Federal University of Alagoas

Computing Institute

Graduate Program in Informatics

Atoms of Confusion Do Really Cause Confusion? A Controlled Experiment Using Eye Tracking

Master Student
Benedito Fernando Albuquerque de Oliveira

Advisor
Márcio de Medeiros Ribeiro

Maceió, AL
October - 2020

Benedito Fernando Albuquerque de Oliveira

Atoms of Confusion Do Really Cause Confusion? A Controlled Experiment Using Eye Tracking

Defense of dissertation for the title of Master in Informatics by the Computing Institute of the Federal University of Alagoas.

Federal University of Alagoas – UFAL

Computing Institute

Graduate Program in Informatics

Supervisor: Márcio de Medeiros Ribeiro

Maceió

2020

Catalogação na fonte
Universidade Federal de Alagoas
Biblioteca Central
Divisão de Tratamento Técnico
Bibliotecária: Taciana Sousa dos Santos – CRB-4 – 2062

O48a Oliveira, Benedito Fernando Albuquerque de.
Atoms of confusion do really cause confusion? A controlled experiment using eye tracking / Benedito Fernando Albuquerque de Oliveira. – 2020.
57 f. : il., figs. e tabs. color.

Orientador: Márcio de Medeiros Ribeiro.
Dissertação (Mestrado em Informática) – Universidade Federal de Alagoas. Instituto de Computação. Maceió, 2021.

Bibliografia: f. 55-57.

1. Compreensão de código. 2. Átomos de confusão (Códigos). 3. Rastreamento visual. I. Título.

CDU: 004.4’ 4

UNIVERSIDADE FEDERAL DE ALAGOAS/UFAL
Programa de Pós-Graduação em Informática – PPGI
Instituto de Computação/UFAL
Campus A. C. Simões BR 104-Norte Km 14 BL 12 Tabuleiro do Martins
Maceió/AL - Brasil CEP: 57.072-970 | Telefone: (082) 3214-1401

Folha de Aprovação

BENEDITO FERNANDO ALBUQUERQUE DE OLIVEIRA

ATOMS OF CONFUSION DO REALLY CAUSE CONFUSION? A CONTROLLED EXPERIMENT USING EYE TRACKING

Dissertação submetida ao corpo docente do Programa de Pós-Graduação em Informática da Universidade Federal de Alagoas e aprovada em 26 de NOVEMBRO de 2020.

Banca Examinadora:

Prof. Dr. MARCIO DE MEDEIROS RIBEIRO
UFAL – Instituto de Computação
Orientador

Prof. Dr. ALAN PEDRO DA SILVA
UFAL – Instituto de Computação
Examinador Interno

Prof. Dr. LEOPOLDO MOTTA TEIXEIRA
UFPE – Universidade Federal de Pernambuco
Examinador Externo

I dedicate this work to everyone
who helped me along this journey.

Acknowledgements

I thank my family for all the support during this journey. Thank you for having supported me in times of difficulty and encouraged me to face this challenge.

I am grateful to my wife, Sarah Oliveira. She always supported me at all times, especially on the countless times that I thought about giving up and felt unable to overcome the obstacles that appeared along the way. Thank you for never have given up on me.

I thank my advisor Márcio Ribeiro, for his dedication, advice, guidance, and patience. This patience has been put to the test too many times. Although he had reasons for that, he never gave up. He remained available and active so that I could complete this stage. Without him, this work would not be possible. Thank you.

I am grateful to the friend and research colleague José Aldo. He helped decisively in several moments, sharing knowledge and techniques necessary for the improvement of the work.

I thank my friend Guilherme Volney for the support and great help given during the experiment's construction.

I thank Professor Alessandro Garcia for the guidance, support, and welcome at his institution during the experiment's execution.

I would also like to thank all the other teachers who somehow added something to my research, such as Baldoino Fonseca, Rohit Gheyi, and Thiago Cordeiro.

I would also like to thank the EASY lab friends who have always supported me on this journey. Thanks for encouraging me not to give up and contribute to improving this work, including Fernando Kenji, Leonardo Fernandes, Pedro Matheus, Marcio Augusto, Elvys, Francisco Dalton. Rodrigo Lima, among others.

I am also grateful to my job's colleagues for their comprehension of the moments I have been absent to work on this research.

Finally, I would like to thank all the people who directly and indirectly contributed to accomplishing this work; I apologize to those that the names are not here.

Thank you to professors, employees, and students at UFAL and PUC-RIO.

"When everything seems to be going against you,
remember that the airplane takes off against the wind,
not with it" – Henry Ford

Resumo

Compreensão de código é crucial nas atividades de manutenção de software, entretanto ela pode ser prejudicada por mal-entendidos e padrões código confusos, ou seja, átomos de confusão. Eles são pequenos trechos de código usando construções específicas de uma linguagem de programação, como *Operadores Condicionais* e *Operadores Vírgula*. Um estudo anterior mostrou que os átomos de confusão afetam o desempenho dos desenvolvedores, ou seja, o tempo e a precisão, e aumentam os mal-entendidos com relação ao código. No entanto, o conhecimento empírico do impacto de tais átomos na compreensão do código ainda é escasso, especialmente quando se trata de analisar esse impacto na atenção visual dos desenvolvedores. O presente estudo avalia se os desenvolvedores interpretam mal o código na presença de átomos de confusão com um rastreador ocular. Para isso, medimos o tempo, a precisão e analisamos a distribuição da atenção visual. Conduzimos um experimento controlado com 30 alunos e profissionais de software. Pedimos aos sujeitos que especifiquem a saída de três tarefas com átomos e três sem átomos designados aleatoriamente usando um Quadrado Latino. Usamos uma câmera de rastreamento ocular para detectar a atenção visual dos participantes enquanto resolvemos as tarefas. De uma perspectiva agregada, observamos um aumento de 43,02% no tempo e 36,8% nas transições de olhar em trechos de código com átomos. Além disso, observamos um aumento de 163,06% no número de regressões quando o átomo está presente. Para precisão, nenhuma diferença estatisticamente significativa foi observada. Também confirmamos que as regiões que recebem mais atenção foram as regiões com átomos. Nossas descobertas reforçam que os átomos atrapalham o desempenho e a compreensão dos desenvolvedores. Portanto, os desenvolvedores devem evitar escrever código com eles.

Palavras-Chave: Átomos de Confusão, Rastreamento Visual.

Abstract

Code comprehension is crucial in software maintenance activities, though it can be hindered by misunderstandings and confusion patterns, namely, atoms of confusion. They are small pieces of code using specific programming language constructs, such as *Conditional Operators* and *Comma Operators*. A previous study showed that these atoms of confusion impact developers' performance, i.e., time and accuracy, and increase code misunderstandings. However, empirical knowledge of the impact of such atoms on code comprehension is still scarce, especially when it comes to analyzing that impact on developers' visual attention. The present study evaluates whether developers misunderstand the code in the presence of atoms of confusion with an eye tracker. For this purpose, we measure time, accuracy, and analyze the distribution of visual attention. We conduct a controlled experiment with 30 students and software practitioners. We ask the subjects to specify the output of three tasks with atoms and three without atoms randomly assigned using a Latin Square design. We use an eye-tracking camera to detect the visual attention of the participants while solving the tasks. From an aggregated perspective, we observed an increase by 43.02% in time and 36.8% in gaze transitions in code snippets with atoms. Also, we observed an increase of 163.06% in the number of regressions when the atom is present. For accuracy, no statistically significant difference was observed. We also confirm that the regions that receive most of the eye attention were the regions with atoms. Our findings reinforce that atoms hinder developers' performance and comprehension. So, developers should avoid writing code with them.

Keywords: atoms of confusion, eye-tracking.

List of Figures

List of Tables

Contents

1 Introduction

Code comprehension is a critical activity in software development, especially in the maintenance and evolution processes. Developers often have to deal with maintaining or improving code that they did not write. To perform any modification in the code, first, developers have to understand it. Indeed, a previous study showed that most of their time is spent on code comprehension activities [1]. However, the comprehension process can be hindered by aspects of the code that cause misunderstandings. For instance, previous works [2–4] have successfully identified a set of tiny code patterns that contribute to the increase of the time and effort necessary to understand the code correctly. Examples of these small code patterns in imperative languages like C and C++, namely *atoms of confusion*, include *Conditional Operators*, *Comma Operators*, *Logic as Control Flow*, among others, and are often found in source code bases [5].

In a previous study [4], researchers conducted a controlled experiment to compare the performance, i.e., time and accuracy, of participants when dealing with code with and without *atoms of confusion*. They have shown that the presence of atoms of confusion makes the code understanding more time-consuming and that, by removing them, participants could understand the code with less effort. However, empirical knowledge on the impact of such atoms on code comprehension is still scarce, given the difficulties in measuring code comprehension [6]. On the other hand, the code comprehension field has been gaining new insights from other dimensions besides time and accuracy with eye-tracking devices [6]. For instance, knowing which parts of the code receive more or less attention would give us insights on the extent of the impact of such atoms on visual effort, which is a dimension not considered before [4] or in any other study we are aware of. Although time and accuracy are good estimators of code confusion, knowing precisely the regions that participants paid attention allows us to have an even better estimation of their real effects. This way, by adding this dimension, we can now triangulate it with the conventional measures time and accuracy.

Given this scenario, we evaluate whether developers misunderstand the code in the presence of atoms of confusion with an eye tracker. For this purpose, we measure time, accuracy, and distinctly from other studies, we also analyze the visual effort given to specific regions in the code. The triangulation of these dimensions allows us to understand better the real effects of atoms of confusion in code comprehension and whether these code snippets can make a difference in software maintenance tasks. Our research is inspired by previous works that contact actual developers using source code repositories (e.g., GitHub) [7, 8] and by the aforementioned experiment [9]. Understanding the advantages and disadvantages of removing atoms of confusion from the code is essential to guide the

development of new tools and improve programming practices on this front.

In this controlled experiment, we selected six functions from real open-source C/C++ systems of different domains containing atoms of three different types (*Assignment as Value*, *Conditional Operator*, and *Logic as Control Flow*), being two functions for each type of atom. We selected these atoms types because they are commonly found in industrial practice [5]. We then manually refactored the code to remove the atoms from the functions and to build our tasks. We also changed the function to remove several internal code dependencies so that the function can get more straightforward, but we keep its primary structure. We executed the experiment with 30 developers, including master and doctoral students, and also with programmers already working in industry. The developers were asked to read, understand, and specify the six tasks' output, three of them with atoms and three with no atoms. We used the Latin Square design to assign the tasks randomly and minimize learning effects. We then measure the completion time of tasks, as well as the number of incorrect answers. Besides, through the eye tracker, we captured the developers' eyes coordinates while looking at the screen. This allowed us to build heatmaps, measure the number of times that participants enter in a certain area of the code, measure the number of gaze transitions, which are the transitions of the eyes from one location of the code to another, and measure the number of regressions, which are the number of times that the participant's visual attention moves to previous code areas. To analyze the regressions, we separated the code into three different areas, upper the area of interest, area of interest, and lower area of interest; we have a regression any time the visual attention gets back to an upper area in the code.

When analyzing all tasks together, we found that the presence of atoms of confusion statistically significantly increased the time required to understand the code by 43.02%. We could not find a statistically significant difference for accuracy. When analyzing the tasks individually, two of the three analyzed atoms allowed us to observe a statistically significant increase in time. Regarding accuracy, none of them showed statistically significant differences. The heatmap analysis allowed us to confirm that the regions that receive most of the attention are precisely the regions where the atoms are placed. The regions with atoms have an increase of 36.8% gaze transitions. We also found that the regression to the area of interest has an increase of 163.06% when analyzing functions that contain atoms. These results confirm and add a new perspective to the previous study [4].

In conclusion, our findings reinforce that atoms of confusion cause confusion by hindering developers' performance and code comprehension. They make the maintenance activity more time-consuming and require more visual effort. We also show that the eye-tracking methodology seems to be promising, revealing a new perspective not seen in previous works.

In summary, this study provides the following contribution: *An empirical controlled*

experiment to evaluate whether developers misunderstand the code in the presence of atoms of confusion with the use of an eye tracker.

We organized this study as follows: Chapter 2 presents the background, covering topics such as code comprehension and atoms of confusion. In this same chapter, we present some motivating examples presenting snippets of code with atoms of confusion that occurred in real projects. Chapter 4 presents the controlled experiment along with results, discussion, threats to validity, and implications. Chapter 5 presents the related work. Finally, Chapter 6 presents the conclusion and future works.

2 Background and Motivating Example

2.1 Code Comprehension

Code comprehension is one of the most critical tasks in software maintenance, mainly due to the high costs associated with this task. In this work, we aim to bring another perspective to analyze code comprehension. We hypothesize that the presence of atoms of confusion has a negative impact on code comprehension, and to measure this, we introduce the use of an eye-tracking camera. Program understanding is crucial in software maintenance and evolution of [10] processes. As Rajlich mentions in [11]: "software that is not comprehended cannot be changed." Software comprehension is also essential in other areas like documentation, visualization, program design, and other areas. In [12], Lakhotia mentions an interesting thing about the code comprehension process. Software developers do not need to know the entire program to work on a maintenance or improvement task. They only need to understand the parts that make it possible for him to complete his task. In this work, we provide a study on code comprehension regarding the named atoms of confusion, which could be some of the "parts" mentioned by Lakhotia.

2.2 Atoms of Confusion

Gopstein [4] defines *atoms of confusion* as the smallest patterns in code that can cause misunderstanding in programmers. He also defines the term confusion when a person and a machine read the same piece of code and they come to different conclusions about its output. Medeiros [5] named these patterns as *Misunderstanding Patterns* and show that the majority of the patterns that they have analyzed are used in practice by developers in open-source projects.

Table 2.2 shows 15 atoms of confusion found by Gopstein [3]. He mentions that there is a significant increase in misunderstandings when they are present in code:

In another study [13], Gopstein performed an experiment and a qualitative analysis to study why and how these atoms cause confusion, in addition to understanding whether atoms cause confusion. Although atoms of confusion were first studied and cataloged using the C programming language, there are some other studies focused on analyzing the atoms of confusion in different programming languages.

Atom Name	**Atom Example**	**Transformed**
Change of Literal Encoding	printf("%d",013) Encoding	printf("%d",11)
Preprocessor in Statement	int V1 = 1 #define M1 1 + 1;	#define M1 1 int int V1 = 1 + 1;
Macro Operator Precedence	#define M1 64-1 2*M1	2*64-1
Assignment as Value	V1 = V2 = 3;	V2 = 3; V1 V1 = V2;
Logic as Control Flow	V1 && F2();	if (V1) F2();
Post-Increment/Decrement	V1 = V2++;	V1 = V2; V2 V2 += 1;
Type Conversion	(double)(3/2)	trunc(3.0/2.0)
Reversed Subscripts	1["abc"]	"abc"[1]
Conditional Operator	V2 = (V1==3)?2:V2	if (V1 == 3) V2 = 2;
Operator Precedence	0 && 1 || 2	(0 && 1) || 2
Comma Operator	V3 = (V1 += 1, V1)	V1 += 1;
Pre-Increment/Decrement	V1 = ++V2;	V2 += 1; V1 = V2;
Implicit Predicate	if (4 % 2)	if (4 % 2 != 0)
Repurposed Variables	argc = 7;	int V1 = 7;
Omitted Curly Braces	if(V) F(); G();	if(V){F();}G(); V1

Table 1 – Examples of atoms of confusion.

