscos e Benefícios:**

Riscos: O pesquisador informa que como os participantes resolverão problemas de programação, os mesmos podem eventualmente sentir cansaço, estresse ou aborrecimento ao se deparar com dificuldades. Para minimizar esses riscos, os participantes poderão, a qualquer momento, desistir de um determinado problema ou até de participar do estudo. Em casos de dúvidas ou dificuldades, um dos pesquisadores estará disponível para ajudar a minimizá-los. Além disso, o pesquisador se compromete a utilizar códigos simples nos experimentos. Esse aspecto diminui o esforço do participante na compreensão da tarefa. São códigos pequenos, com estilo de fontes legíveis de tamanhos relativamente grandes o que minimiza desconfortos. Pequenas pausas são realizadas entre cada tarefa também. Além disso, será utilizada uma câmera de rastreamento ocular (Tobii eye tracker 4C) remota, acoplada a um computador. A câmera é desenvolvida para jogadores com o intuito de prover uma experiência imersiva e comercializável em larga escala. Portanto, os riscos ou desconforto promovido pela câmera já são minimizados para o público de jogadores. Mas além destes, serão tomados cuidados adicionais. Quanto à montagem do equipamento, acontece antes do estudo e o participante não tem contato físico com a câmera, que fica posicionada à uma distância de 50cm-100cm. A câmera emite um raio infravermelho que se projeta sobre os olhos do participante e permite computar onde está o foco visual do participante. Luzes vermelhas na câmera podem causar desconforto, contudo, a câmera não será posicionada em frente dos olhos do participante. A câmera é posicionada abaixo do foco visual, em uma região periférica, e os participantes serão orientados a não olhar diretamente para a câmera. De acordo com o fabricante, existem pessoas que possuem epilepsia fotossensível e são suscetíveis a ataques epiléticos ou perda de consciência quando expostas a alguns tipos de luzes piscando ou padrões de luz no seu dia-a-dia. Isso pode acontecer mesmo que a pessoa não tenha histórico médico de epilepsia ou nunca tenha tido crises epilépticas. Isso pode acontecer com telas de TV, alguns jogos de fliperama e lâmpadas fluorescentes tremeluzentes. Essas pessoas podem sofrer uma convulsão enquanto assistem a certas imagens ou padrões em um monitor, ou mesmo quando expostas às fontes de luz de um rastreador ocular. De acordo com o fabricante, estima-se que cerca de 3-5% das pessoas com epilepsia tenham esse tipo de epilepsia fotossensível. Muitas pessoas com

epilepsia fotossensível experimentam uma "aura" ou sentem sensações estranhas antes da ocorrência da convulsão. Para evitar que isso ocorra, os pesquisadores se comprometem a ser muito cautelosos em perguntar ao participante se tem ou já teve algum histórico médico de epilepsia ou já passou por qualquer tipo de situação parecida com as descritas acima no seu dia-a-dia envolvendo uso de TVs, imagens de computador, ou luzes piscantes. Se o participante mencionar que sim, por precaução, não participará do estudo. Para aqueles que realizarem o estudo, ou seja, não apresentam histórico de epilepsia, serão questionados se estão sentindo sensações estranhas durante o estudo por causa da luz emitida. Caso isso ocorra, por razões de segurança, será removido do estudo.

Benefícios: Este trabalho beneficia diretamente a pesquisadores na academia, bem como a desenvolvedores que trabalham em empresas de programação ou startups. Para os pesquisadores, uma metodologia que utiliza métricas baseadas no olhar podem revelar nuances e fornecer insights distintos de outras métricas para transformações refinadas. Por exemplo, para transformações de granularidade fina, é difícil avaliar se a redução de uma linha de código extra ou a adição de uma variável extra realmente melhora a qualidade do código. No entanto, com métricas baseadas no olhar, observamos se os desenvolvedores fazem menos varreduras no código, quais trechos exigem mais atenção, ou quais construções são mais complexas. Essa metodologia tem o potencial de aprofundar a discussão sobre as vantagens de anotações disciplinares e adicionar uma nova perspectiva de análise além do tempo e da precisão utilizada em estudos anteriores. Vale a pena salientar que os resultados deste trabalho também podem contribuir para confirmar ou descartar a necessidade da proposição de novos refatoramentos para lidar com o uso indisciplinado de #ifdefs, bem como da avaliação de cada um deles na prática para entender qual versão (disciplinada vs. indisciplinada) é melhor. Essa pesquisa também tem potencial de beneficiar desenvolvedores para que sejam mais produtivos, e dessa forma, beneficiar empresas para que gastem menos recursos financeiros em atividades de evolução de código. Através da análise de métricas como varreduras no código, varreduras em áreas específicas do código, frequência de entradas nessas áreas, tempo para primeira varredura, temos meios para inferir padrões de leitura de código, o que pode nos fornecer informações úteis sobre como os desenvolvedores compreendem o código. Esses insights podem fornecer uma estratégia interessante para integrar um rastreador ocular em uma ferramenta IDE (Integrated Development Environment) para fornecer feedback ao desenvolvedor sobre como obter mais produtividade. Os