2.3 Eye Tracking

Eye-tracking cameras are capable of tracking the exact position on the screen that the user is looking at. Some studies have used such cameras in researches on different domains, including program comprehension [14] [15] [16] [17].

Carpenter [18] studied reading comprehension tasks using an eye-tracking camera and have drawn some interesting conclusions. For instance, he has found that "readers make longer pauses at points where processing loads are greater," and that "greater loads occur while readers are accessing infrequent words, integrating information from important clauses, and making inferences at the ends of sentences." With this conclusion, he showed that indeed there is a relation between text comprehension and visual attention. Regarding code comprehension, Busjahn [19] conducted a study, also using an eye-tracking camera, with 15 programmers in order to understand what influences the dwell time on source code reading. He showed that developers focus most of their attention on understanding identifiers, operators, keywords, and literals. Another critical point identified [19] is that the fixation duration is highly influenced by word length, predictability, and frequency. For instance, the less frequent a word is in the text, the longer the readers stay focused.

2.4 Motivating Examples

Atoms of confusion frequently appear in open source repositories. A previous study on their prevalence on 50 projects found more than 109 thousand occurrences of 11 out of 12 atoms types considered, including the ones we focus on this study: *Assignment as Value*, *Conditional Operator*, and *Logic as Control Flow* [5].

Previous studies indicate that Atoms of Confusion can lead to code misunderstandings. These misunderstandings, in turn, may lead to the introduction of bugs. In particular, these problems may get worse when performing maintenance tasks on code [5].

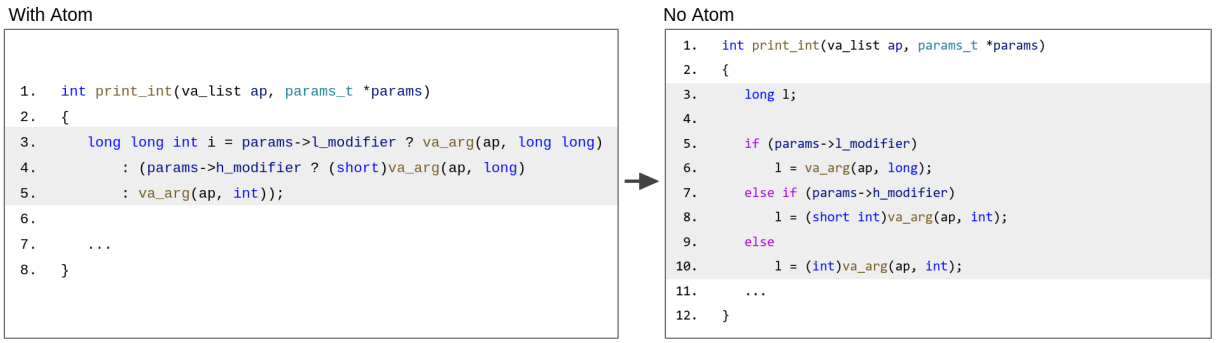


Figure 1 – A transformation removing an atom of confusion (Conditional Operator).

Figure 1 illustrates two code snippets from functions `printf` written in C programming language. The left-hand side snippet contains a function with an example of the atom of confusion called *Conditional Operator*.[1] Notice the two nested ternary operators. At the right-hand side, a developer refactored the function to remove the *Conditional Operator* atom by replacing it with a regular `if-else` statement.[2]

According to Gopstein et al. [3], there is empirical evidence that atoms of confusion such as this one impact developers' performance, causing misunderstanding, and that their removal makes code easier to understand. However, to have an even more realistic understanding of the impact of such atoms on code comprehension, we need to assess other dimensions besides time and accuracy to draw more insights. In this scenario, eye tracking devices come at hand to allow us to locate where and for how long subjects are looking at the screen, in other words, which elements on the code drive their attention [17]. These devices have been used before in the code comprehension field (e.g., conditional compilation with `#ifdefs`) along with time and accuracy to provide additional information related to visual effort [16, 20].

[1] https://github.com/narnat/printf/commit/6b4f1fe6ef454f1a2c820e88582844151dcb8e6c
[2] https://github.com/narnat/printf/commit/385233be85dd400fc17be255b529e5b0cf14e29b

Thus, to gather more information on how atoms impact code comprehension, we use an eye-tracking camera to analyze how those atoms influence attention distribution. By triangulating the time, accuracy, and focus of the developers, we can better understand the atoms' impact. For instance, it helps us to confirm whether the regions of the code that receive most of the attention are the places where the atoms are positioned, how the focus of attention changes interacting with code elements, and whether removing those patterns alleviates developers' effort from a visual perspective not considered in previous works.

3 Study: Controlled Experiment

We now present our controlled experiment. We present the settings, experimental units, and the discussion of results.

3.1 Study Settings

In this experiment, we analyze programs written in C with atoms of confusion and with no atoms of confusion using an eye-tracking camera to investigate the effects of the presence of the atoms concerning time, accuracy, and focus of attention in solving "specify correct output" tasks from the point of view of developers in the context of code comprehension.

We focus on the following research questions:

- RQ_1*: To what extent do atoms of confusion affect task completion time?* To answer this question, we measure the total time developers need to solve each task. Thus, our first null hypothesis ($H1_0$) is: there is no significant difference between time required to understand code with no atoms of confusion (the control treatment) and code with atoms of confusion (the treatment under investigation);
- RQ_2*: To what extent do atoms of confusion affect task accuracy?* To answer this question, we measure the number of errors committed by developers while solving the task. Thus, our second null hypothesis ($H2_0$) is: there is no significant difference in the number of errors committed by developers when understanding code with no atoms of confusion (the control treatment) and code with atoms of confusion (the treatment under investigation).
- RQ_3*: To what extent do atoms of confusion affect the focus of attention?* We measure the number of gaze points captured using an eye tracking system and generate heatmaps. The heatmaps of the aggregated tasks solved by the developers help us to visualize possible differences on attention. Our third null hypothesis ($H3_0$) is: there is no significant difference in the number of gaze points when analyzing code with no atoms of confusion (the control treatment) and code with atoms of confusion (the treatment under investigation). In addition, we measure the number of gaze points inside the area of the atom of confusion and compare it with the number of points in the region modified to remove the atom.

Since we are comparing two treatments (with atom vs. no atom), in case we reject the null hypothesis, we only have to compare the mean value of the control and treatment

observations to estimate the effect of the atoms of confusion on code comprehension tasks.

We use a latin square design of order two with replicas [21] in our experiment, mainly because our goal is to compare two treatments and block two variables: (a) participant skill and engagement; and (b) two sets of comprehension tasks. This design is widely used for experiments with these characteristics because it blocks the two sources of variability and each treatment appears in each Latin Square line. Using this design, we control or eliminate these two variability sources [21]. All participants will perform all treatments, each in a different and randomized order. In other words, this design helps us to avoid, for example, the effects of the different subjects' knowledge and background experience. Each Latin Square replica comprises two participants (randomly assigned to the rows of the squares) and two sets of comprehension tasks (representing the columns of each square, i.e., ST_1 and ST_2). Each set of tasks has three tasks, one containing a *Logic as Control Flow*, one containing an *Assignment as Value*, and one containing a *Conditional Operator*. Figure 2 presents the design of our experiment. These three kinds of atoms were chosen due to their big prevalence in open source repositories. All of them are part of the Gopstein [4] catalog and were also analyzed by Medeiros [5]. Still, according to Medeiros [5] all of these three atoms appear at least in more then 50% of the repositories analyzed by him. He found occurrences of *Conditional Operator* in 98% of the repositories, *Assignment as Value in 94%*, and *Logic as Control Flow* in 50%.

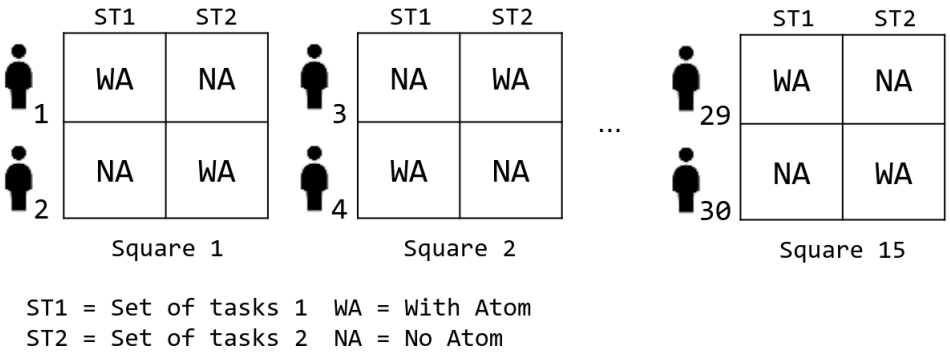


Figure 2 – Design of the experiment using Latin squares.

We also randomly set the treatments that each participant should use in ST_1 and ST_2. For example, the fourth participant of Figure 2 should analyze ST_1 functions with atoms of confusion (WA) and ST_2 functions with no atoms of confusion (NA) in this order.

This design uses randomization to assign the participants to the squares and assign the treatments to the cells of each square. Each treatment appears once in each row and column. This way, we block the two sources of variability: the participants and the two sets of tasks [22, 23]. The design also leads to one replica for each Latin Square (increasing the number of errors' degrees of freedom) [24].

3.2 Experimental Units

In total, 30 subjects participated in the experiment. Among them, masters and doctoral students at the Pontifical Catholic University of Rio de Janeiro and other software development practitioners working on projects at different sectors of the same university. Although we have used code structures common to most programming languages, one of the criteria for participating in this experiment was that the participant had prior knowledge in at least one of the following programming languages: C, C++, or Java, regardless of the level of expertise. In table 2 we show the number of participants by academic degree and their range of experience by years and number of projects.

Table 2 – Participants Characterization

Subjects (n = 30)		
Academic degree	**Count**	**Percentage**
Undergraduated	9	30
Master	17	56,6
Doctor	4	13,3
Experience	**Range**	**Median**
Years	1 - 40	6
Number of Projects	0 - 50	7,5

Initially, we recruited 33 subjects. These subjects were randomly organized in an initial set of 17 *Latin Squares*. However, we had to discard 3 of them. First, we discarded the last square because we could not run the experiment with one more subject to complete the last square. Then, we discarded another subject because he answered the cell phone while participating in the experiment. This way we also had to discard the other participant on his *Latin Square*.

For the experiment execution, we have prepared a laptop with a system that was developed to control the experiment's execution flow. The system is a web application written in *Python* using the *Django Web Framework*. All captured data was stored in a *PostgreSQL* database to posterior analysis. Each task was also stored on that database, and the system showed the functions to the participants automatically when they finish a task based on the previous Latin Squares randomization. Figure 3 illustrates our setup. It shows the two possible configurations depending on the Latin Squares randomization. The first configuration should be presented to the participants who (a) solve the first set of tasks (ST_1) comprising code with atoms of confusion; and (b) solve the second set of tasks (ST_2) comprising code with no atoms of confusion. Both sets contain three tasks: that is, ST_1 contains the tasks $T1$, $T2$, and $T3$, while ST_2 contains the tasks $T4$, $T5$, and $T6$. In the second configuration, we invert our treatments, according to the Latin square design (Figure 3).

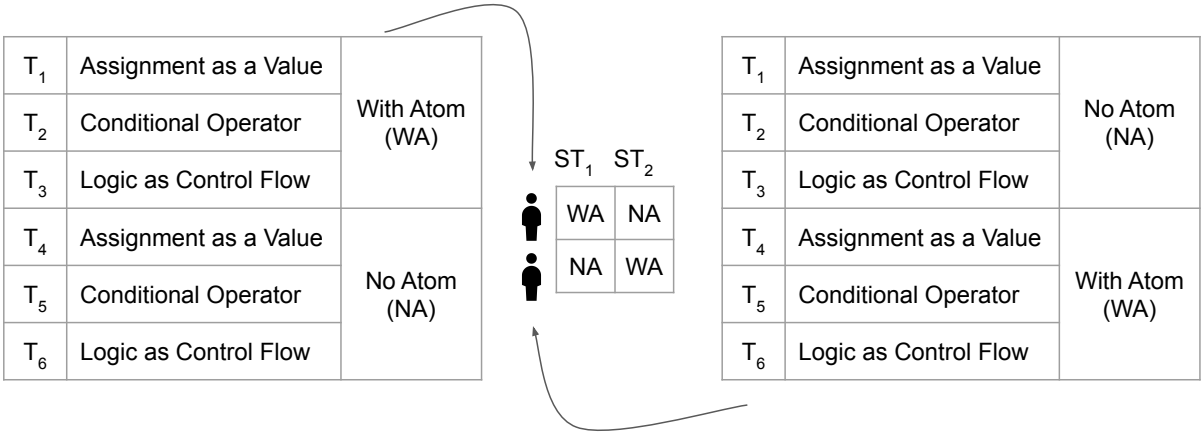


Figure 3 – Structure of the experiment in terms of experimental units. We have two Set of Tasks (ST_1 and ST_2) the comprehend Tasks 1 to 6 (T_1, T_2, .., T_6). Half of the tasks are with atoms (WA) and half with no atoms (NA).

To create the tasks for the experiment, we mined the code of open source projects to find functions containing atoms of confusion. The selected functions were adapted and simplified to make them self-contained and fit them to the scope of our research, as well as to reduce the time required for completion by participants. We have tasks with three different types of atoms (see Figure 4): *Assignment as Value*, *Conditional Operator*, and *Logic as Control Flow*. Each task has one or more `printf` statements; the assignment for each participant is to specify the correct output to be printed on the standard output when the function is executed. We have selected six functions in total, and for each selected function, we created a refactored version to remove the atom of confusion and use it as a control treatment group. Altogether, we have prepared 12 comprehension tasks: six with atoms of confusion and six with no atoms of confusion.

3.3 Execution and Data Analysis Procedures

We first randomly assigned the treatments (With Atom × with No Atom) to the 15 Latin square replicas. This assignment was done automatically by the previously mentioned system. We created a script to generate the Latin Squares and assign the treatments randomly. When a new participant starts the experiment execution, the system automatically assigns him to the next available row in the Latin Square's list. Before starting the tasks, for each participant, the eye-tracking device is re-calibrated in order to make it as accurate as possible. This procedure is required because the calibration varies according to both the physical characteristics of the participant and other external characteristics such as the use of glasses, positioning, and distance of the chair relative to the computer, participant's height, among others. We have plugged the device at the bottom of the laptop's screen

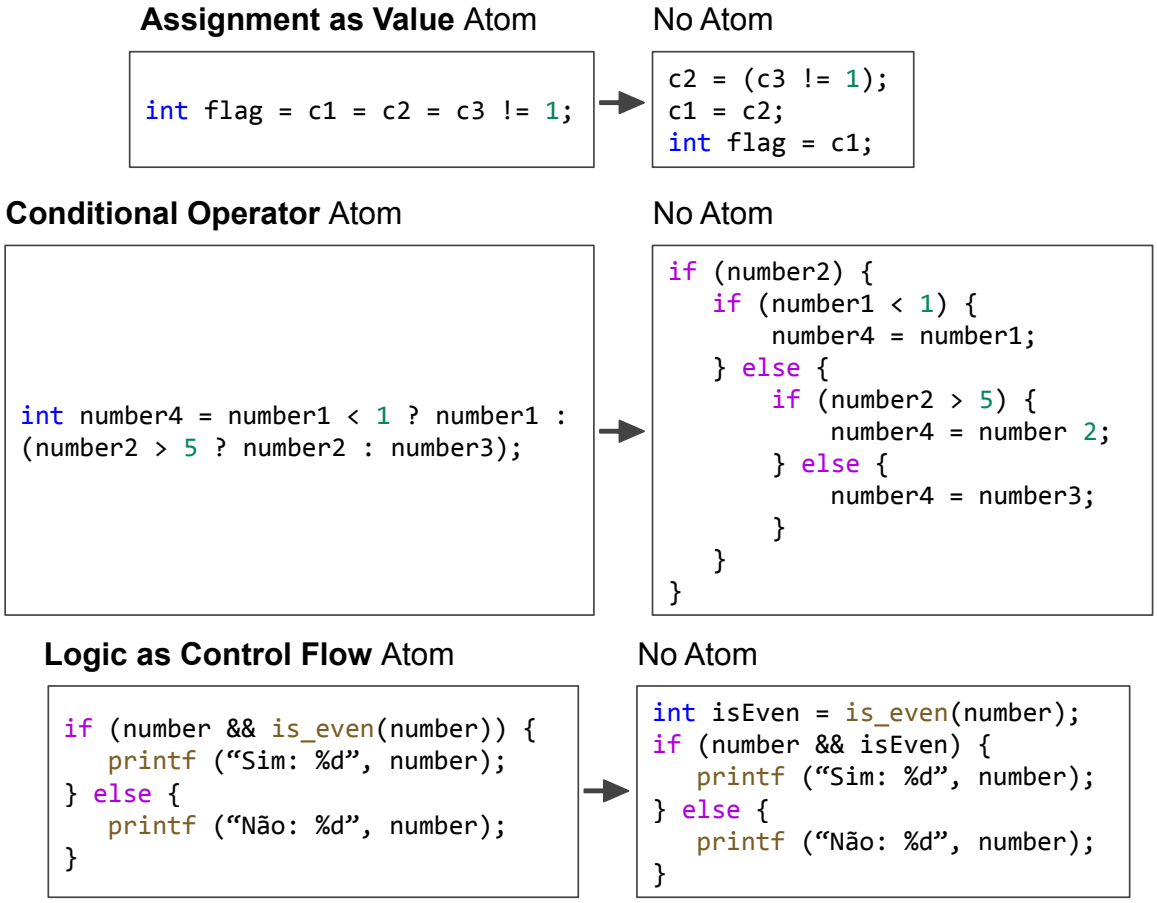


Figure 4 – Code snippets with the three types of atoms evaluated in this study. At the left-hand side, we have the atom of confusion (treatment group) and at the right-hand side we have the same code with no atoms (control group).

to capture the points where the participant is looking on the screen. Then, the captured eye gaze points were stored in a database. Before executing the two sets of tasks, each participant completed a Warm-Up with just two very simple and straightforward functions that did not contain any atom of confusion to understand how the flow of the experiment works and get acquainted with the eye tracker equipment setup.

Once the Warm-Up is successfully done, the system allows the participant to execute the main tasks. When a participant believes to have a correct answer, he or she should click anywhere on the screen, and the system will pause the experiment and show a screen that allows him or her to submit the answer.

If the participant's answer is wrong, the system increments the error count for this execution and allows the user to get back to the task or enter a new answer. Once the participant provides the correct answer, the system allows him or her to go to the next task. During the execution of the tasks, the participants are allowed to pause anytime by clicking anywhere on the screen. The system collects the start and end time for the

execution of each task as well as for the complete experiment. It also stores the number of wrong answers, the number of pauses and their duration, each provided answer, and the points collected by the eye tracker device. We considered the execution completed only when all tasks are correctly solved.

We proceeded with an exploratory data assessment and tested the hypotheses introduced in Section 3.1. Since our dataset does not follow a normal distribution, we used the Kruskal-Wallis test. Kruskal-Wallis is a nonparametric test that can be used to determine if there are statistically significant differences between two or more groups of an independent variable on a continuous or ordinal dependent variable, as we detail in the next section.

We analyzed the eye gaze perspective using a graphical representation, namely the heatmaps, and measured the number of gaze transitions, along with other metrics associated with how gaze points are located inside or move between regions of the code. Heatmaps are static graphical representations in colors, and the intensity of the color represents the concentration of gaze points over a particular area. We aggregated the heatmaps of the same task performed by 15 distinct participants. The number of gaze transitions adds a more dynamic perspective of analysis, expressing how many transitions of the focus of attention occurred in the code. The eye tracker camera captures about 80 sample points per second, and tasks took from a few seconds to a few minutes to be answered. To ease the analysis, we computed the median of the coordinates x and y of these 80 points in one second. This coordinate is the *focus of attention per second*. The transitions of the eyes from one focus to another is the gaze transition. We analyzed the chronological sequence of the focus points in the code lines of all participants in the same task. To simplify the analysis, we considered only transitions that occur at least 1/3 times. That is, at least 1/3 of the participants must have performed the same transition allowing us to observe a pattern.

3.3.1 Tool to collect the data

To perform this experiment, we developed a web application in *Python* using the *Django Framework*. This application controls the flow of the experiment and collects all the necessary data to extract our metrics. This tool collects the duration of each task and the duration of pauses done by the participant, the number of responses given and whether they were correct or incorrect, and the points on the screen where the participant looked during the execution of the experiment. The tool includes an administration area that allows us to set up all the experiments and tasks. Figure 5 and Figure 6 show the list of experiments and the details of an experiment respectively in this admin area.

It is also responsible for randomizing the *Latins Squares*. When a new experiment is created in the tool, a default quantity of *Latin Square* is informed, and the tool creates

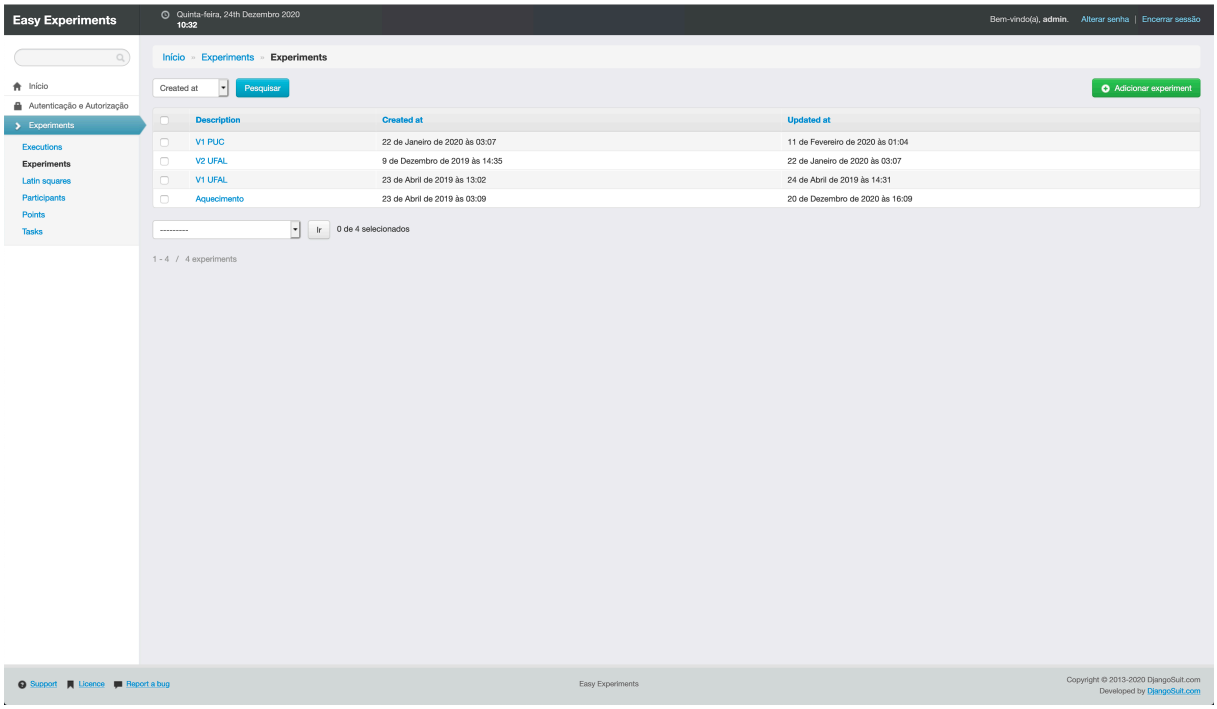


Figure 5 – Admin area - List of Experiments.

the structure for them. When a participant starts the experiment, the tool allocates him in a row in the next *Latin Square* with available space. If we perform the experiment with a number of participants higher than expected, the tool automatically creates a new randomized *Latin Square* to allocate the new participants.

Figure 7 shows the first screen of the tool. This screen shows a list of all existing experiments to allow the participant to select the experiment that he is going to perform.

Figure 8 show the next screen, where we ask the name and email of the participant. We clarify to the participant that these information is optional and that he does not need to inform if he does not want to.

After the participant informs his name and email, we start the tasks. Figure 9 is an example of the screen with a task from our warm-up tasks. We tried to show the code that looks as similar as possible as the participant is used to work in their day-to-day.

When the participant thinks he knows the right answer, he can click on any point of the screen, and the tool will pause the execution as showed in Figure 10. We collect the start and end time of each pause to remove this duration from the execution's total time and remove the collected *eye tracking* points on this interval. On this screen, the user has two options: getting back to the task or providing an answer.

If the participant clicks to provide an answer, we show a modal screen (Figure 11) to collect his answer. If the participant provides the correct answer, we show him the next

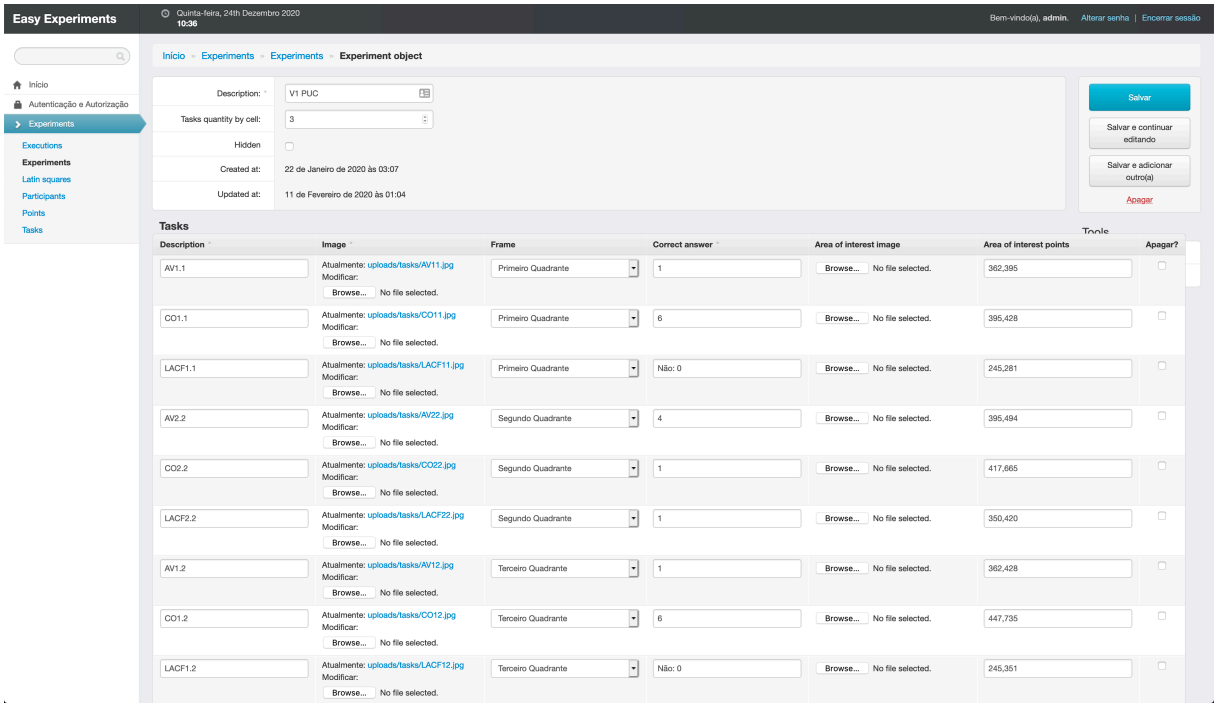


Figure 6 – Admin area - List of Experiments.

Figure 7 – List of experiments registered in the tool.

task until he answers all the tasks allocated for him. If he provides the wrong answer, we allow him to get back to the task and provide a new answer anytime. All answers are stored in the database, whether they are correct or incorrect.

Figure 8 – User information (These information are optional).

```c
#include<stdio.h>

void print_greater(int a, int b) {
    if (a == b) {
        printf("numero iguais");
    } else if(a > b) {
        printf("%d", a);
    } else {
        printf("%d", b);
    }
}

int main() {
    print_greater(5, 34);

    return 0;
}
```

Figure 9 – Task presented to the participant.

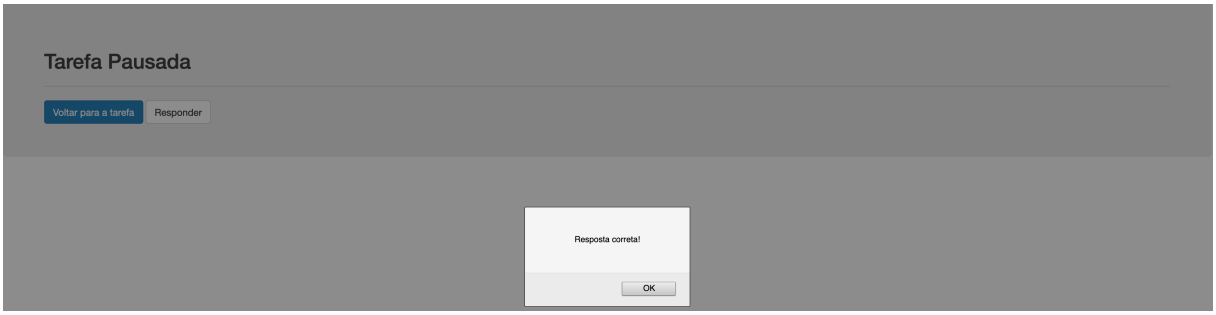


Figure 10 – Task Paused.