participantes da pesquisa também poderão ser beneficiados com uma experiência prática na resolução de problemas no âmbito da computação. Eles serão apresentados aos meios tecnológicos utilizados, bem como conceitos de compilação condicional, o que pode contribuir para sua prática de escrita e evolução de código-fonte no seu cotidiano.

De acordo com a RESOLUÇÃO Nº 466, DE 12 DE DEZEMBRO DE 2012, item V – DOS RISCOS E BENEFÍCIOS; Toda pesquisa com seres humanos envolve risco em tipos e gradações variados. Quanto maiores e mais evidentes os riscos, maiores devem ser os cuidados para minimizá-los e a proteção oferecida pelo Sistema CEP/CONEP aos participantes. Devem ser analisadas possibilidades de danos imediatos ou posteriores, no plano individual ou coletivo. A análise de risco é componente imprescindível à análise ética, dela decorrendo o plano de monitoramento que deve ser oferecido pelo Sistema CEP/CONEP em cada caso específico.

**Comentários e Considerações sobre a Pesquisa:**

A pesquisa realizará um estudo com programadores que serão expostos a certas atividades e observados através da câmera, mais especificamente os seus movimentos oculares serão observados. Com este estudo os pesquisadores pretendem encontrar evidências para afirmar se anotações disciplinadas promovem uma maior produtividade, acurácia, e compreensão de código em evolução de software quando comparada com anotações não disciplinadas. Trata-se de pesquisa relevante para a sociedade mas que precisa satisfazer todas as exigências dos CEPs acerca da documentação a ser apresentada.

**Considerações sobre os Termos de apresentação obrigatória:**

O pesquisador apresentou todos os documentos de apresentação obrigatória:

1- Folha de Rosto;

2- Declaração de divulgação dos resultados;

3- Termo de compromisso dos pesquisadores;

4- Termo de Consentimento Livre e Esclarecido – TCLE;

5- Termos de anuência das instituições;

6- Instrumentos de coleta;

7- Projeto completo;

8- Informações Básicas do Projeto de Pesquisa.

**Conclusões ou Pendências e Lista de Inadequações:**

Todos os documentos obrigatórios foram apresentados e estão de acordo com o esperado por

este CEP. Por esta razão sou de parecer favorável à realização desta pesquisa, salvo melhor juízo deste comitê.