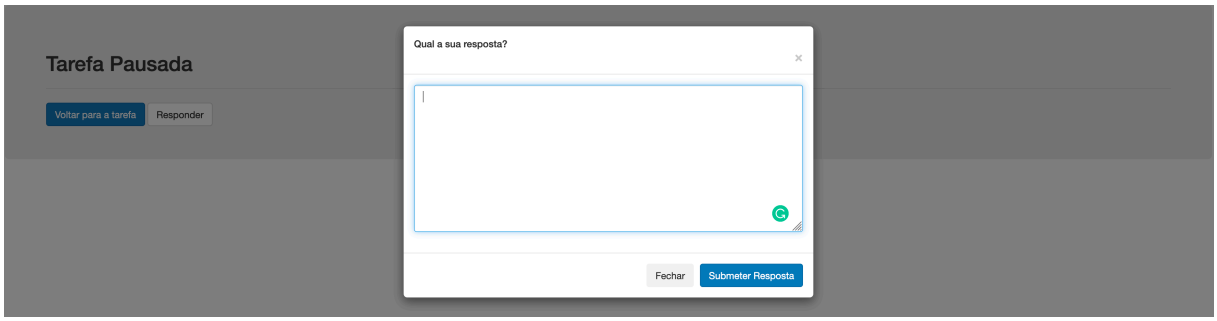


Figure 11 – Task Response Form.

4 Results and Discussion

In this section, we present the results of our research questions along with a discussion.[1]

4.1 RQ_1: To what extent do atoms of confusion affect task completion time?

We analyze our data under two perspectives: from an aggregated perspective, i.e., all the three types of atoms together, and individual perspective, i.e., each atom type separately. The boxplots of Figure 12 shows some descriptive statistics related to the total time spent by the participants to conclude the different tasks on both perspectives. We perform a comparison between the control group, i.e., with No Atom, and treatment group, i.e., With Atom. Some relevant information could be drawn from this figure.

The Aggregate boxplot represents an aggregation of the tasks, either with atoms or with no atoms. Under the aggregated perspective, the total time median when analyzing code with atoms of confusion (294.5 seconds) is 65.4% greater than the median when analyzing code with no atoms of confusion (178 seconds). The observations related to the total time to conclude all tasks, when analyzing code with no atoms of confusion, lie between 74 and 778 seconds; in contrast, the observations when analyzing code with atoms of confusion lie between 91 and 736 seconds. There is only a small overlapping between the two "boxes", which leads to some evidence that understanding code with atoms of confusion is more *time consuming* than understanding code with no atoms of confusion.

We tested our first null hypothesis and found evidences for rejecting it ($\text{p}-value = 0.001972 < 0.05 = \alpha$). This leads to the first conclusion of this study.

Under an individual perspective, considering the hypothesis test for each type of atom of confusion separately, we found diverging results. Two atoms of confusion allowed us to reject the null-hypothesis $H1_0$, particularly, the atoms *Assignment as Value* ($\text{p-}value = 0.008$) and *Logic as Control Flow* and ($\text{p}-value = 0.0001$). In both cases, they favored the code with no atoms.

Even though the size of the code can interfere in time, making it longer to look at more elements is not necessarily true. For instance, in Figure 4, the number of elements added to remove the atoms is considerably higher, especially in the *Conditional Operator*. The code with atom usually has longer lines with more operations, but its corresponding version with no atom has a higher number of lines. These results suggest that, by removing

[1] All results are available at the GitHub repository website: https://github.com/easy-software-ufal/Atoms-of-Confusion-Experiment-Data

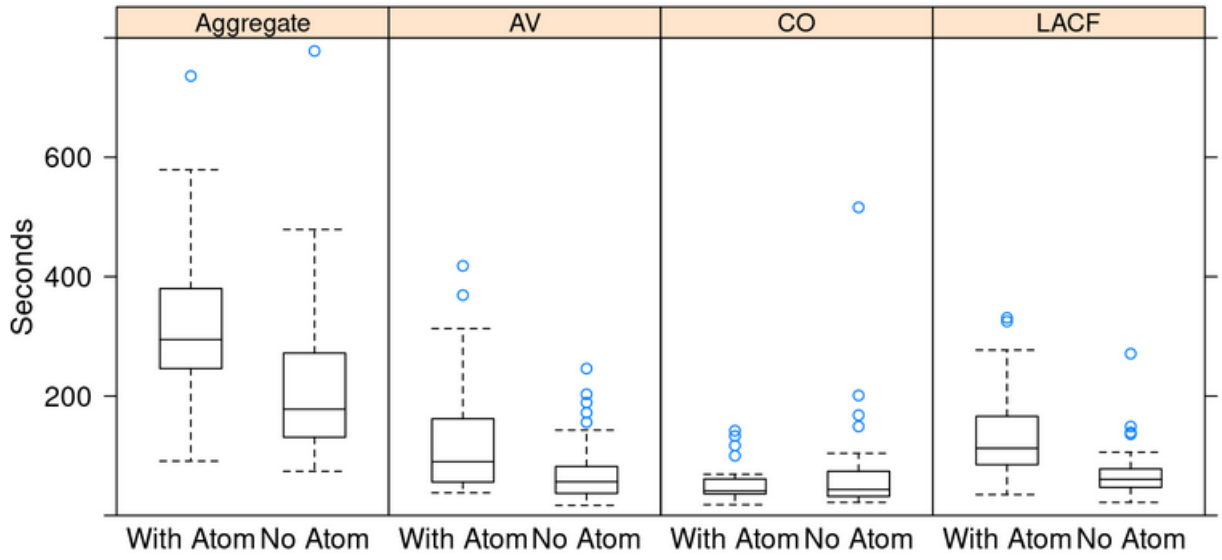


Figure 12 – Time to conclude the tasks. Aggregate = aggregation of all tasks; AV = *Assignment as Value*; CO = *Conditional Operator*; LACF = *Logic as Control Flow*.

the atom, the developers tend to achieve slightly higher productivity, even in longer-code with more elements. The slight increase in the time for the *Conditional Operator* with no atom can be explained by the relatively large amount of code added to remove such atom (see Figure 4b).

All the three evaluated atoms were present in the catalog of Gopstein et al. [4] and were evaluated by them. The only atom that showed distinct results from theirs was the *Conditional Operator*, which in their work showed to be confusing with statistical differences, but we could not observe such effect. The discrepancy might be due to other factors involved, such as methodology. Our methodology is distinct from theirs in some aspects. For instance, we have used code from real projects, we assigned fewer tasks to participants, and the participants were not exposed to the same code with atom and with no atom. Besides, in our case, there is also a possibility that both codes with and with no atoms are confusing due to nesting. The fact is that in the version with no atom, for *Conditional Operator*, we have many more lines of code, which can affect time.

Previous studies have investigated the prevalence of atoms on real projects. Among the investigated atoms, *Conditional Operator*, *Logic as Control Flow*, and *Assignment as Value* were on the top seven of the most frequent atoms in projects according to Gopstain et al. [3]. On Medeiros et al. [5], *Conditional Operator* and *Assignment as Value* were found on the top three most commonly used. Given that *Conditional Operator* is commonly used on both evaluations, and we found contrasting results with Gopstain et al. [4] regarding its real effects, we need more studies on this topic. According to Medeiros et al. [5], participants accepted patches to remove the three evaluated atoms in this study.

However, at least a patch of each atom was also rejected.

> **Finding 1.** The presence of atoms of confusion increases 43.02% in time required to understand code correctly.

4.2 RQ_2: To what extent do atoms of confusion affect task accuracy?

We followed a similar approach to investigate the second hypothesis, which relates to answers correctness or accuracy when analyzing code with atoms of confusion. We first carry out an exploratory data analysis. Figure 13 shows boxplots that present some descriptive statistics related to the number of submitted answers to conclude all tasks under an aggregated and individual perspective. Under an aggregated perspective, which is the Aggregate boxplot, the median number of answers when analyzing code with atoms of confusion is the same when analyzing code with no atoms of confusion, which is 1. This result means getting the task solved on the first try. They only differ in terms of discrepant values. The same applies to the atoms individually. Consequently, we cannot infer a sound conclusion about our second hypothesis $H2_0$ by only observing accuracy alone.

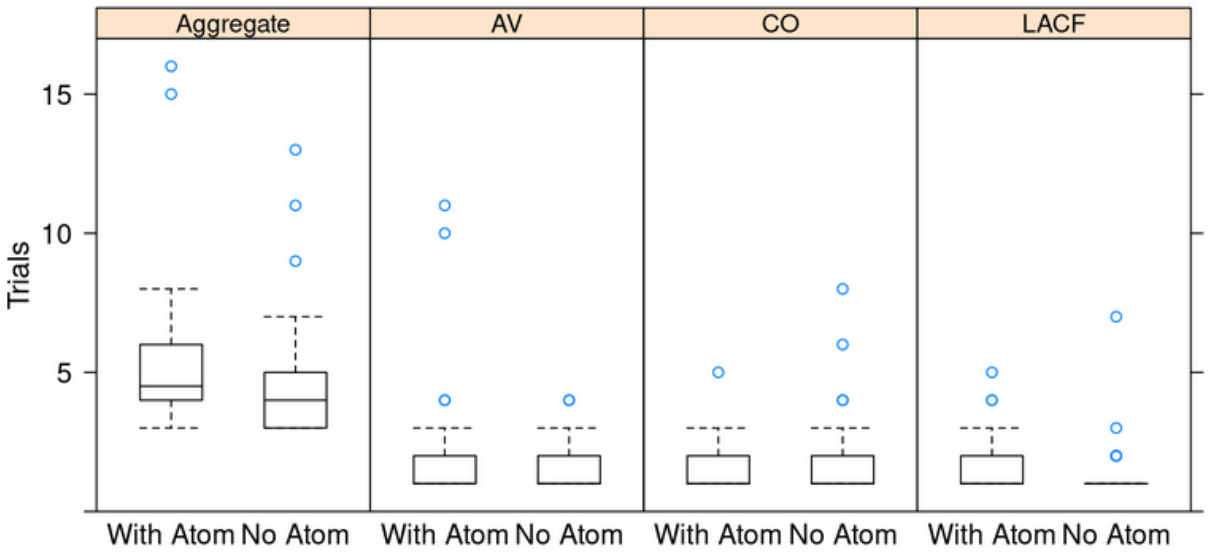


Figure 13 – Number of trials to conclude all tasks. Aggregate = aggregation of all tasks; AV = *Assignment as Value*; CO = *Conditional Operator*; LACF = *Logic as Control Flow*.

One main reason why conclusive results could not be obtained by analyzing accuracy alone may rely on the fact that we have used very simple tasks, with small pieces of code with a few operations. Thus, the number of submitted answers did not vary so much. In

Gopstein et al [4], for instance, even though the tasks were also small, the number of tasks answered by participants was higher.

Finding 2. The presence of atoms of confusion does not impact on accuracy, differently from a previous study [4].

4.3 RQ_3: To what extent do atoms of confusion affect the focus of attention?

By the distribution of visual attention of the developers and how the focus of attention change over distinct code elements, we can infer whether a particular code element is confusing. The distribution of attention can be assessed by the analysis of heatmaps. The colors in the heatmap represent the relative concentration of gaze points over an area, and the higher the amount of points on it, the more intense the color gets. In other words: if the participant spotted a region for too long, that region is likely to get an intense color, which is an indication that the attention was directed to that area. Previous works have showed that atoms can cause confusion [4, 5]. Thus, if the atom of confusion areas require more attention from the participants, those areas are likely to appear on more intense colors on the heatmaps, which is a phenomena we are interested in investigating. Most importantly, we aim to verify the extent of the impact of the atoms on the attention of the participants. To get more insights on this matter, we separated the areas of interest to perform a more specific analysis. We have defined these Areas Of Interest (AOI) as the lines of code where the atom of confusion is present, and correspondingly, lines of the version with no atom. In Figure 4, we present the areas of interest in the code, which are the areas in which both codes differ.

Table 3 – Summarizing Metrics. Bold font denotes a significant statistical difference with a significance level of 5%. Signal ↑ corresponds to an increase with atom while ↓ corresponds to a reduction.

Atoms	Time	Errors	Points in AOI	Entries in AOI	Gaze Trans.
AV	↑ 66.05%	↑ 46.7%	↓ 34.5%	↑ 17.9%	↑ 15.7%
CO	↓ 28.9%	↓ 9.3%	↑ 85.0%	↑ **121.2%**	↑ 93.7%
LACF	↑ 91.7%	↑ 16.3%	↑ 26.2%	↑ 36.7%	↑ 22.7%
Aggregated	↑ **43.0%**	↑ 16.2%	↑ 20.6%	↑ **49.3%**	↑ 36.8%

In Figures 14, 15, and 16, we see a comparison of the three heatmaps, one of each evaluated atom, showing how the attention is distributed over different parts of the

program relative to time spent by the developers. In the heatmaps, blue (cold) areas are areas of the code that did not receive much attention while red (hot) areas represent the areas that received most of the attention. Each heatmap is an aggregation of individual heatmaps of the participants who performed the same task.

In Figure 14, we have a more clear distinction in the heatmaps for *Logic as Control Flow* atom. We observe that the attention is mainly focused on one main region where the atom is located while when the atom is refactored, the developers directed their attention to two main regions, causing a higher distribution of attention over distinct parts. With atom, the total time was increased by 91.7% and, in it, participants focused the area of the atom with 26.2% more points. Participants also needed to enter the atom area 36.7% more and performed 22.7% more transitions. Those numbers give us and indication that code with atom is more confusing.

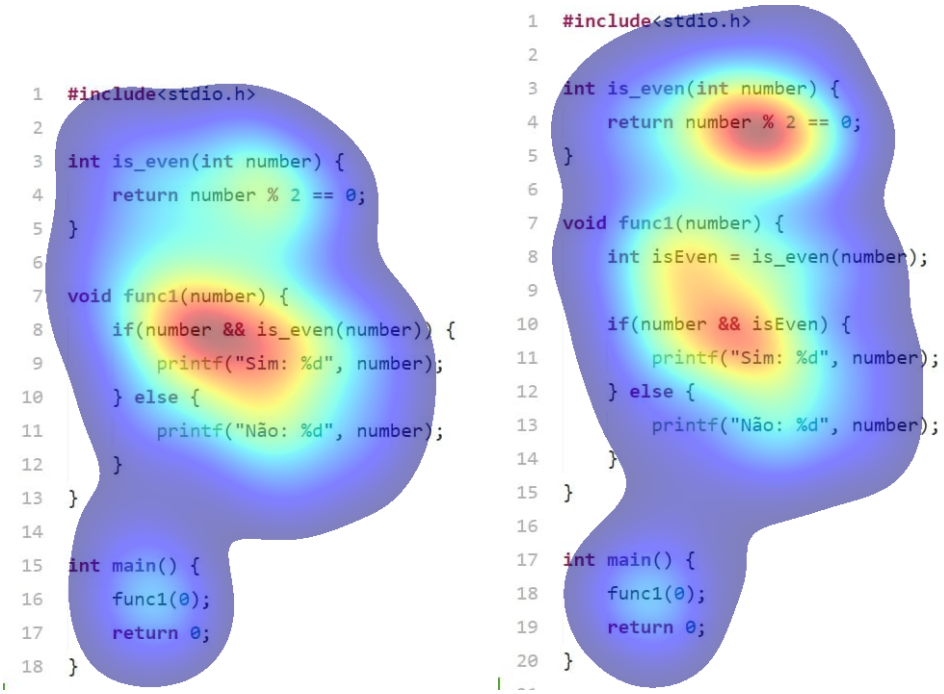


Figure 14 – *Logic as Control Flow* - With atom vs. No atom.

In Figure 15, we could not observe so much difference in the heatmaps for the *Assigment as a Value* atom. The developers tend to focus their attention on the atom region. When the atom is refactored, the developers focus on the relative lines of code that are extracted. However, we observe that the attention is more restricted and focused on a relatively smaller area at the right-hand side of the heatmap compared to a more sparse at the left-hand side. It is important to emphasize that the color intensity is relative to the amount of time to solve the code. However, according to Table 3, with atom, the time spent in the code increases along with errors. Gopstein et al. [4] have showed that this atom causes confusion, and by intense color on the atom area, we conclude that the participants are focusing on it because they are confused. However, in our results, the

number of points in this area is actually decreased by 34.5% which means that other parts of the same code were also confusing. This conclusion is supported by that fact that we have 17.9% more entries in atom area and 15.7% more transitions in the code.

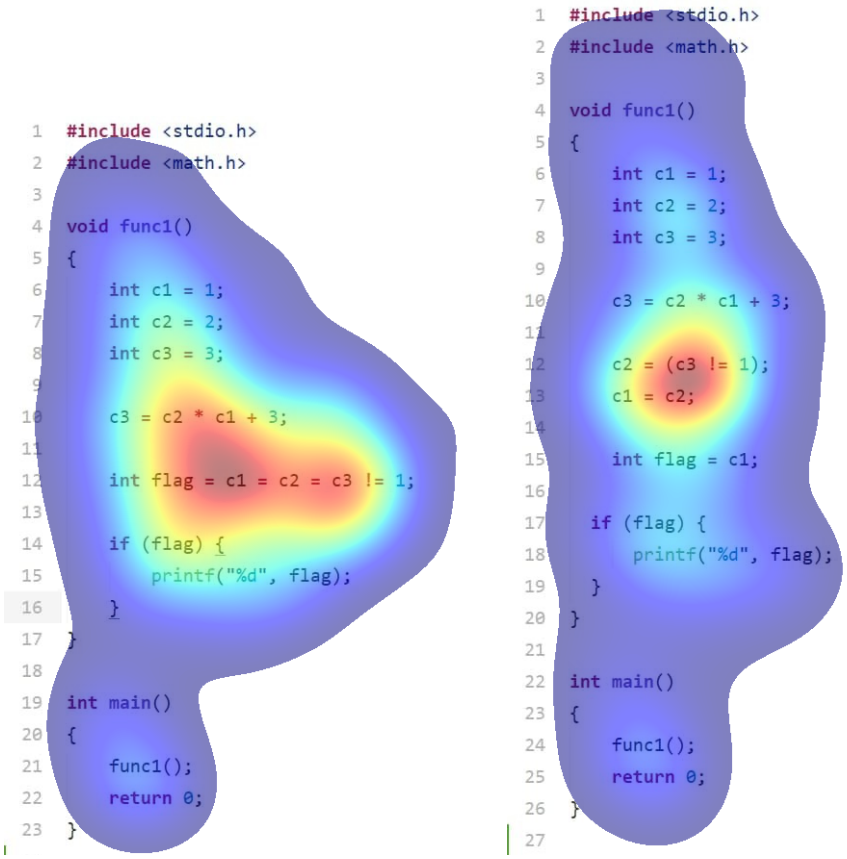
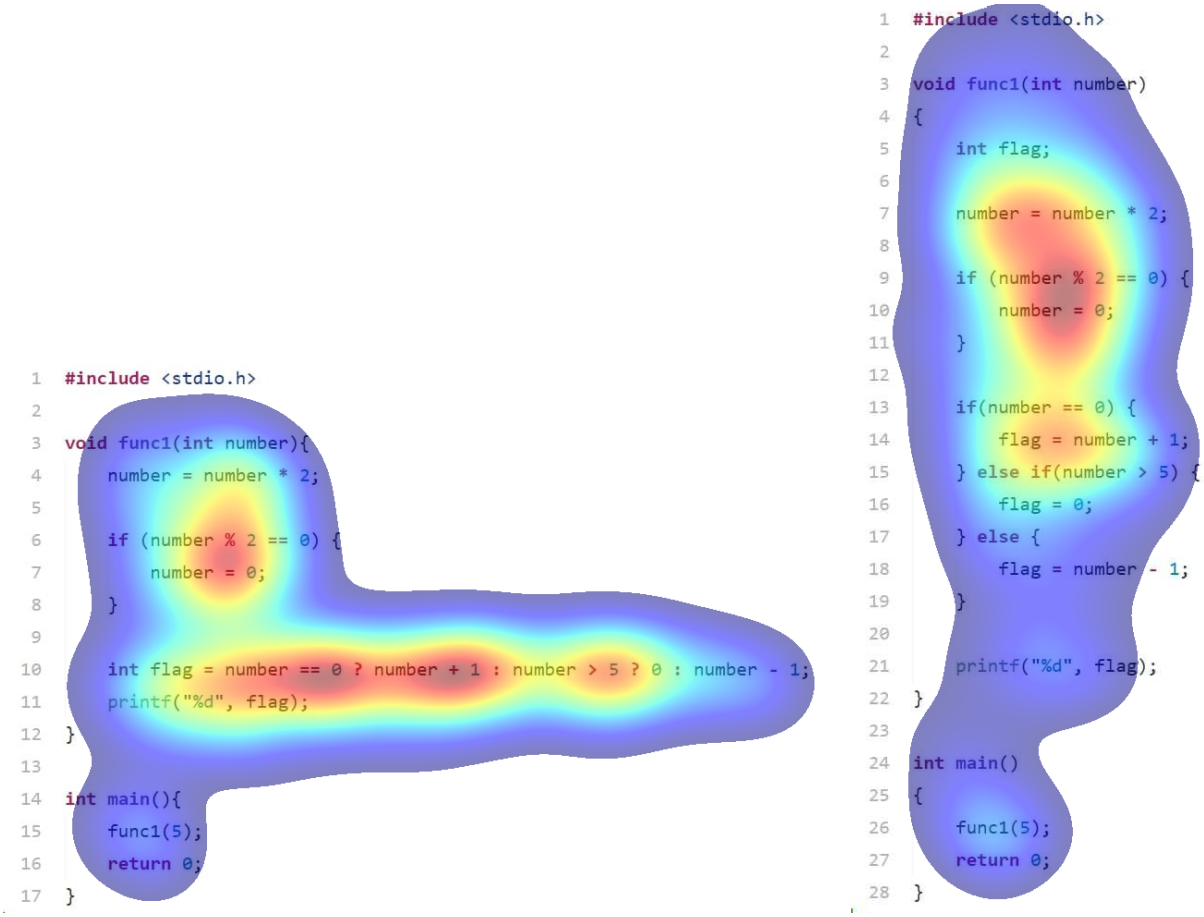


Figure 15 – *Assignment as Value* - With atom vs. No atom.

In Figure 16, at the top, we observe that the attention is distributed more horizontally while at the bottom, vertically, given the disposition of the refactored code. The visual difference is not so clear, however, in terms of the concentration of points in the area of interest, there is a clear difference. Gopstein et al. [4] showed that the atom causes confusion but, in our results, time and accuracy did not support those results. However, the eye gaze metrics indeed support those results. There is a higher concentration of points on the atom region, 85% more points, according to Table 3, which means that the atom region requires more attention. In addition, there is a statistically significant increase by 121.2% in entries in the atom area, which supports confusion associated with that area. It also does not seem to help understanding other parts of the code since we have 93.7% more transitions with atom.

Now, we compare the entries and exits in AOI for one specific user executing the tasks for the same kind of atom and their corresponding refactored version. Figure 17 compares the execution performed by such user considering the task with a *Conditional Operator* and its corresponding refactored version. The x-axis is the timeline in percentage



```c
#include <stdio.h>

void func1(int number){
    number = number * 2;

    if (number % 2 == 0) {
        number = 0;
    }

    int flag = number == 0 ? number + 1 : number > 5 ? 0 : number - 1;
    printf("%d", flag);
}

int main(){
    func1(5);
    return 0;
}
```

```c
#include <stdio.h>

void func1(int number)
{
    int flag;

    number = number * 2;

    if (number % 2 == 0) {
        number = 0;
    }

    if(number == 0) {
        flag = number + 1;
    } else if(number > 5) {
        flag = 0;
    } else {
        flag = number - 1;
    }

    printf("%d", flag);
}

int main()
{
    func1(5);
    return 0;
}
```

Figure 16 – *Conditional Operator* - With atom (top) vs. No atom. (bottom)

and the y-axis represents the critic region, basically with three points: upper the critical region, inside the critical region, and lower the critical region. We perform the same analysis for *Assignment as a Value* in Figure 18, and for *Logic as Control Flow* in Figure 19.

The code with the *Conditional Operator* atom required, on average, less time to be solved but more visual effort due to some reasons. We analyze the focus of visual attention and code regressions. These transitions represent the inputs and outputs in the atom area, the AOI (see Figure 17). With no atom, the participant makes fewer transitions, going up and down less often, with about 5 long peaks, and 2 of them touching the AOI in about 20 seconds. However, when considering the code with an atom, the participant stays in the AOI for about 41 seconds against only 22 seconds for the function with no atom. We can see that for the code with atom the participant needs to make much more transitions going back and forth more often, with about 5 long picks in the area of interest. The increased number of transitions may indicate the need for a higher visual effort. We call regression the number of times the user needs to get back to some previous code part. The high number of regressions is also an indication of confusion since the participant had to get back to parts of the code already read many times. Thus, the permanence analysis in the AOI, such as the number of regressions, may indicate a symptom of confusion introduced

by the atom.

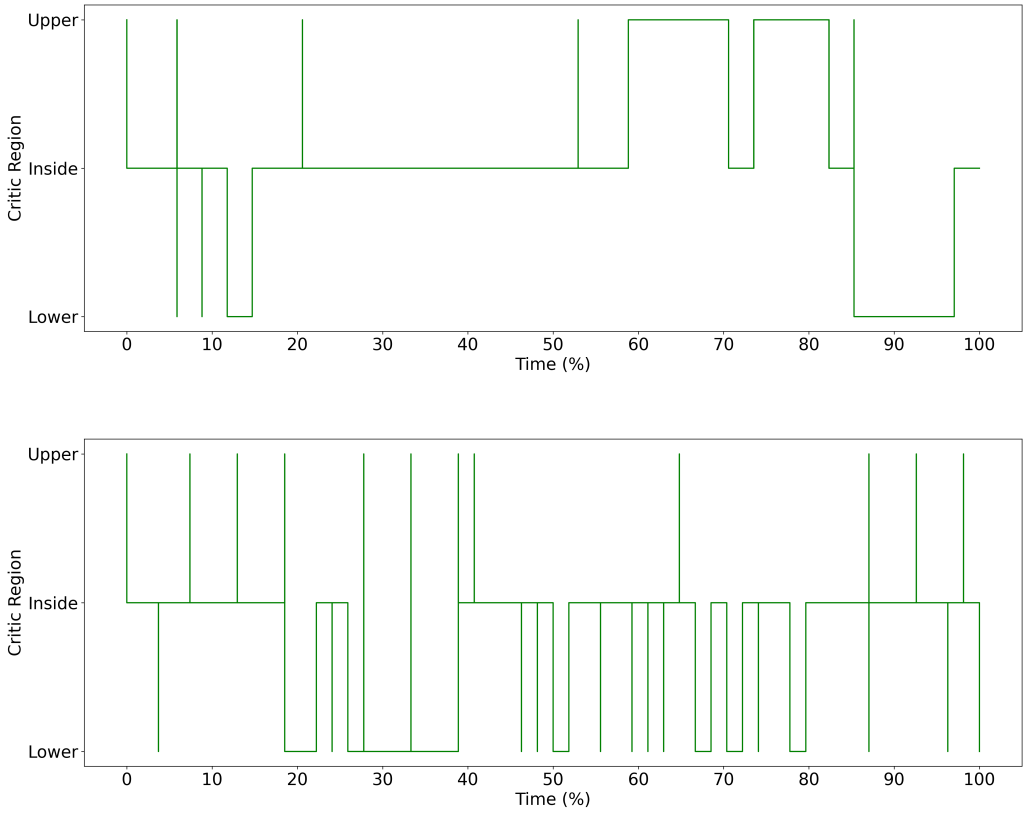


Figure 17 – AOI permanence graph and regressions for a *Conditional Operator* with no atom (graph at the top) and with atom (graph at the bottom) of one specific user.

Figure 18 shows the inputs and outputs in AOI for a user performing the tasks with an *Assignment as Value*. This figure shows a similar pattern when compared to Figure 17, where the participant needs to make much fewer transitions when performing the task with no atom. It also has about 5 long peaks, and 2 touching the area of interest less than 20 seconds. When comparing to the same task when the atom is present, although the participant keeps his visual attention in AOI, he had to go back and forth many times, which might indicate confusion, since the subject had to get back to code already read some times.

In Figure 19 we perform the same analysis presented in Figure 17 and Figure 18. However, at this time, we focus on the *Logic as Control Flow* atom. Again we can see the same pattern where we have fewer transitions and regressions for the task with no atom and a higher number of transitions and regressions for the task with an atom. We can also see that, in this case, the subject spent twice as much time to perform the task with atom than the task with no atom. The user also keeps his visual attention much more time at the AOI.