**Considerações Finais a critério do CEP:**

**Este parecer foi elaborado baseado nos documentos abaixo relacionados:**

| Tipo Documento | Arquivo | Postagem | Autor | Situação |
|---|---|---|---|---|
| Informações Básicas do Projeto | PB_INFORMAÇÕES_BÁSICAS_DO_PROJETO_1431425.pdf | 10/10/2019 16:36:28 | | Aceito |
| TCLE / Termos de Assentimento / Justificativa de Ausência | TCLE_Termo.pdf | 10/10/2019 16:35:47 | José Aldo Silva da Costa | Aceito |
| Outros | TERMO_DE_ANUENCIA_INSTITUCIONAL_COPARTICIPANTE_UEPB.pdf | 25/09/2019 10:01:29 | José Aldo Silva da Costa | Aceito |
| Outros | DECLARACAO_DE_CONCORDANCIA_COM_PROJETO_DE_PESQUISA.pdf | 25/09/2019 09:57:40 | José Aldo Silva da Costa | Aceito |
| Projeto Detalhado / Brochura Investigador | PROJETO_DETALHADO.pdf | 25/09/2019 09:54:21 | José Aldo Silva da Costa | Aceito |
| Outros | TERMO_DE_ANUENCIA_INSTITUCIONAL_COPARTICIPANTE_UFAL.pdf | 25/09/2019 09:48:40 | José Aldo Silva da Costa | Aceito |
| Outros | TERMO_DE_ANUENCIA_INSTITUCIONAL.pdf | 25/09/2019 09:38:29 | José Aldo Silva da Costa | Aceito |
| Outros | DECLARACAO_DE_PESQUISA_NAO_INICIADA.pdf | 16/09/2019 21:30:20 | José Aldo Silva da Costa | Aceito |
| Outros | DECLARACAO_DE_DIVULGACAO.pdf | 16/09/2019 21:29:51 | José Aldo Silva da Costa | Aceito |
| Outros | DECLARACAO_DE_ANEXO_DOS_RESULTADOS.pdf | 16/09/2019 21:28:05 | José Aldo Silva da Costa | Aceito |
| Declaração de Pesquisadores | TERMO_DE_COMPROMISSO_DO_ORIENTADOR.pdf | 16/09/2019 21:16:48 | José Aldo Silva da Costa | Aceito |
| Declaração de Pesquisadores | DECLARACAO_DO_PESQUISADOR_RESPONSAVEL.pdf | 16/09/2019 21:15:39 | José Aldo Silva da Costa | Aceito |
| Orçamento | ORCAMENTO.pdf | 16/09/2019 21:12:32 | José Aldo Silva da Costa | Aceito |
| Cronograma | CRONOGRAMA.pdf | 16/09/2019 21:11:16 | José Aldo Silva da Costa | Aceito |
| Folha de Rosto | folhaDeRosto.pdf | 16/09/2019 20:50:47 | José Aldo Silva da Costa | Aceito |

**Situação do Parecer:**
Aprovado

**Necessita Apreciação da CONEP:**
Não

CAMPINA GRANDE, 18 de Dezembro de 2019

**Assinado por:**
**Andréia Oliveira Barros Sousa**
**(Coordenador(a))**

**Endereço:** Rua: Dr. Carlos Chagas, s/ n
**Bairro:** São José **CEP:** 58.107-670
**UF:** PB **Município:** CAMPINA GRANDE
**Telefone:** (83)2101-5545 **Fax:** (83)2101-5523 **E-mail:** cep@huac.ufcg.edu.br

## PARECER CONSUBSTANCIADO DO CEP

**Elaborado pela Instituição Coparticipante**

**DADOS DO PROJETO DE PESQUISA**

**Título da Pesquisa:** Uma Avaliação sobre Refatoramentos para #ifdefs com Rastreamento Ocular
**Pesquisador:** José Aldo Silva da Costa
**Área Temática:**
**Versão:** 1
**CAAE:** 24042719.8.3002.5187
**Instituição Proponente:** Universidade Estadual da Paraíba - UEPB
**Patrocinador Principal:** Financiamento Próprio

**DADOS DO PARECER**

**Número do Parecer:** 3.784.876

**Apresentação do Projeto:**
Trata-se de um estudo com a finalidade de elaboração de Tese de Doutorado em Ciências da Computação da Universidade Federal de Campina Grande - PB.
Lê-se:
Contexto
Durante a evolução de um software, os desenvolvedores executam várias alterações de código para lidar com mudanças nos requisitos e melhorar a manutenção do software. Essas alterações de código são frequentemente associadas à introdução de smells [1] que degradam a estrutura e a qualidade do código [2] [3], ou ainda à introdução de fraquezas que podem tornar um determinado software vulnerável [21]. Uma técnica comum usada para melhorar a qualidade do código consiste no refatoramento, que pode ser definida como uma transformação para melhorar a estrutura do código, preservando seu comportamento observável [2] [4]. As pesquisas apontam para várias métricas disponíveis que são usadas para avaliar o quanto a estrutura e a qualidade do código podem ser afetadas pela aplicação de refatoramentos [5] [6]. Por exemplo, algumas dessas métricas são linhas de código (LOC) [7], acoplamento entre objetos (CBO), relacionadas ao número de casamentos com outras classes e falta de
coesão nos métodos (LCOM), relacionadas à coesão de uma classe [8].
Problema
No entanto, para refatoramentos de código de granularidade fina, estudos anteriores não