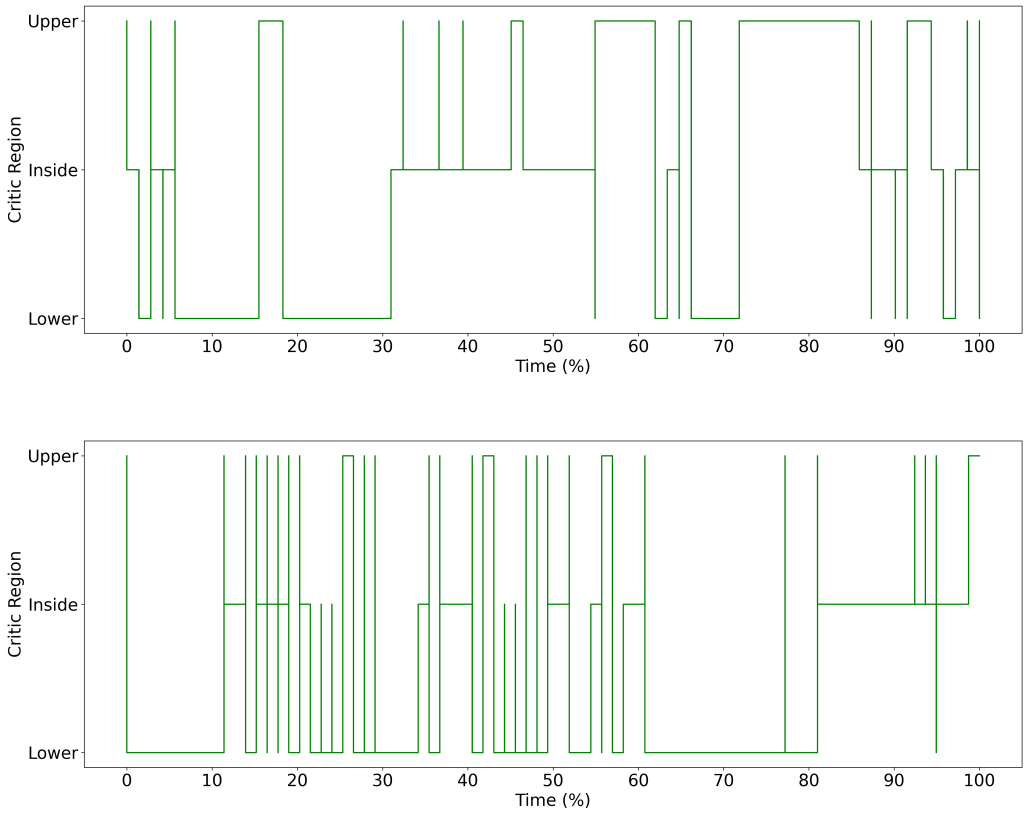


Figure 18 – AOI permanence graph and regressions for a *Assignment as Value* with no atom (graph at the top) and with atom (graph at the bottom) of one specific user.

After analyzing the executions of one specific user, we now analyze all the executions (from all participants together) for every single task. To do so, we consider the execution of each subject and plot all the graphs on top of each other (i.e., we overlap all plots). As in the previous graphs, the x-axis is the time in percentage and the y-axis represents the critic region.

Figure 20 shows this analysis for the first task containing an *Assignment as Value* (graph at the top) and its corresponding refactored task (graph at the bottom). The smaller number of blank areas indicates the greater number of entries and exits in the critical area, confirming the previous graphs' pattern, presented in Figures 17, 18, and 19. We can see that the graph with code data without an atom has more blank areas than the graph of code analysis with an atom. The same pattern can be observed in Figure 21 that shows the entries and exits in AOI for the second task of this same atom (*Assignment as Value*). We can observe that in the graph for the code with no atom, a higher number of blank spaces indicates a smaller number of entries and exits from the critic region.

Figures 22 and 23 present the same analysis for the tasks containing the *Conditional Operator* atom. In Figure 22 we can see the pattern found in the previous ones more

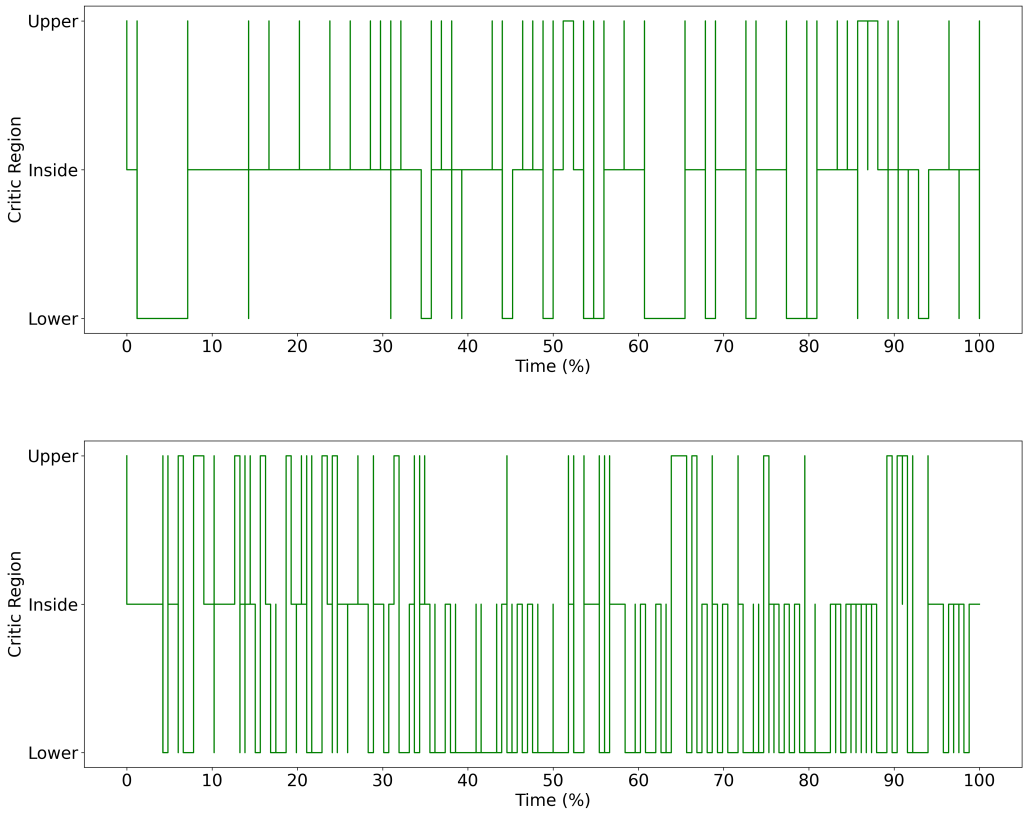


Figure 19 – AOI permanence graph and regressions for a *Logic as Control Flow* with no atom (graph at the top) and with atom (graph at the bottom) of one specific user.

clearly, where the participants performed much more transitions and regressions to the area of interest when analyzing code with an atom. The graph with no atoms has some significant blank spaces between the transitions entering and leaving the critic region. In contrast, the graph for the task with atoms contains smaller blank spaces.

We now analyze the *Logic as Control Flow* atom (see Figures 24 and 25). For the first task, we cannot see the same pattern from the other tasks. We have fewer significant blank spaces, and both graphs look similar. However, for the second task (Figure 25), the graphs are different, and we can see that the task with atom has more entries in the critic region.

Finding 3. The presence of atoms of confusion increases in 36.8% the number of gaze transitions, 49.3% the number of entries in AOI, 163.06% the number of regressions to AOI, and 20.6% the number of points in AOI.

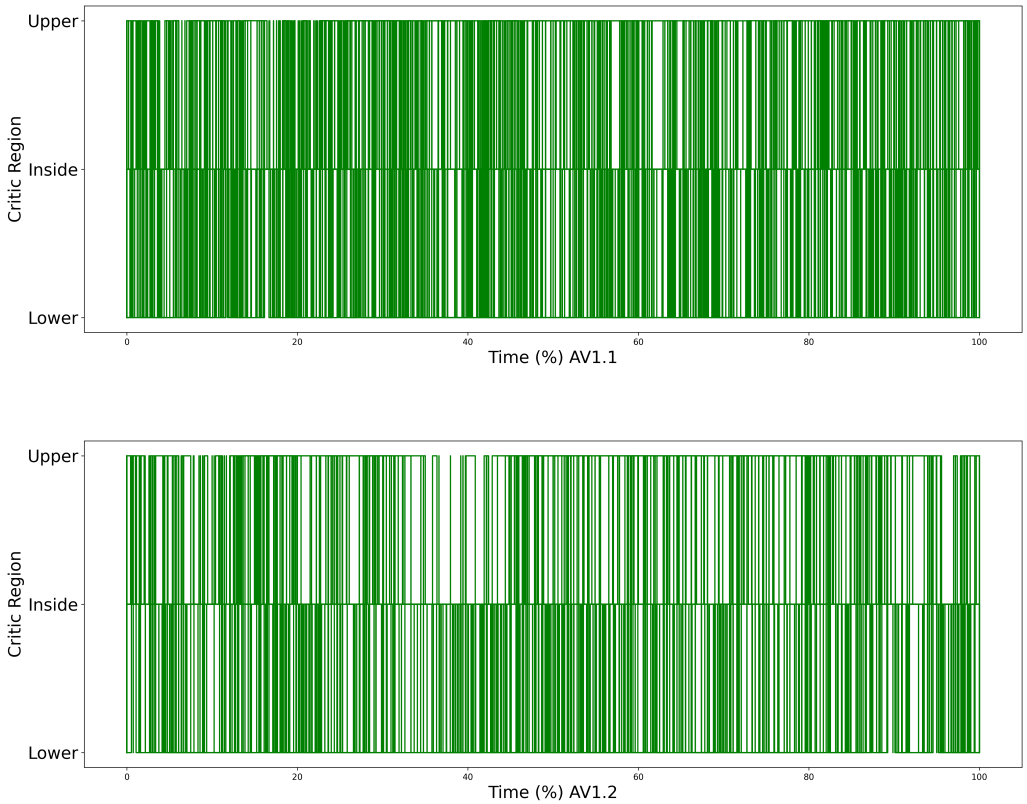


Figure 20 – AOI permanence graph and regressions for the first task with an *Assignment as Value* (graph at the top) and another with this atom removed with the execution of all subjects. The plots of all subjects are overlapped.

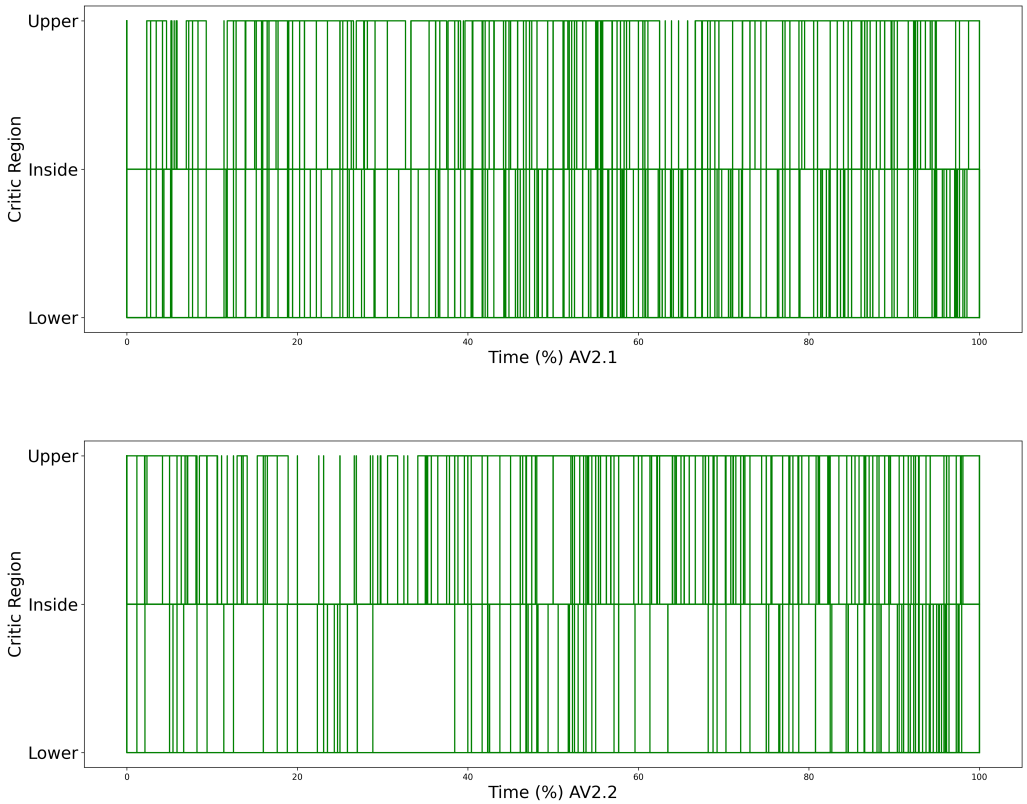


Figure 21 – AOI permanence graph and regressions for the second task with an *Assignment as Value* (graph at the top) and another with this atom removed with the execution of all subjects. The plots of all subjects are overlapped.

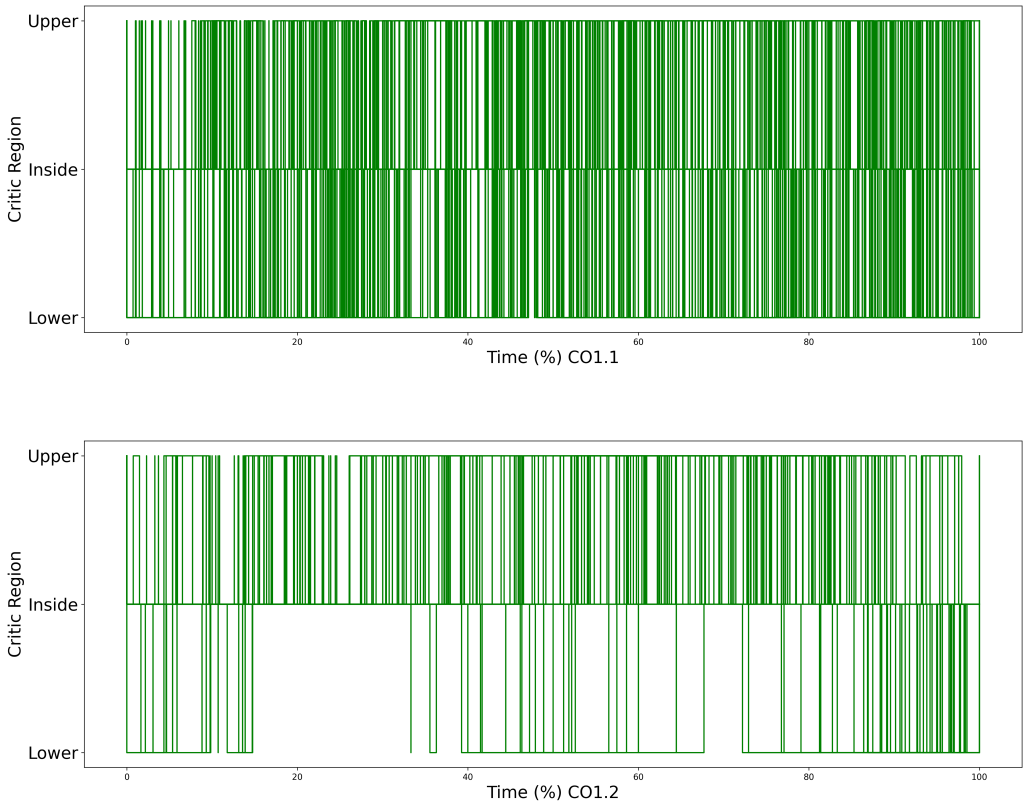


Figure 22 – AOI permanence graph and regressions for the first task with an *Conditional Operator* (graph at the top) and another with this atom removed with the execution of all subjects. The plots of all subjects are overlapped.

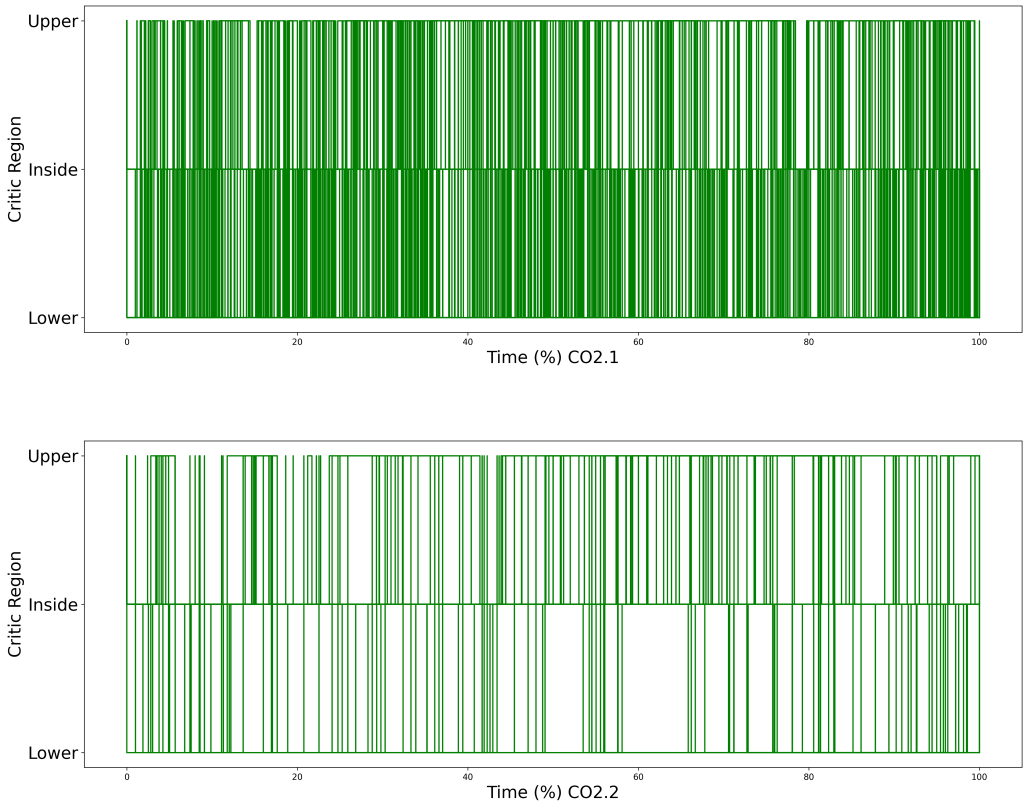


Figure 23 – AOI permanence graph and regressions for the second task with an *Conditional Operator* (graph at the top) and another with this atom removed with the execution of all subjects. The plots of all subjects are overlapped.

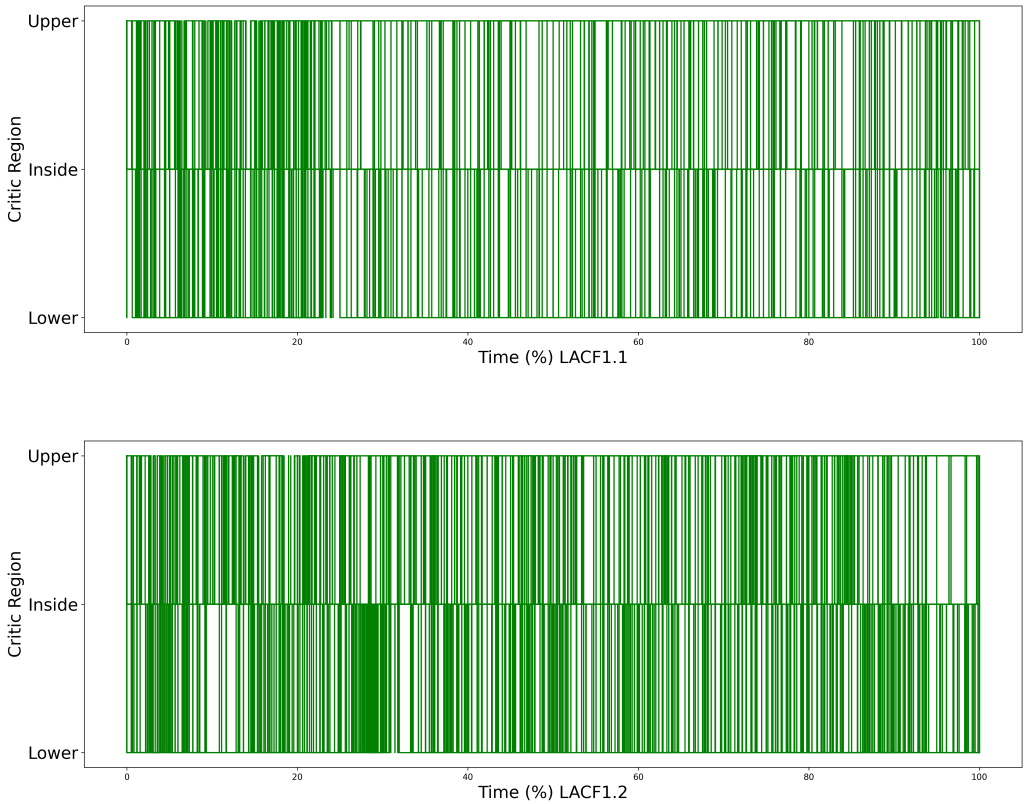


Figure 24 – AOI permanence graph and regressions for the first task with an *Logic as Control Flow* (graph at the top) and another with this atom removed with the execution of all subjects. The plots of all subjects are overlapped.

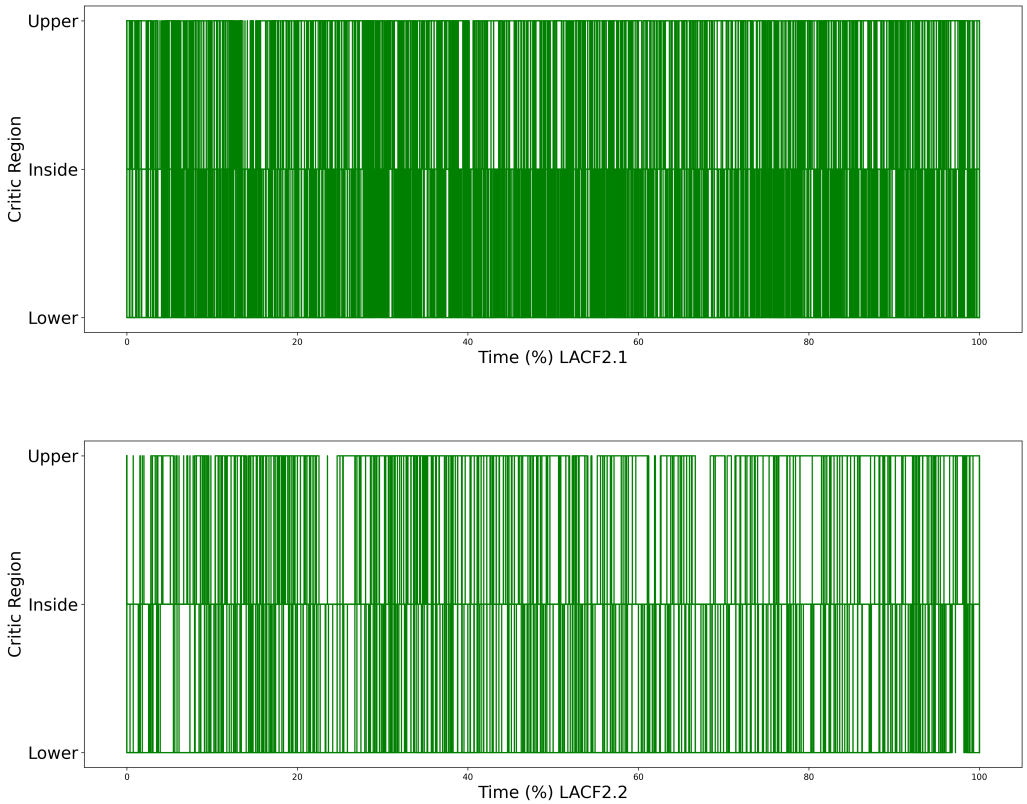


Figure 25 – AOI permanence graph and regressions for the second task with an *Logic as Control Flow* (graph at the top) and another with this atom removed with the execution of all subjects. The plots of all subjects are overlapped.

4.4 Threats to Validity

We now discuss potential threats to the validity of our controlled experiment. We distinguish between the threats using the classification of Wohlin et al. [25].

Conclusion Validity. Since our data do not follow a normal distribution, our hypotheses tests are based on a nonparametric test named Kruskal-Wallis. The Kruskal-Wallis test can be used to determine if there are statistically significant differences between two or more groups of an independent variable on a continuous or ordinal dependent variable. Regarding the reliability of our measurements, the participant's eyes can gaze over areas out of the screen or out of the code, and the eye tracker can still capture those points. We observed, by analyzing the generated heatmap, that some participants tend to look into emptiness sometimes when they are thinking. However, this situation is likely to occur in both treatments, i.e., with atoms and with no atoms. Regarding the points captured, techniques are commonly employed to reduce their number, which the literature refers to as fixations. We did not reduce their number, but the raw data points can also be used to generate the heatmaps once they show the density of concentration of such points over distinct areas.

Internal Validity. There were some other threats to validity in this study that we realized during the experiment execution, such as the rotating chair used by the participants. This rotating chair can lead the eye tracker device to lose contact with the participants' eyes. Nevertheless, we minimized this threat because we calibrated the eye tracker device for each participant right before the execution of the experiment. The potential different luminosity between the two rooms environments we used to execute the experiment can also be considered a threat, since they can change the sensibility of eye-tracking and expose different conditions to participants. To minimize this threat, we closed all curtains in both rooms to avoid the external luminosity. In addition, the size of the font and style might have an influence on the participant's attention. To mitigate this threat, we have chosen a popular font style as well as a size that fits the screen. All tasks are displayed in the same font size, highlighted with a light theme, and no bold font, to mitigate possibles effects. The time duration of the whole experiment can also influence visual effort if it is too long. We mitigated it by using simple and only a few tasks to not let participants tired.

Construct Validity. Our controlled study can involve threats regarding mono-operation bias or possible effects of interaction between treatments. To deal and control these threats, we employed in our experiment the Latin Square design.

External Validity. In our experiment we have used programs written in the C language, which might restrict the generalization capacity to other languages. However, we used simple code snippets with constructs incorporated in several programming languages,

such as variable declarations and assignments and `if-else` statements. Additionaly, we cannot generalize the results of our experiment to other code comprehension scenarios—although we explored three different types of atoms of confusion: *Assignment as Value*, *Conditional Operator*, and *Logic as Control Flow*. Moreover, we cannot generalize our results to the population of professional developers since most of our experiment participants were graduate students. However, some studies argue in favor of using students to conduct controlled experiments [2, 26]. Our strategy in this study was to conduct an empirical experiment that initially consisted of a seek on open source repositories to find candidate functions to the experiment; then we created a complete environment to make it possible to execute the experiment with the least possible interaction, and then the experiment allowed us to better understand some causal relationships of code comprehension of code containing atoms of confusion. This work aims to validate preexisting works favoring internal validity.

4.5 Implications

Our results reveal and reinforce what a previous study has been claiming. From an additional perspective which includes eye tracking, the presence of atoms can confuse, affecting developers' productivity, making them to spend more time and to misjudge code behavior, and bringing effects to their attention, which makes them to focus on the atom regions.

The catalogs of atoms of confusion created by previous works are not yet widespread among software developers. Thus, developers need to be aware of any possible atoms of confusion which appear as bottlenecks in code comprehension. This way, we invite and suggest the community to give a greater diffusion of these atoms and the negative effects of their presence on the code.

These results also motivate the development of plugins and tools in IDEs and code editors that can make it easier for programmers to detect and refactor atoms of confusion in the code. A good practice is also to make project managers and senior developers more concerned about these atoms to be more restrictive when doing code reviews and accepting contributions (e.g., merging pull requests in GitHub) from other developers.

The methodology we adopted by using an eye tracker to assess visual attention has shown to be a promising alternative to analyzing code comprehension in the presence of atoms of confusion. It reveals how a particular atom can change the amount or direction of the attention, its distribution, and on this perspective, we have a more realistic idea of the impact of atoms on the code. The eye perspective is not explored by previous works in the context of atoms of confusion.

Last but not least, in this work, we have found empirical evidence that atoms of confusion can cause code misunderstandings and that some types of atoms are more prejudicial than others. Thus, we encourage researchers, the academic community, and practitioners to conduct more controlled experiments, specially considering types of atoms of confusion not explored in this study.

5 Related Work

Code comprehension is a frequently explored domain in computer science, particularly in the software engineering field, since a significant portion of the maintenance effort is dedicated to understanding existing software. A considerable amount of effort to properly understand code can be a problem not only for the business environment but also for education since code comprehension tasks are widely used in assessments, mainly in the first years of graduation [27, 28].

Regarding code comprehension, some prior works [3–5] have studied some particular small code patterns which have a big potential to make the code harder to understand, leading to code misunderstandings. These code patterns were named *atoms of confusion* [3, 4], or *misunderstanding patterns* [5]. Such code patterns can even lead to the introduction of bugs [3, 5]. Medeiros et al. [5] have exposed a case of bug introduction related to the atom of confusion *Dangling Else* in a function used on a codec library called OpenH264 for which they have submitted a pull request removing this bug.

Gopstein et al. [4] have performed an experiment with 73 participants and showed empirically that these atoms of confusion could lead to a significantly increased rate of misunderstanding when compared with code without atoms of confusion. As they affirmed [4], people and machines often draw different conclusions about the behavior of a piece of code. These different conclusions can naturally lead to bugs. Still, according to Gopstein et al. [4], atoms of confusion can also cause diminished productivity, faulty products, and higher costs. They have also cataloged 15 atoms of confusion and provided a methodology for empirically deriving these complex atoms. In their study [4], they have conducted two experiments. One of them is similar to ours. However, we use an eye tracker device to assess the impact of atoms of confusion on the attention of the developers while they are trying to understand the code. We observed an increase by 36.8% of gaze transitions in code snippets with atoms.

Other works [14, 15, 18, 19, 29] have studied the correlation and effectiveness of visual attention tracking on code and text understanding. They have shown that a longer fixation on any specific part of the code can indicate a higher cognitive workload trying to understanding the text meaning. They also found that the more frequent the word is in the text, the less difficult it is to understand. However, to the best of our knowledge, no studies with eye tracking were performed to investigate the impact of atoms in code comprehension.