identificaram muitas diferenças usando métricas como as mencionadas previamente [9] [10]. Por exemplo, de acordo com um estudo anterior, a maioria dos refatoramentos estudados são neutros, no sentido de que o número absoluto de smells permanece o mesmo após a aplicação do refatoramento [10]. No contexto da variabilidade com #ifdefs, encontramos um cenário semelhante no sentido de que não há consenso se anotações indisciplinadas devem ser refatoradas para se tornarem disciplinadas [11] [12]. Malaquias et al. [11] realizaram um experimento comparando anotações indisciplinadas e suas versões refatoradas para torná-las disciplinadas. No estudo,

eles descobriram que anotações indisciplinadas tomam mais tempo e são mais propensas a erros. Em outro experimento controlado, Schulze et al. [12] analisaram o efeito de anotações disciplinadas e não disciplinadas na compreensão do programa. No entanto, eles não encontraram diferenças entre anotações disciplinadas e indisciplinadas em relação ao tempo e à precisão. No que diz respeito às fraquezas introduzidas no código, Muniz et al. [24] identificaram que ocorrem mais fraquezas dentro de anotações #ifdefs do que fora dela, porém o estudou não investigou como o tipo de anotação pode afetar o tempo ou precisão com que desenvolvedores as encontram.

Solução

Neste sentido, objetivamos realizar uma análise quanti-qualitativa avaliando refatoramentos de código com granularidade fina em relação à sua capacidade de melhorar a qualidade e compreensão do código. Será avaliado refatoramentos para resolver anotações indisciplinados, por exemplo, anotações que abrangem um colchete de abertura, mas não abrangem aquele que o fecha. Ainda no contexto de variabilidade, também objetivamos investigar avaliar a dificuldade dos desenvolvedores de encontrarem fraquezas no código.

Avaliação

Deste modo, será realizado um experimento controlado usando rastreamento ocular com

sujeitos humanos, com o objetivo de observar como o código disciplinado afeta o tempo, a precisão e o número de varreduras nos refatoramentos de códigos de granularidade fina. Além disso, mediremos também o tempo gasto em regiões de código modificadas, as quais denominamos Área de Interesse (AOI), bem como o número de vezes que as pessoas entram nessas regiões. Assim, objetivamos usar evidências estatísticas para apoiar nossas suposições e validar nossas hipóteses.

**Objetivo da Pesquisa:**

Lê-se:

Esta pesquisa tem como objetivo principal analisar programas em linguagem C com #ifdefs usando rastreamento ocular com o objetivo de avaliar a qualidade de refatoramentos, dificuldade de se encontrar defeitos em anotações indisciplinadas ou disciplinadas, e dificuldade de se encontrar fraquezas no código. Para tal, medimos tempo, quantidade de varreduras, e número de tentativas na execução de tarefas de evolução de software do ponto de vista dos desenvolvedores no contexto da compreensão do código.

**Avaliação dos Riscos e Benefícios:**

Os riscos previstos aos participantes da pesquisa estão claramente definidos e a forma como minimizá-los.

Quanto aos benefícios previstos para a pesquisa serão:

Este trabalho beneficia diretamente a pesquisadores na academia, bem como a desenvolvedores que trabalham em empresas de programação ou startups. Para os pesquisadores, uma metodologia que utiliza métricas baseadas no olhar podem revelar nuances e fornecer insights distintos de outras métricas para transformações refinadas. Por exemplo, para transformações de granularidade fina, é difícil avaliar se a redução de uma linha de código extra ou a adição de uma variável extra realmente melhora a qualidade do código. No entanto, com métricas baseadas no olhar, observamos se os desenvolvedores fazem menos varreduras no código, quais trechos exigem mais atenção, ou quais construções são mais complexas. Essa metodologia tem o potencial de aprofundar a discussão sobre as vantagens de anotações disciplinares e adicionar uma nova perspectiva de análise além do tempo e da precisão utilizada em estudos anteriores [12] [13]. Vale a pena salientar que os resultados deste trabalho também podem contribuir para confirmar ou descartar a necessidade da proposição de