In their experiment, Gopstein et al. [4] instructed subjects to step through the program as if they were the computer, execute each instruction in their mind, and to

record the standard output of the program. They also modified the experimental programs to include a *printf* after every control flow operation and otherwise frequently enough to gather information from the subject. Our experiment followed a similar approach, but we instructed the subjects to record the outputs of each function directly in our system. The system only allows the subject to move to the next function when he or she provides the right answer. This approach was mixed with the use of an eye tracker device to record the subjects' eyes movements.

Gospstein et al. [3] showed that there is a strong correlation between atoms of confusion and bug fix commits as well as a tendency for atoms of confusion to be commented. They also observed a higher rate of security vulnerabilities in projects with more atoms. They did it by selecting 14 of the most representative and popular open-source C and C++ repositories to measure the prevalence and significance of atoms of confusion. They correlate the occurrence of these atoms on the selected projects with some external factors like bugs and comment rates. They showed that projects with more atoms tend to have more CVEs (Common Vulnerabilities and Exposures) and bugs by domain. In this experiment, they did not analyze developers' performances or asked them about difficulties in understanding the code; they searched for atoms of confusion in existing code. They analyzed the correlation between the rate of atoms on these repositories with the presence of bugs and if these codes were refactored, and the atoms were removed from the code, among other questions. In our study, we include the measurement of how long developers take to understand code extracted from real open-source projects and which part of the code they fix their eyes to have more accurate data about the difference in code understanding with and with no atoms.

Another study conducted by Medeiros et al. [5] tried to better understand the occurrence and relevance of atoms of confusion by mining some repositories seeking for the occurrence of these atoms and also applied a survey with 93 developers. They found that the atoms studied (92%) are highly used in practice and that developers agreed that 6 of the studied atoms hinder code understanding, representing 50% of the 12 atoms studied. They also showed that the atoms are not frequently cited on guidelines, only a few guidelines cite rules related explicitly to atoms. Our experiment tests three types of atoms and allows us to compare if they cause code misunderstanding in practice. Moreover, it allows us to catalog which atoms cause confusion and propose a tool to indicate and refactor them. Besides, this study uses the perception of developers to know if the atom confuses, but if the developer is confused about the meaning of code, he or she can also be confused about the rate of confusion in the code; in order words, he or she can misunderstand the correct behavior of the atoms. The use of eye-tracker allows us to compare the perception of programmers with quantitative data about real confusion generated by the atoms.

More recently, Gopstein [13] conducted an experiment aiming to understand not

only if developers misunderstand some confusing code but also why and how this happens. They experimented both with professionals and students, and they did a qualitative analysis using the think-aloud methodology. One crucial finding of their study regarding atoms of confusion is that there are reasons other than atoms that can lead to errors when analyzing code with atoms of confusion. This means that misunderstandings are not the only reason for confusing code. To better understand this phenomenon, they divided confusion into four categories that lead developers to confuse code, such as unfamiliarity, misunderstanding, language transfer, and attention. They showed that when a programmer misinterprets a code with an atom, the region of the code with the atom of confusion is not always responsible for that misunderstanding; in most situations, the confusion was caused partially or totally by some other factor, such as another atom or potential atom. This opens a new perspective to investigate to what level the atoms cause confusion or if they only confuse the presence of other constructors. In our experiment, we did not include multiple atoms on the tasks. We make the functions as straightforward as possible to avoid many distractions outside the areas of interest.

Most works on atoms of confusion have been investigating the impact of these code snippets in C programs. Castor [30] expanded the investigation to the Swift programming language and used different investigation methods from the previously used by Gopstein [4], such as measuring the infrequency of occurrence in large codebases and expert opinions. Castor identified in the Swift language some sources of confusion that exists in the Gopstein [4] original catalog and presented a preliminary atoms of confusion catalog for Swift. Our work does not try to catalog or find new kinds of atoms; we selected three kinds of atoms present in the first Gopstein's catalog [4] and used an eye-tracking camera to better understand the impact of atoms of confusion during code comprehension tasks.

Regarding the use of an eye-tracking camera, Carter [31] mentions that it is a common method for observing the focus of visual attention. A previous work [32] has shown that visual attention starts mental processes. The greater the cognitive processing, the greater the visual effort employed. Carter [31], mentions that it is possible to have insights on the mental processing through eye activity, regardless of where the subject is looking. For that reason, eye tracking is important to be applied in works that analyze and try to map mental processes.

In another study, Begel [33] investigates the code review process using an eye-tracking camera. He has found that an eye-tracking device can help to discover precisely how engineers scan through source code, looking for suspicious patterns of code. The authors also found that when a particular part of the code catches the person's attention, he/she will slow down to read it more carefully, what reinforces that developers tend to spend more time in some parts of the code, especially when the code catches his/her attention. Begel's work is similar to our work since he analyzed code reviews that is,

at a high level, a comprehension task. However, they analyzed different things such as reading rate. Although they measure time, they could not measure tries because their analysis focuses on giving a code review instead of understanding the output. In our study, in addition to analyzing a specific construct (atoms of confusion), we also collected how often the participant provided the wrong answer to have a metric with respect to misunderstanding. We also provide some further analysis, such as heatmaps and the analysis of gaze transitions entering and exiting in the AOI. The eye-tracking methodologies have been widely used in other some different domains, such as mental health, education, and many other areas where the mental and perception process is involved [14] [15] [16] [17].

Jbara [16] executed an experiment similar to ours, but the authors investigated what they called *regular code* and *non-regular code*. He defines regular as code containing regularities. Regularity is the repetition of code segments (patterns), where instances of these patterns are usually successive. He showed that although it is common to identify larger code, with more code lines or greater cyclomatic complexity as a more difficult to understand code, this is not always true. Regular codes usually have a higher number of lines. The study showed that this type of code does not lead to a worsening of understanding. This happens because developers tend to need less time to understand a block of code since they have just read a similar block. They have also used an eye-tracking device on his experiment. His experiment is similar to ours since the code with an atom of confusion is usually smaller than code with no atom. Although similar, their work has some design differences. Their subjects had to analyze only 1 function, some of them with regular code and others without regular code. In our experiment, we have 12 functions, and each participant had to analyze 6 of them, 3 with atoms and 3 with no atoms. The question the subjects had to answer was almost the same *What does the program do?*, but we developed a system to gather the answers and control all the experiment executions. They were interested in the same metrics (they called correctness, completion time, and visual effort). Besides, they did some qualitative questions to know the participants' opinion on the effort needed to understand the presented programs.

6 Concluding Remarks

In this study, we presented a controlled experiment with an eye tracker with 30 participants to investigate the effects of atoms of confusion on code comprehension. In particular, we analyzed the effects of the atoms on time, accuracy, and participants' attention while specifying the output of six code tasks adapted from real open-source systems. We found evidence that code with atoms of confusion is more time-consuming, unlike code with no atoms of confusion, leading the participants to code misunderstandings. We also learned that developers tend to focus their attention on the particular areas of the atoms more often. We observed an increase by 36.8% of gaze transitions in code snippets with atoms.

In a previous work, Gopstein et al. [4] showed that the presence of these patterns in the code affects developers' performance, i.e., time and accuracy, and increase code misunderstandings. We showed that other aspects related to code comprehension besides time and accuracy, namely attention, should also be taken into consideration being affected by the presence of atoms. This was clear by the increase in the number of points in AOI, entries in AOI, and the number of gaze transitions, indicating more effort when analyzing code with atoms of confusion. These nuances would not be possible without an analysis from the eye perspective. Therefore, we presented a perspective of analysis that contributes to more insights on the impact of the atoms on the attention.

For the developers' community, we recommend that they should avoid writing code with atoms of confusion since they can hamper their code comprehension.

For the research community, we encourage more empirical studies investigating the impact of other particular types of atoms of confusion, since we have seen that some atoms can cause more negative effects than others. Other comprehension code scenarios and activities should also be explored, and other eye-tracking metrics should also be investigated. We believe that more empirical studies on this subject can contribute to widespread catalogs of atoms of confusion among software developers. We also recommend them to propose and implement refactorings in IDEs to remove atoms of confusion automatically.

Bibliography

[1] R. Minelli, A. Mocci, and M. Lanza, "I know what you did last summer-an investigation of how developers spend their time," in *Proceedings of the International Conference on Program Comprehension*, 2015, pp. 25–35.

[2] M. Yeh, Y. Yan, D. Gopstein, and Y. Zhuang, "Detecting and comparing brain activity in short program comprehension using eeg," in *Frontiers in Education Conference*, 2017, pp. 1–5.

[3] D. Gopstein, H. H. Zhou, P. Frankl, and J. Cappos, "Prevalence of confusing code in software projects: Atoms of confusion in the wild," in *Proceedings of the Mining Software Repositories*, 2018, pp. 281–291.

[4] D. Gopstein, J. Iannacone, Y. Yan, L. A. Delong, Y. Zhuang, M. K.-C. Yeh, and J. Cappos, "Understanding misunderstandings in source code," in *Proceedings of the Foundations of Software Engineering*, 2017, pp. 129–139.

[5] F. Medeiros, G. Lima, G. Amaral, S. Apel, C. Kästner, M. Ribeiro, and R. Gheyi, "An investigation of misunderstanding code patterns in C open-source software projects," *Empirical Software Engineering*, vol. 24, no. 4, pp. 1693–1726, 2019.

[6] J. Siegmund, "Program comprehension: Past, present, and future," in *Proceedings of the Software Analysis, Evolution, and Reengineering*, vol. 5, 2016, pp. 13–20.

[7] K. Narasimhan and C. Reichenbach, "Copy and paste redeemed (T)," in *International Conference on Automated Software Engineering*. IEEE/ACM, 2015, pp. 630–640.

[8] F. Medeiros, C. Kästner, M. Ribeiro, S. Nadi, and R. Gheyi, "The love/hate relationship with the C preprocessor: An interview study," in *Proceedings of the 29th European Conference on Object-Oriented Programming (ECOOP'15)*. Schloss Dagstuhl, 2015, pp. 999–1022.

[9] S. Schulze, J. Liebig, J. Siegmund, and S. Apel, "Does the discipline of preprocessor annotations matter?: A controlled experiment," in *Proceedings of 12th International Conference on Generative Programming: Concepts and Experiences (GPCE)*, 2013, pp. 65–74.

[10] A. Von Mayrhauser and A. M. Vans, "Program comprehension during software maintenance and evolution," *Computer*, vol. 28, no. 8, pp. 44–55, 1995.

[11] V. Rajlich and N. Wilde, "The role of concepts in program comprehension," in *Proceedings 10th International Workshop on Program Comprehension*, 2002, pp. 271–278.

[12] A. Lakhotia, "Understanding someone else's code: An analysis of experience", the," *Journal of Systems and Software*, pp. 269–275, 1993.

[13] A. Gopstein, Fayard and Cappos, "Thinking aloud about confusing code: A qualitative investigation of program comprehension and atoms of confusion," 11 2020.

[14] P. Gordon, R. Hendrick, M. Johnson, and Y. Lee, "Similarity-based interference during language comprehension: Evidence from eye tracking during reading." 2006, pp. 1304–1321.

[15] R. Bednarik and M. Tukiainen, "An eye-tracking methodology for characterizing program comprehension processes," in *Proceedings of the Symposium on Eye Tracking Research &Amp; Applications*, 2006, pp. 125–132.

[16] A. Jbara and D. G. Feitelson, "How programmers read regular code: a controlled experiment using eye tracking," *Empirical software engineering*, vol. 22, no. 3, pp. 1440–1477, 2017.

[17] Z. Sharafi, T. Shaffer, B. Sharif, and Y.-G. Guéhéneuc, "Eye-tracking metrics in software engineering," in *Proceedings of the Asia-Pacific Software Engineering Conference*, 2015, pp. 96–103.

[18] M. Just and P. Carpenter, "A theory of reading: From eye fixations to comprehension. psychological review," 1965, pp. 329–354.

[19] T. Busjahn, R. Bednarik, A. Begel, M. Crosby, J. H. Paterson, C. Schulte, B. Sharif, and S. Tamm, "Eye movements in code reading: Relaxing the linear order," in *Proceedings of the International Conference on Program Comprehension*, 2015, pp. 255–265.

[20] J. Melo, F. Narcizo, D. Hansen, C. Brabrand, and A. Wasowski, "Variability through the eyes of the programmer," in *Proceedings of the International Conference on Program Comprehension*, 2017, pp. 34–44.

[21] G. E. P. Box, J. S. Hunter, and W. G. Hunter, *Statistics for Experimenters: Design, innovation, and discovery.* Wiley-Interscience, 2005.

[22] M. Ribeiro, P. Borba, and C. Kästner, "Feature maintenance with emergent interfaces," in *Proceedings of the International Conference on Software Engineering*, 2014, pp. 989–1000.

[23] R. Malaquias, M. Ribeiro, R. Bonifácio, E. Monteiro, F. Medeiros, A. Garcia, and R. Gheyi, "The discipline of preprocessor-based annotations does #ifdef tag n't #endif matter," in *Proceedings of the International Conference on Program Comprehension*, 2017, pp. 297–307.

[24] D. C. Montgomery, *Design and Analysis of Experiments*. John Wiley & Sons, 2006.

[25] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in software engineering*. Springer Science & Business Media, 2012.

[26] D. Sjoberg, B. Anda, E. Arisholm, T. Dyba, M. Jorgensen, A. Karahasanovic, E. F. Koren, and M. Vokác, "Conducting realistic experiments in software engineering," in *Proceedings of the International Symposium on Empirical Software Engineering*, 2002, pp. 17–26.

[27] C. Schulte, T. Clear, A. Taherkhani, T. Busjahn, and J. H. Paterson, "An introduction to program comprehension for computer science educators," in *Proceedings of the ITiCSE Working Group Reports*, 2010, pp. 65–86.

[28] L. A. Sudol-DeLyser, M. Stehlik, and S. Carver, "Code comprehension problems as learning events," in *Proceedings of the Innovation and Technology in Computer Science Education*, 2012, pp. 81–86.

[29] J. Melo, C. Brabrand, and A. Wkasowski, "How does the degree of variability affect bug finding?" in *Proceedings of the International Conference on Software Engineering*, 2016, pp. 679–690.

[30] F. Castor, "Identifying confusing code in swift programs," 09 2018.

[31] B. T. Carter and S. G. Luke, "Best practices in eye tracking research," *International Journal of Psychophysiology*, vol. 155, pp. 49 – 62, 2020. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167876020301458

[32] A. T. Duchowski, *Eye Tracking Methodology: Theory and Practice*, 3rd ed. Springer Publishing Company, Incorporated, 2017.

[33] A. Begel and H. Vrzakova, "Eye movements in code review," in *Proceedings of the Workshop on Eye Movements in Programming*, ser. EMIP '18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3216723.3216727