novos refatoramentos para lidar com o uso indisciplinado de #ifdefs, bem como da avaliação de cada um deles na prática para entender qual versão (disciplinada vs. indisciplinada) é melhor Essa pesquisa também tem potencial de beneficiar desenvolvedores para que sejam mais produtivos, e dessa forma, beneficiar empresas para que gastem menos recursos financeiros em atividades de evolução de código. Através da análise de métricas como varreduras no código, varreduras em áreas específicas do código, frequência de entradas nessas áreas, tempo para primeira varredura, temos meios para inferir padrões de leitura de código, o

que pode nos fornecer informações úteis sobre como os desenvolvedores compreendem o código. Esses insights podem fornecer uma estratégia interessante para integrar um rastreador ocular em uma ferramenta IDE (Integrated Development Environment) para fornecer feedback ao desenvolvedor sobre como obter mais produtividade.

Os participantes da pesquisa também poderão ser beneficiados com uma experiência prática na resolução de problemas no âmbito da computação. Eles serão apresentados aos meios tecnológicos utilizados, bem como conceitos de compilação condicional, o que pode contribuir para sua prática de escrita e evolução de código-fonte no seu cotidiano.

**Comentários e Considerações sobre a Pesquisa:**

A pesquisa está bem fundamentada, com metodologia claramente definida e atende à Resolução 466/2012 e suas complementares do CONEP/ 2012.

**Considerações sobre os Termos de apresentação obrigatória:**

Todos os termos encontram-se devidamente anexados.

**Recomendações:**

Recomenda-se Envio do Relatório quando da realização do estudo.

**Conclusões ou Pendências e Lista de Inadequações:**

Somos de parecer favorável à realização do estudo.

**Considerações Finais a critério do CEP:**

**Este parecer foi elaborado baseado nos documentos abaixo relacionados:**

| Tipo Documento | Arquivo | Postagem | Autor | Situação |
|---|---|---|---|---|
| TCLE / Termos de Assentimento / Justificativa de Ausência | TCLE_Termo.pdf | 10/10/2019 16:35:47 | José Aldo Silva da Costa | Aceito |
| Outros | TERMO_DE_ANUENCIA_INSTITUCIONAL_COPARTICIPANTE_UEPB.pdf | 25/09/2019 10:01:29 | José Aldo Silva da Costa | Aceito |
| Outros | DECLARACAO_DE_CONCORDANCIA_COM_PROJETO_DE_PESQUISA.pdf | 25/09/2019 09:57:40 | José Aldo Silva da Costa | Aceito |
| Projeto Detalhado / Brochura Investigador | PROJETO_DETALHADO.pdf | 25/09/2019 09:54:21 | José Aldo Silva da Costa | Aceito |
| Outros | TERMO_DE_ANUENCIA_INSTITUCIONAL_COPARTICIPANTE_UFAL.pdf | 25/09/2019 09:48:40 | José Aldo Silva da Costa | Aceito |

| | | | | |
|---|---|---|---|---|
| Outros | TERMO_DE_ANUENCIA_INSTITUCIONAL.pdf | 25/09/2019 09:38:29 | José Aldo Silva da Costa | Aceito |
| Outros | DECLARACAO_DE_PESQUISA_NAO_INICIADA.pdf | 16/09/2019 21:30:20 | José Aldo Silva da Costa | Aceito |
| Outros | DECLARACAO_DE_DIVULGACAO.pdf | 16/09/2019 21:29:51 | José Aldo Silva da Costa | Aceito |
| Outros | DECLARACAO_DE_ANEXO_DOS_RESULTADOS.pdf | 16/09/2019 21:28:05 | José Aldo Silva da Costa | Aceito |

**Situação do Parecer:**
Aprovado

**Necessita Apreciação da CONEP:**
Não

CAMPINA GRANDE, 19 de Dezembro de 2019

**Assinado por:**
**Dóris Nóbrega de Andrade Laurentino**
**(Coordenador(a))**

**Endereço:** Av. das Baraúnas, 351- Campus Universitário
**Bairro:** Bodocongó **CEP:** 58.109-753
**UF:** PB **Município:** CAMPINA GRANDE
**Telefone:** (83)3315-3373 **Fax:** (83)3315-3373 **E-mail:** cep@uepb.edu.br